GraphGuard: Private Time-Constrained Pattern Detection Over Streaming Graphs in the Cloud

Songlei Wang and Yifeng Zheng, *Harbin Institute of Technology;*
Xiaohua Jia, *Harbin Institute of Technology and City University of Hong Kong*

https://www.usenix.org/conference/usenixsecurity24/presentation/wang-songlei

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

GraphGuard: Private Time-Constrained Pattern Detection Over Streaming Graphs in the Cloud

Songlei Wang[1], Yifeng Zheng[1,*], Xiaohua Jia[1,2]
[1]*School of Computer Science and Technology, Harbin Institute of Technology, Shenzhen*
[2]*Department of Computer Science, City University of Hong Kong*

Abstract

Streaming graphs have seen wide adoption in diverse scenarios due to their superior ability to capture temporal interactions among entities. With the proliferation of cloud computing, it has become increasingly common to utilize the cloud for storing and querying streaming graphs. Among others, streaming graphs-based time-constrained pattern detection, which aims to continuously detect subgraphs matching a given query pattern within a sliding time window, benefits various applications such as credit card fraud detection and cyber-attack detection. Deploying such services on the cloud, however, entails severe security and privacy risks. This paper presents GraphGuard, the *first* system for privacy-preserving outsourcing of time-constrained pattern detection over streaming graphs. GraphGuard is constructed from a customized synergy of insights on graph modeling, lightweight secret sharing, edge differential privacy, and data encoding and padding, safeguarding the confidentiality of edge/vertex labels and the connections between vertices in the streaming graph and query patterns. We implement and evaluate GraphGuard on several real-world graph datasets. The evaluation results show that GraphGuard takes only a few seconds to securely process an encrypted query pattern over an encrypted snapshot of streaming graphs within a time window of size 50,000. Compared to a baseline built on generic secure multiparty computation, GraphGuard achieves up to 60× improvement in query latency and up to 98% savings in communication.

1 Introduction

Streaming graphs are constantly evolving graphs, where the vertices and edges of the graphs change over time. The use of streaming graphs has seen great popularity in various scenarios (e.g., social media, computer networks, and financial transactions [27]), because of its excellent ability to characterize the complex temporal interactions among entities. Due to the well-understood benefits of cloud computing [23], utilizing the cloud to store and query streaming graph databases has become increasingly common (e.g., [1, 34]). However, such service outsourcing can put the privacy of information-rich streaming graph data at risk [37, 44]. It is thus crucial to bring privacy assurance into such cloud-empowered service paradigm from the beginning, protecting the outsourced streaming graphs, the queries, and the query results.

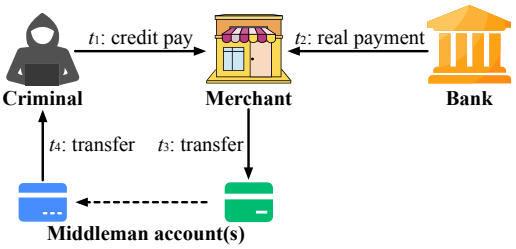


Figure 1: An example credit card fraud pattern.

As one of the most popular streaming graph search functionalities, time-constrained pattern detection, which is the focus in this paper, aims to continuously detect subgraphs over streaming graph data that are isomorphic to a given query pattern and whose edge occurrence orders adhere to the timing order constraints specified by the query pattern [27]. Note that as the subgraph isomorphism problem is NP-complete [12], plaintext-domain algorithms [27, 35] typically perform the detection on the edges/vertices falling within the most recent time units (as specified by a sliding time window), so as to make the problem solvable in a reasonable time. In this paper, we also adhere to this practice. Time-constrained pattern detection can benefit various applications, e.g., credit card fraud detection [35] and cyber-attack detection [11]. The following example demonstrates its practical usefulness.

Example 1. *Fig. 1 illustrates a credit card fraud pattern. Here, a criminal launches a scheme involving a merchant and a series of middleman accounts under its control to unlawfully withdraw funds [35]. Utilizing fake IDs, the criminal may obtain a credit from the bank. The criminal tries to ille-*

*Corresponding author.

gally cash out money by faking a purchase with the help of a merchant at time t_1. Upon receiving payment from the bank at time t_2, the merchant tries to send the money back to the criminal via a series of middleman accounts from time t_3 to time t_4. This particular pattern is characterized by the specific order of transactions w.r.t. time ($t_1 < t_2 < t_3 < t_4$), and it can be conveniently modeled as a query pattern with timing order constraints. If the system can detect the pattern timely, it then becomes possible to thwart such fraudulent activities.

In the literature, considerable efforts have been invested in privacy-preserving query processing over graphs. However, previous works primarily focus on privately querying *static* graphs, such as private subgraph matching [39, 42], private shortest path search [17, 19], and private breadth-first search [4, 7]. To the best of our knowledge, no prior studies have explored privacy-preserving time-constrained pattern detection over outsourced streaming graphs.

In this paper, we design, implement, and evaluate GraphGuard, a new system enabling privacy-preserving outsourcing of time-constrained pattern detection over streaming graphs. GraphGuard aims to allow the cloud to obliviously manage a constantly evolving streaming graph and provide time-constrained pattern detection services. Following the emerging trend of leveraging distributed trust in security designs [6, 16, 38] as well as in industrial applications [2, 31], GraphGuard employs a three-server secure computation architecture, where three cloud servers hosted by independent service providers work collaboratively to empower the secure service. We then build GraphGuard from a customized synergy of insights on graph modeling, lightweight secure computation [3], edge differential privacy [22], and data encoding and padding strategies. GraphGuard protects the confidentiality of labels of edges/vertices and hides the connections between vertices in the streaming graph and query patterns.

Below we describe the challenges that arise in designing GraphGuard and our key ideas in developing the solutions.

Challenge 1: How to model the streaming graph to enable secure updates and graph isomorphism check at the cloud? Unlike secure outsourced computation for *static* graphs, where the graph owner can conceal vertex connections locally, the dynamic nature of streaming graphs means that updates, if not treated carefully, can easily leak vertex connections. For example, if the cloud servers observe that the numbers of neighboring vertices for both vertices v_1 and v_2 increase by 1 simultaneously, they can infer that a new edge emerges between them. To address this challenge, we depart from the commonly employed posting list structure [14] for graph modeling and instead resort to the edge list structure [32]. Consequently, individual encryption of each edge is performed in GraphGuard, facilitating the secure addition (resp. removal) of a new (resp. outdated) edge. In addition, to enable efficient secure graph isomorphism check, we propose the concept of "endpoint adjacency matrix (EAM)" to model the graph structure. With EAM, checking the isomorphism between two graphs can be simplified as the comparison between their EAMs, which only needs basic XOR and AND operations. This greatly facilitates the efficient realization of graph isomorphism checking in the ciphertext domain.

Challenge 2: How to enable the cloud servers to efficiently and obliviously obtain the matched edges? To detect the subgraphs in the streaming graph that are isomorphic to the query pattern, the cloud servers first need to obtain the edges (i.e., matched edges) whose labels are identical to those in the query pattern. The generic secure multiparty computation (MPC)-based secure equality test protocols (e.g., [33] and [9]) are natural tools. However, these protocols typically require substantial online communication and offline preparation, both of which scale linearly with the number of secure equality tests. We instead propose to layer one-hot data encoding with replicated secret sharing [3], and devise a custom mechanism for securely producing the encrypted equality test result, which *eliminates the need for any online communication or offline preparation*. We further propose an oblivious dummy edges padding protocol to obfuscate the locations of the true edges, so as to enable the cloud servers to obliviously obtain the matched edges.

Challenge 3: How to enable the cloud servers to efficiently and securely process encrypted timing order constraints specified by the query pattern? Different from subgraph matching over *static* graphs, time-constrained pattern detection is much more challenging as it requires that the (dynamic) subgraphs isomorphic to the query pattern must also comply with specific timing order constraints. To securely evaluate the temporal order consistency of edges, the generic MPC-based secure comparison protocols (e.g., [30] and [8]) are natural tools. However, these protocols also require substantial online communication and offline preparation, both of which scale linearly with the number of secure comparisons. Our insight is to first decompose the query pattern into several timing-connected subquery patterns [27]. Then we design a tailored mechanism for modeling and encrypting the timing order constraints, which enables the cloud servers to securely evaluate the temporal order consistency of edges *without the need for any online communication or offline preparation*.

Contributions. We highlight our contributions below:

- We initiate the *first* study on privacy-preserving outsourcing of time-constrained pattern detection over streaming graphs, and design a tailored secure system GraphGuard.
- We propose a new data structure EAM and show how to model and encrypt the streaming graph and time-constrained query patterns to facilitate secure time-constrained pattern detection over streaming graphs.
- On the basis of custom modeling and encryption strategies, we develop techniques to support secure partial matches detection and secure partial matches compatibility checking.
- We provide formal security analysis, make a GPU-accelerated prototype implementation, and conduct extensive experiments over real-world graph datasets. The results show

that GraphGuard takes only a few seconds to securely perform one graph pattern detection, under a window size of 50,000. Compared with a baseline simply using the generic and popular MPC framework MP-SPDZ [24], GraphGuard achieves up to 60× improvement in query latency and up to 98% savings in communication cost.

2 Preliminaries

2.1 Time-Constrained Pattern Detection

***Definition* 1.** *(Streaming graph [27]). A streaming graph $\mathbb{G}$ is a dynamic (monotonically increasing) set of directed and labeled edges $\{e_x\}_{x\in[X]}$, where X represents the total count of edges that have emerged over time. Each edge $e_x = (sid_x, eid_x, l_x, t_x)$, which indicates that e_x appears at time t_x (named as timestamp) and is an edge with label l_x that connects the vertex with ID sid_x to the vertex with ID eid_x. Two edges are deemed connected if they share common vertices.*

In this paper, we write $[S]$ for the set $\{1, 2, \cdots, S\}$, use $\{a_i\}_{i\in[S]}$ to represent the set $\{a_1, \cdots, a_S\}$, and omit the subscript $i \in [S]$ when the context is clear. We refer to the pair of vertices that an edge connects as the edge's *endpoints*. Additionally, the labels of vertices can be implicitly indicated by the edge labels. For example, in a financial transaction network, we can use "001" to label the transactions (i.e., edges) with type "credit pay" from an entity (i.e., vertex) with type "credit card" to an entity with type "restaurant", and use "002" to label the transactions with type "credit pay" from an entity with type "credit card" to an entity with type "supermarket". Graph pattern detection over a streaming graph operates on each snapshot individually, which can be defined as follows.

***Definition* 2.** *(Snapshot [27]). Given the time window size W and the current time point t, the current snapshot $\mathbb{G}_t$ of the streaming graph $\mathbb{G}$ is a subgraph of $\mathbb{G}$ that includes edges with timestamps falling within time interval $(t-W, t]$ and the vertices that are adjacent to these edges.*

***Definition* 3.** *(Query pattern [27]). A query pattern $\mathbb{Q}$ is defined by the query graph $\{\sigma_y\}_{y\in[Y]}$ and the timing order constraints $\{\sigma_m \prec \sigma_n\}$. Here, Y represents the number of edges in $\mathbb{Q}$, and $\sigma_y = (sid_y, eid_y, l_y)$ is an edge with label l_y that connects the vertex with ID sid_y to the vertex with ID eid_y. $\sigma_m \prec \sigma_n$ means that in a match g for $\mathbb{Q}$ where edge e_i matches edge σ_m and edge e_j matches edge σ_n ($e_i, e_j \in g$), e_i should not appear before e_j (denoted as $e_i \prec e_j$).*

Time-constrained pattern detection aims to identify *all time-constrained matches* of a query pattern over each snapshot. A time-constrained match can be defined as follows.

***Definition* 4.** *(Time-constrained match [27]). Given query pattern $\mathbb{Q}$ and snapshot $\mathbb{G}_t$, a subgraph $g \in \mathbb{G}_t$ is a time-constrained match of $\mathbb{Q}$ if there exists a bijective function $f(\cdot)$ from $\mathbb{Q}$'s vertices $V(\mathbb{Q})$ to g's vertices $V(g)$ such that the following conditions hold: (1) isomorphism: the label of the edge between each pair of connected vertices $v_i, v_j \in V(\mathbb{Q})$ is identical to that between $f(v_i), f(v_j) \in V(g)$, and vice versa; (2) timing order consistency: considering each timing order constraint $\sigma_m \prec \sigma_n$ and given that edges $e_i, e_j \in g$ respectively match edges σ_m and σ_n, $e_i \prec e_j$ holds.*

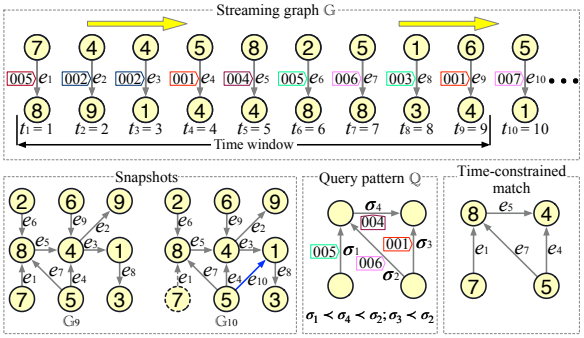


Figure 2: Illustration of time-constrained pattern detection over a streaming graph under the time window of size 9.

Example 2. *For clarity, we illustrate in Fig. 2 a streaming graph $\mathbb{G}$ along with its snapshots $\mathbb{G}_9$ and $\mathbb{G}_{10}$ under a time window of size 9, a query pattern $\mathbb{Q}$, and its time-constrained match. $\mathbb{G}$ contains edges with 7 different labels: $\{001, \cdots, 007\}$. At time point $t = 10$, edge e_1 expires and edge e_{10} arrives since the time window W is $(1, 10]$, which forms a new snapshot $\mathbb{G}_{10}$. In the figure, the expired edge is shown in dotted line while the newly arrived edge is shown in blue line. $\mathbb{Q}$ consists of four edges with labels $\{\sigma_1 : 005, \sigma_2 : 006, \sigma_3 : 001, \sigma_4 : 004\}$ and timing order set $\{\sigma_1 \prec \sigma_4 \prec \sigma_2; \sigma_3 \prec \sigma_2\}$. The single time-constrained match of $\mathbb{Q}$ formed by edges $\{e_1, e_7, e_4, e_5\}$ in $\mathbb{G}_9$, where e_1 matches σ_1, e_7 matches σ_2, e_4 matches σ_3, and e_5 matches σ_4.*

For clarity, we give the plaintext-domain graph pattern detection process underlying the security design of GraphGuard in Appendix A.

2.2 Replicated Secret Sharing

Replicated secret sharing (RSS) [3] divides a private value x in the ring $\mathbb{Z}_{2^l}$ into three shares $\langle x\rangle_1, \langle x\rangle_2, \langle x\rangle_3 \in \mathbb{Z}_{2^l}$, where $x = \langle x\rangle_1 + \langle x\rangle_2 + \langle x\rangle_3$. Three pairs of shares $(\langle x\rangle_1, \langle x\rangle_2)$, $(\langle x\rangle_2, \langle x\rangle_3)$ and $(\langle x\rangle_3, \langle x\rangle_1)$ are then distributed to three parties P_1, P_2 and P_3, respectively[1]. If $l = 1$, the secret sharing is called *binary sharing*, and otherwise *arithmetic sharing*. In this paper, we use $[\![\cdot]\!]$ to represent an encrypted entity. For the sake of presentation, we use $i \pm 1$ to indicate the next(+)/previous(-) party (or secret share) with wrap around, i.e., P_{3+1} indicates P_1, P_{1-1} indicates P_3, $\langle x\rangle_{3+1}$ indicates $\langle x\rangle_1$, and $\langle x\rangle_{1-1}$ indicates $\langle x\rangle_3$. With this, we can use $(\langle x\rangle_i, \langle x\rangle_{i+1})$ to denote the shares held by $P_i, i \in \{1, 2, 3\}$. In the binary RSS

[1] In this paper, we refer to the secret sharing process as "encryption".

domain, the basic operations are as follows (all operations are within the ring $\mathbb{Z}_2$). (1) To compute $[\![h]\!] = [\![x \oplus y]\!]$ given two secret-shared values $[\![x]\!]$ and $[\![y]\!]$, each party $P_i, i \in \{1,2,3\}$ computes $\langle h \rangle_i = \langle x \rangle_i \oplus \langle y \rangle_i$ and $\langle h \rangle_{i+1} = \langle x \rangle_{i+1} \oplus \langle y \rangle_{i+1}$ locally. (2) To compute $[\![z]\!] = [\![x \otimes y]\!]$ given two secret-shared values $[\![x]\!]$ and $[\![y]\!]$, each party $P_i, i \in \{1,2,3\}$ first computes $\langle z \rangle_i = \langle x \rangle_i \otimes \langle y \rangle_i \oplus \langle x \rangle_i \otimes \langle y \rangle_{i+1} \oplus \langle x \rangle_{i+1} \otimes \langle y \rangle_i$ locally. However, this produces z in 3-out-of-3 additive secret sharing (i.e., each P_i holds $\langle z \rangle_i$ and $z = \langle z \rangle_1 \oplus \langle z \rangle_2 \oplus \langle z \rangle_3$) instead of 2-out-of-3 RSS. To facilitate subsequent computations in the RSS domain, a re-sharing operation is required.

2.3 Differential Privacy for Graphs

Differential privacy (DP) [18] guarantees that the outcomes or insights derived from the analysis conducted on two neighboring datasets, which differ by the inclusion or exclusion of a single individual's data, are nearly indistinguishable. When applying DP to the domain of graph analysis, it becomes necessary to establish a notion of "neighboring graphs". Previous research [22, 43] in this field commonly considers two graphs to be neighboring if they differ by the presence or absence of a single edge. In other words, given a graph $\mathbb{G}$, its neighboring graph $\mathbb{G}'$ can be obtained by adding or deleting a single edge from $\mathbb{G}$. This definition is known as *edge DP* [22]. The formal definition of edge DP is as follows.

Definition 5. *(Edge DP [22]). A randomized mechanism $\mathcal{M}$ with domain $\mathcal{G}$ provides (ε, δ)-edge DP, if and only if for any two neighboring graphs $\mathbb{G}, \mathbb{G}' \in \mathcal{G}$ that differ in one edge:*

$$\forall \hat{\mathbb{G}} \in Range(\mathcal{M}), Pr[\mathcal{M}(\mathbb{G}) = \hat{\mathbb{G}}] \leq e^{\varepsilon} \cdot Pr[\mathcal{M}(\mathbb{G}') = \hat{\mathbb{G}}] + \delta,$$

where $Range(\mathcal{M})$ denotes the set of all possible outputs of $\mathcal{M}$, ε is the privacy budget, and δ is a privacy parameter.

Discrete Laplace distribution is widely used to draw discrete noises for providing DP, which is defined as follows [20].

Definition 6. *(Discrete Laplace distribution [20]). A discrete random variable $x \in \mathbb{Z}_{2^l}$ follows discrete Laplace distribution $Lap(\varepsilon, \delta, \Delta)$ if its probability mass function is*

$$Pr[x] = \frac{e^{\frac{\varepsilon}{\Delta}} - 1}{e^{\frac{\varepsilon}{\Delta}} + 1} \cdot e^{\frac{-\varepsilon \cdot |x-\mu|}{\Delta}}, \forall x \in \mathbb{Z}_{2^l},$$

where μ is the mean of the distribution and Δ is the sensitivity.

In the context of graph, a query Q's sensitivity Δ is the maximum L_1 norm between Q's outputs on any two neighboring graphs $\mathbb{G}$ and $\mathbb{G}'$:

$$\Delta = \max_{\mathbb{G}, \mathbb{G}'} \|Q(\mathbb{G}) - Q(\mathbb{G}')\|_1$$

This measures the maximum impact that any individual edge in the input graph can have on the output of query Q.

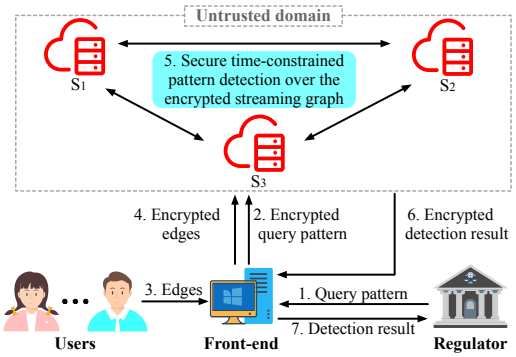


Figure 3: The system architecture of GraphGuard.

3 Problem Statement

3.1 System Architecture

Fig. 3 shows GraphGuard's system architecture, which consists of three types of entities: the user, the on-premise service front-end (denoted as FE) dedicatedly maintained by the regulator, and the cloud server. The regulator, such as a bank or a cybersecurity center, aims to continuously detect specific time-constrained subgraph patterns over the constantly evolving streaming graph that its users generate via the FE.

The regulator is attracted by the benefits of cloud computing, such as scalability and flexibility, cost savings, ubiquitous and on-demand network access, and relief of the burden for storage management [23]. Therefore, it wants to leverage the power of cloud computing to manage the streaming graph and alert it when the specified patterns occur in the streaming graph[2]. In practice, such cloud-based service paradigm has been widely adopted in various streaming graph search systems (e.g., [1, 34]). However, due to privacy concerns regarding the proprietary streaming graph and query patterns, it is essential to embed security in such outsourced services from the very beginning to protect the streaming graph, the query patterns, and the detection results.

GraphGuard adopts a distributed trust architecture, where the power of the cloud providing the secure graph pattern detection service is split into three servers (denoted as $\mathsf{S}_{123} = \{\mathsf{S}_1, \mathsf{S}_2, \mathsf{S}_3\}$) which can be operated by independent commercial cloud service providers in practice. Such distributed trust architecture has been widely adopted in other security designs [6, 16, 38, 40] as well as in industry [2, 31].

3.2 Threat Model and Security Guarantees

Threat model. Similar to prior works that adopt the multi-server setting for security designs [6, 38, 40], we consider a semi-honest and non-colluding adversary model where each of S_{123} honestly follows our protocol, but may *individually* try to deduce the private information during providing pattern

[2]We provide evaluation in Appendix B to demonstrate the computational benefit for the regulator from such graph service outsourcing.

detection services. Besides, with the focus on protecting the privacy of the streaming graph and query pattern against the cloud servers, we consider that the regulator, FE, and users are trustworthy parties since the regulator can be an official institution (e.g. a bank) and FE is dedicatedly maintained by the regulator [26], which can examine the users' behaviors.

Security guarantees. Under the aforementioned adversary model, GraphGuard guarantees that each cloud server only learns the following information and nothing more. (1) The differentially private frequency of distinct edge labels in each snapshot. (2) The occurrence of a pattern match in the streaming graph. This requirement is inherent for servers to provide the (secure) service (i.e., securely detecting pattern matches to alert the regulator); otherwise, the service would be rendered useless. (3) The count of edges in the streaming graph and query patterns, as well as the timestamp associated with each edge. Here we consider the timestamps as public because the cloud servers can infer them based on each edge's upload time. As for the count of edges, we are not aware of any concrete harm from revealing such size information. We discuss how to mitigate these leakages in Appendix C.

4 The Design of GraphGuard

Overview. GraphGuard consists of four phases: (1) streaming graph encryption (Section 4.1), (2) query pattern encryption (Section 4.2), (3) secure partial matches detection (Section 4.3), and (4) secure partial matches compatibility checking (Section 4.4). In phase (1), FE continuously encrypts the edges generated by the users, and uploads the resulting ciphertext to S_{123}. In phase (2), FE encrypts the query pattern and uploads the resulting ciphertext to S_{123}. Afterwards, S_{123} continuously perform secure time-constrained pattern detection on the encrypted streaming graph. Specifically, in phase (3), S_{123} securely detect time-constrained partial matches of the query pattern over the encrypted edges in the sliding window. In phase (4), S_{123} securely combine the compatible partial matches to produce the final encrypted matches. If any matches are detected, they are returned to the regulator for decryption. It is noted that similar to the plaintext-domain works [27, 35], the current design of GraphGuard assumes that the streaming graph monotonically increases. We discuss how to securely handle vertex/edge deletion in Appendix D.

4.1 Streaming Graph Encryption

We first introduce how a newly generated edge is encrypted in GraphGuard. In order to facilitate secure updates of the encrypted streaming graph, we adopt the edge list structure [32] to model the streaming graph. Specifically, a newly generated edge is modeled as $e_x = (sid_x, eid_x, l_x, t_x)$ as definition 1. Only sid_x, eid_x, and l_x need to be encrypted, as the timestamp t_x is considered public. To allow lightweight encryption as well as support subsequent secure computation, GraphGuard has FE apply the RSS technique [3] over each private value. Through a detailed analysis of time-constrained subgraph pattern detection [27], we observe that *equality test* plays a key role, and thus the performance of its secure realization is crucial. In order to support efficient secure equality test, GraphGuard does not directly use arithmetic RSS to encrypt each value.

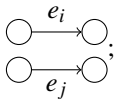
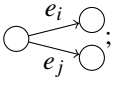
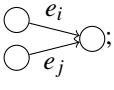
Instead, GraphGuard introduces a preprocessing step in which each private value is encoded as a *one-hot vector* of length equal to the number of all possible values for the private value. In a one-hot vector, all elements are set to "0", except for the element at the location corresponding to the value, which is set to "1". To simplify presentation, we abuse the symbol ρ to represent the length of all one-hot vectors in this paper. GraphGuard applies binary RSS to encrypt these one-hot vectors. So an edge e_x is encrypted into $[\![e_x]\!] = ([\![\mathbf{sid}_x]\!], [\![\mathbf{eid}_x]\!], [\![\mathbf{l}_x]\!], t_x)$, where a one-hot vector is written in bold. How this can benefit efficient secure equality test in the RSS domain will be clear later in Section 4.3.1. FE continuously sends the encrypted edges to S_{123}, who hold the encrypted graph as $[\![\mathbb{G}]\!] = \{[\![e_x]\!]\}_{x\in[X]}$.

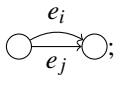
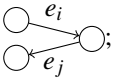
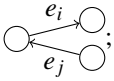
4.2 Query Pattern Encryption

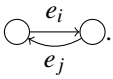
Modeling the structure. We start with considering how to model the structure of the query pattern. Note that the essential step of graph pattern detection is to check the isomorphism between the query pattern and a candidate subgraph in a snapshot. To realize this, a common strategy adopted in the plaintext-domain works is to find the bijective match function (as per Definition 4) by constructing the search tree *along the connections between vertices* [29]. However, for secure graph pattern detection, information about the vertex connections must be kept private. This makes it difficult to directly follow the above strategy in the ciphertext domain. We instead propose a new data structure - *endpoint adjacency matrix* (EAM) - to model vertex connections. With EAM, checking the isomorphism between the query pattern and a candidate subgraph can be simplified as the comparison between their EAMs, consisting of only basic $\oplus$ and $\otimes$ operations. Such modeling greatly facilitates the efficient realization of graph isomorphism checking in the ciphertext domain.

***Definition* 7.** *(Endpoint adjacency matrix). Given a graph structure $\{e_y = (sid_y, eid_y)\}_{y\in[Y]}$, its EAM is a 4-bit matrix $\mathbf{M} \in \{0000, 1000, 0100, 1100, 0010, 0001, 0011\}^{Y\times Y}$. Each row/column of $\mathbf{M}$ corresponds to an edge, and each element $\mathbf{M}[i,j], i,j \in [Y], i \neq j$ indicates whether the edges e_i and e_j share the endpoints. Specifically, the first bit of $\mathbf{M}[i,j]$ is equal to 1 if and only if $sid_i = sid_j$; the second bit is equal to 1 if and only if $eid_i = eid_j$; the third bit is equal to 1 if and only if $eid_i = sid_j$; and the fourth bit is equal to 1 if and only if $sid_i = eid_j$. In addition, $\mathbf{M}[i,i] = 0000, i \in [Y]$.*

For clarity, the structure of the given two edges e_i, e_j corresponding to different values of $\mathbf{M}[i,j]$ is illustrated as follows.

(1) 0000:  ; (2) 1000:  ; (3) 0100:  ;

(4) 1100:  ; (5) 0010:  ; (6) 0001:  ;

(7) 0011:  .

Example 3. *Consider the query $\mathbb{Q}$ in Fig. 2, its EAM is*

$$\begin{bmatrix} 0000 & 0100 & 0000 & 0010 \\ 0100 & 0000 & 1000 & 0010 \\ 0000 & 1000 & 0000 & 0100 \\ 0001 & 0001 & 0100 & 0000 \end{bmatrix}.$$

With EAM, we can simplify the isomorphism checking between the query pattern and a subgraph as follows.

Proposition 1. *Given a query pattern $\mathbb{Q} = \{\sigma_i\}_{i\in[n]}$ and its EAM $\mathbf{M}_1$; a candidate match $P = \{e_i\}_{i\in[n]}$ and its EAM $\mathbf{M}_2$, where edge e_i matches edge $\sigma_i, i \in [n]$. If $\mathbf{M}_1 = \mathbf{M}_2$, P is isomorphic to $\mathbb{Q}$; if $\mathbf{M}_1 \neq \mathbf{M}_2$, P is not isomorphic to $\mathbb{Q}$.*

Proposition 1 holds because $\mathbf{M}_1 = \mathbf{M}_2$ means that $\forall i, j \in [n]$, $\mathbf{M}_1[i,j] = \mathbf{M}_2[i,j]$. This indicates that $\forall i, j \in [n]$, the connection relationship between (σ_i, σ_j) in $\mathbb{Q}$ is identical to that between (e_i, e_j) in P. Instead, $\mathbf{M}_1 \neq \mathbf{M}_2$ means that $\exists i, j \in [n], \mathbf{M}_1[i,j] \neq \mathbf{M}_2[i,j]$. This indicates that $\exists i, j \in [n]$, the connection relationship between (σ_i, σ_j) is different from that between (e_i, e_j). Note that with Proposition 1, the subgraph isomorphism checking can be simplified as comparing the EAMs of graphs. The comparison of two EAMs $\mathbf{M}_1$ and $\mathbf{M}_2$ can be completed using $\oplus$ and $\otimes$ operations:

$$\omega = \bigvee_{i=1}^{n} \bigvee_{j=1}^{n} \bigvee_{s=1}^{4} \mathbf{M}_1[i,j][s] \oplus \mathbf{M}_2[i,j][s], \tag{1}$$

where $\mathbf{M}[i,j][s], s \in [4]$ denotes the s-th bit of $\mathbf{M}[i,j]$. The OR ($\vee$) operation on two bits b_1 and b_2 can be realized by $b_1 \vee b_2 = b_1 \otimes b_2 \oplus b_1 \oplus b_2$. Here, the bit $\omega = 0$ indicates that $\mathbf{M}_1 = \mathbf{M}_2$, and $\omega = 1$ indicates that $\mathbf{M}_1 \neq \mathbf{M}_2$.

Representing the timing order constraints. We now introduce the method for representing the timing order constraints of the query. FE first decomposes query $\mathbb{Q}$ into several *timing-connected subquery patterns* [27] (referred to as TC-subquery pattern) $\mathbb{Q} := \{\mathbb{Q}_d\}_{d\in[D]}$.

***Definition* 8.** *(TC-subquery pattern [27]). A subquery pattern of κ edges is a TC-subquery pattern, if there exists a permutation of all the edges such that $\sigma_1 \prec \cdots \prec \sigma_\kappa$.*

The decompositions of $\mathbb{Q}$ can be represented as $\mathcal{Y} = \{\mathbf{y}_d\}_{d\in[D]}$. Each $\mathbf{y}_d$ is a list used to represent the indexes of edges in a TC-subquery pattern $\mathbb{Q}_d$. More specifically,

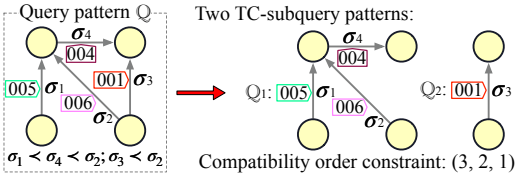


Figure 4: An example of query pattern decomposition.

$\mathbb{Q}_d = \{\sigma_{\mathbf{y}_d[i]}\}_{i\in[\kappa_d]}$ where $\sigma_{\mathbf{y}_d[1]} \prec \cdots \prec \sigma_{\mathbf{y}_d[\kappa_d]}$ and κ_d is $\mathbf{y}_d$'s length. Note that the decompositions may vary in length, and the decompositions of a query pattern are not unique. In addition, it is possible for timing order constraints to exist among the edges belonging to *different* TC-subquery patterns. These constraints are named as *compatibility order constraints* and denoted as $O = \{o_c\}_{c\in[C]}$, where C is their count. A compatibility order constraint $o_c = (y_1, y_2, \theta)$ indicates the presence of a timing order constraint between edges σ_{y_1} and σ_{y_2} that belong to different TC-subquery patterns. Here, $\theta = 1$ signifies $\sigma_{y_1} \prec \sigma_{y_2}$, while $\theta = 0$ signifies $\sigma_{y_2} \prec \sigma_{y_1}$.

Example 4. *Fig. 4 illustrates how the query pattern $\mathbb{Q}$ shown in Fig. 2 can be decomposed. In this case, $\mathbb{Q}$ is decomposed into two TC-subquery patterns: $\mathbb{Q}_1 = \{\sigma_1, \sigma_4, \sigma_2\}$ and $\mathbb{Q}_2 = \{\sigma_3\}$ (with corresponding decompositions $\mathbf{y}_1 = [1,4,2]$ and $\mathbf{y}_2 = [3]$). Besides, there is a compatibility order constraint between them: $(3,2,1)$, i.e., $\sigma_3 \prec \sigma_2$.*

Encrypting the query pattern. FE then encrypts the private information contained in the query pattern $\mathbb{Q}$ as follows. Firstly, FE encrypts each bit of $\mathbb{Q}$'s EAM $\mathbf{M}$ by binary RSS, producing $[\![\mathbf{M}]\!]$. Secondly, FE encodes $\mathbb{Q}$'s each edge label $l_y, y \in [Y]$ into a one-hot vector $\mathbf{l}_y$, and then encrypts these one-hot vectors by binary RSS, producing $[\![\mathcal{L}]\!] = \{[\![\mathbf{l}_y]\!]\}_{y\in[Y]}$. Thirdly, FE encrypts the compatibility order constraints as $[\![O]\!] = \{[\![o_c]\!]\}_{c\in[C]}$, where $[\![o_c]\!] = (y_1, y_2, [\![\theta]\!])$. Note that FE does not encrypt the edge indexes y_1, y_2 of each o_c since they only reveal the existence of timing order constraints between the edges $\sigma_{y_1}, \sigma_{y_2}$, but do not reveal the exact timing order constraint. In addition, FE also does not encrypt the decompositions $\mathcal{Y}$, as they only reflect the edge indexes that are not relevant to the private information of $\mathbb{Q}$. Specifically, since FE has the freedom to assign indexes to the edges in $\mathbb{Q}$ and the security of RSS [3] guarantees that the secret shares of each encrypted edge are indistinguishable from uniformly random values, $\mathcal{Y}$ do not reveal any private information about $\mathbb{Q}$. Finally, the ciphertext of $\mathbb{Q}$ is represented as $[\![\mathbb{Q}]\!] = ([\![\mathbf{M}]\!], [\![\mathcal{L}]\!], [\![O]\!], \mathcal{Y})$.

4.3 Secure Partial Matches Detection

Overview. Upon receiving the encrypted query pattern from FE, S_{123} continuously detect encrypted time-constrained matches of the query pattern over each encrypted snapshot (denoted as $[\![\mathbb{G}_t]\!]$). Note that S_{123} can retrieve the encrypted snapshot from the encrypted streaming graph by checking

Subroutine 1 Oblivious Dummy Edges Padding

Input: A snapshot $[\![\mathbb{G}_t]\!]$; privacy budget ε and parameter δ.
Output: Encrypted snapshot $[\![\hat{\mathbb{G}}_t]\!]$ after oblivious padding.

1: S_1: $\hat{\mathcal{E}}_1 \leftarrow \emptyset$. //`Initialize the dummy edge set.`
2: **for** each possible edge label l **do**
3: $\quad\mathsf{S}_1$: $n_l \leftarrow \max(Lap(\varepsilon,\delta,1),0)$.
4: $\quad$**for all** $x \in [n_l]$ **do**
5: $\quad\quad\mathsf{S}_1$: $t_x \leftarrow \mathsf{randTime}()$.
6: $\quad\quad\mathsf{S}_1$: $\hat{\mathcal{E}}_1.\mathsf{add}((-\mathbf{1},-\mathbf{1},\mathbf{l},t_x))$.
7: $\quad$**end for**
8: **end for**
9: S_1:$[\![\hat{\mathcal{E}}_1]\!] \leftarrow \mathsf{secShare}(\hat{\mathcal{E}}_1)$. //`Secretly share with` S_{23}.
10: S_2: Construct dummy edge set $\hat{\mathcal{E}}_2$ by the same process.
11: S_2:$[\![\hat{\mathcal{E}}_2]\!] \leftarrow \mathsf{secShare}(\hat{\mathcal{E}}_2)$.//`Secretly share with` S_{13}.
12: S_{123}: Integrate $[\![\hat{\mathcal{E}}_1]\!]$ and $[\![\hat{\mathcal{E}}_2]\!]$ into $[\![\mathbb{G}_t]\!]$.
13: S_{123}: Convert true edges' timestamps to the RSS format.
14: S_{123}: $[\![\hat{\mathbb{G}}_t]\!] \leftarrow \mathsf{olivSort}([\![\mathbb{G}_t]\!], \{[\![t_x]\!]\})$.
15: **return** Encrypted snapshot $[\![\hat{\mathbb{G}}_t]\!]$ after oblivious padding.

the public timestamp of each edge. Before performing the secure detection, S_{123} first perform a **Pre-processing** step to decompose the encrypted query pattern into encrypted TC-subquery patterns. Then the process of secure detection over an encrypted snapshot proceeds as follows. Firstly, given an encrypted TC-subquery pattern, S_{123} securely detect its encrypted matches (i.e., partial matches of the complete query pattern), which involves the components of secure candidate partial matches fetching (Section 4.3.1) and secure candidate partial matches filtering (Section 4.3.2). We will show how S_{123} can securely construct the final encrypted detection results based on the encrypted partial matches in Section 4.4.

Pre-processing. Recall that each decomposition $\mathbf{y}_d \in \mathcal{Y}$ consists of the indexes of edges of the corresponding TC-subquery pattern $\mathbb{Q}_d$. Therefore, S_{123} can easily obtain each encrypted TC-subquery pattern $[\![\mathbb{Q}_d]\!] = ([\![\mathcal{L}_d]\!], [\![\mathbf{M}_d]\!]), d \in [D]$ by: (1) $[\![\mathcal{L}_d]\!] = \{[\![\mathbf{l}_y]\!] | [\![\mathbf{l}_y]\!] \in [\![\mathcal{L}]\!], y \in \mathbf{y}_d\}$; (2) $[\![\mathbf{M}_d[i,j]]\!] = [\![\mathbf{M}[\mathbf{y}_d[i], \mathbf{y}_d[j]]]\!], i,j \in [\kappa_d]$. Here, $[\![\mathbf{M}_d]\!]$ is a permutation submatrix of $[\![\mathbf{M}]\!]$, comprised of the elements located at the intersection of rows $\mathbf{y}_d$ and columns $\mathbf{y}_d$ of $[\![\mathbf{M}]\!]$.

Example 5. *Referring to Example 4, since the decompositions* $\mathbf{y}_1 = [1,4,2]$ *and* $\mathbf{y}_2 = [3]$, $[\![\mathbb{Q}]\!]$ *is decomposed into* $[\![\mathbb{Q}_1]\!] = ([\![\mathcal{L}_1]\!], [\![\mathbf{M}_1]\!])$ *and* $[\![\mathbb{Q}_2]\!] = ([\![\mathcal{L}_2]\!], [\![\mathbf{M}_2]\!])$. *Here,* $[\![\mathcal{L}_1]\!] = \{[\![\mathbf{l}_1]\!], [\![\mathbf{l}_4]\!], [\![\mathbf{l}_2]\!]\}$ *and* $[\![\mathbf{M}_1]\!] = \begin{bmatrix} [\![0000]\!] & [\![0010]\!] & [\![0100]\!] \\ [\![0001]\!] & [\![0000]\!] & [\![0001]\!] \\ [\![0100]\!] & [\![0010]\!] & [\![0000]\!] \end{bmatrix}$; $[\![\mathcal{L}_2]\!] = \{[\![\mathbf{l}_3]\!]\}$ *and* $[\![\mathbf{M}_2]\!] = \emptyset$.

4.3.1 Secure Candidate Partial Matches Fetching

Design rationale. At a high level, the phase consists of two steps: (1) S_{123} obliviously fetch from $[\![\mathbb{G}_t]\!]$ encrypted edges that have label $[\![\mathbf{l}_y]\!] \in [\![\mathcal{L}_d]\!]$. These edges are named as *matched edges* and the set of matched edges for $[\![\mathbf{l}_y]\!]$ is denoted as $[\![\mathcal{E}_y]\!]$.

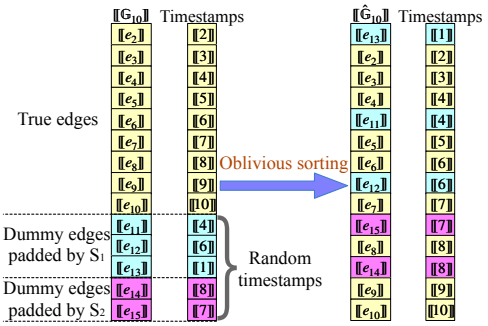


Figure 5: An example of oblivious dummy edges padding.

(2) S_{123} construct candidate partial matches by the edges from different sets $[\![\mathcal{E}_y]\!], y \in \mathbf{y}_d$ that obey $[\![\mathbb{Q}_d]\!]$'s timing order constraints. For step (1), we let S_{123} first perform a secure equality test over $[\![\mathbf{l}_y]\!] \in [\![\mathcal{L}_d]\!]$ and the encrypted label $[\![\mathbf{l}_x]\!]$ of each edge in the snapshot. To identify the matched edges, simply revealing the equality test results would leak which edges in the snapshot are matched edges (thereby leaking the access pattern [15]). Also, this would expose the frequency of distinct edge labels in the snapshot, which could be exploited by potential attacks [25]. Our insight is to let S_{123} *obliviously* pad encrypted dummy edges into the snapshot. This allows secure revelation of the test results while preventing S_{123} from learning the access pattern and the frequency of edge labels.

For step (2), our solution is based on the fact that $\mathbb{Q}_d$ is a TC-subquery pattern. If S_{123} can preserve the arrangement of true edges in the snapshot even after the oblivious padding of dummy edges, they can determine the (non-private) timing orders of all matched edges. As a result, S_{123} can trivially construct the encrypted candidate partial matches using the encrypted matched edges based on their clear timing orders and the clear timing order constraints of $[\![\mathbb{Q}_d]\!]$.

Oblivious dummy edges padding (Subroutine 1: olivPad). We first let S_1 and S_2 independently pad a specific number of encrypted dummy edges with different labels into $[\![\mathbb{G}_t]\!]$, while ensuring that S_{123} are oblivious to the *count* and *placement* of dummy edges with each label. Specifically, given each possible edge label l, S_1 first locally generates dummy edges in the form of $\{(-1,-1,l,t_x)\}$ (lines 3-7), where $\{t_x\}$ are random timestamps. The IDs of each dummy edge's two endpoints are set to -1, which distinguishes them from true vertices and prevents them from affecting the detection accuracy. S_1 encodes $-1,-1,l$ into one-hot vectors, and secret-shares all dummy edges with S_2 and S_3 to produce $\{([\![-\mathbf{1}]\!], [\![-\mathbf{1}]\!], [\![\mathbf{l}]\!], [\![t_x]\!])\}$ (line 9). The timestamps are also encrypted. Similarly, S_2 performs the above process (lines 10 and 11). S_{123} then integrate these encrypted dummy edges into the encrypted snapshot (line 12). Note that neither S_1 nor S_2 can learn the number of dummy edges with each label padded by the other party because the edge labels are encrypted.

However, S_{123} can still identify which edges in the padded snapshot are dummy, as they are aware of the edges recently

Subroutine 2 Secure Equality Test

Input: Two encrypted one-hot vectors $[\![\mathbf{a}]\!]$ and $[\![\mathbf{b}]\!]$.
Output: The encrypted test result $[\![\omega]\!]$ between $[\![\mathbf{a}]\!]$ and $[\![\mathbf{b}]\!]$.

// $\mathsf{S}_{\alpha\in\{1,2,3\}}$ locally performs the following:
1: **for all** $i \in [\rho]$ **do**
2: $\quad \langle\omega_i\rangle_\alpha \leftarrow \langle\mathbf{a}[i]\rangle_\alpha \otimes \langle\mathbf{b}[i]\rangle_\alpha \oplus \langle\mathbf{a}[i]\rangle_\alpha \otimes \langle\mathbf{b}[i]\rangle_{\alpha+1} \oplus \langle\mathbf{a}[i]\rangle_{\alpha+1} \otimes \langle\mathbf{b}[i]\rangle_\alpha$.
3: **end for**
4: $\langle\omega\rangle_\alpha \leftarrow \bigoplus_{i=1}^{\rho} \langle\omega_i\rangle_\alpha$.
5: **return** 3-out-of-3 additive secret sharing of ω, where $\mathsf{S}_{\alpha\in\{1,2,3\}}$ holds $\langle\omega\rangle_\alpha$ and $\omega = \langle\omega\rangle_1 \oplus \langle\omega\rangle_2 \oplus \langle\omega\rangle_3$.

Subroutine 3 Candidate Partial Matches Construction

Input: Encrypted matched edges $[\![\mathcal{E}_y]\!], y \in \mathbf{y}_d$.
Output: Encrypted candidate partial match set $[\![\mathcal{CP}_d]\!]$.

// S_{123} locally perform the following:
1: $[\![\mathcal{CP}_d]\!] \leftarrow \emptyset$. // Candidate partial match set.
2: $[\![\mathcal{C}]\!] \leftarrow \{[\![P_l]\!] = \{[\![e_i]\!]\}_{i\in[\kappa_d]} | \{[\![e_i]\!] \in [\![\mathcal{E}_{\mathbf{y}_d[i]}]\!], i \in [\kappa_d]\}\}$.
3: **for all** $[\![P_l]\!] \in [\![\mathcal{C}]\!]$ **do**
4: $\quad$ **if** $[\![P_l]\!] : [\![e_1]\!] \prec \cdots \prec [\![e_{\kappa_d}]\!]$ **then**
5: $\quad\quad [\![\mathcal{CP}_d]\!]$.add($[\![P_l]\!]$). // $[\![P_l]\!]$ is a candidate.
6: $\quad$ **end if**
7: **end for**
8: **return** Encrypted candidate partial match set $[\![\mathcal{CP}_d]\!]$.

shared by other cloud servers. Hence, the challenge we must address is to enable S_{123} *to rearrange the edges in the padded snapshot while ensuring that* S_{123} *remain unaware of the new locations of each dummy and true edge.* Additionally, it is essential to preserve the temporal order of the true edges. To accomplish both objectives, GraphGuard has S_{123} conduct oblivious sorting on the edges of the padded snapshot based on their encrypted timestamps (line 14). As the timestamps of dummy edges are randomized and encrypted, S_{123} cannot determine the new locations of dummy and true edges in the padded snapshot after oblivious sorting (denoted as $[\![\hat{\mathbb{G}}_t]\!]$), and thus the access pattern is hidden.

Here what we need is an oblivious sorting protocol that allows parties holding the secret-shared key-value pairs $\{([\![val]\!], [\![key]\!])\}$ to jointly sort $\{[\![val]\!]\}$ based on $\{[\![key]\!]\}$, while no party can learn the ranking of each encrypted value. We identify that the state-of-the-art protocol (denoted as $\mathsf{olivSort}(\cdot,\cdot)$) from [5] is well-suited for our purpose, as it enables fast oblivious stable sorting in the RSS domain.

Example 6. *Consider* $\mathbb{G}_{10}$ *shown in Fig. 2, with the corresponding oblivious dummy edge padding depicted in Fig. 5. Following oblivious sorting on* $\mathbb{G}_{10}$*, the dummy edges are repositioned based on their random timestamps. It is evident that the temporal order of the true edges is preserved.*

The remaining challenge is how to appropriately set the number of dummy edges for each edge label to delicately balance the trade-off between *efficiency* and *privacy*. More dummy edges will lead higher performance overhead, while fewer dummy edges will result in weaker privacy guarantees. Our key idea is to rely on edge DP [22] so as to make the leakage about the frequency of edge labels in the snapshot *differentially private*. Below, we present how S_1 determines the number of dummy edges, and S_2 follows the same approach.

Given each possible edge label l, S_1 first draws a noise n_l from the discrete Laplace distribution $Lap(\varepsilon, \delta, \Delta)$, which is set as the number of dummy edges for l (line 3). Here, the sensitivity Δ is set to 1 as the addition or removal of a single edge is limited to changing the frequency of the edge labels by at most 1. However, the drawn noise cloud be negative, which means that S_1 needs to delete some true edges from the snapshot. Obliviously, this will seriously degrade the accuracy of the subsequent computation. To address this issue, GraphGuard lets S_1 truncate the drawn noise to 0, inspired by [20], i.e., $n_l \leftarrow \max(Lap(\varepsilon, \delta, 1), 0)$. However, one challenge that still needs to be addressed is how to set the mean μ of $Lap(\varepsilon, \delta, 1)$ such that the truncated noise can still offer DP. We propose to set μ as

$$\mu = -\frac{\ln[(e^{\varepsilon}+1)\cdot(1-\delta)]}{\varepsilon}. \tag{2}$$

In Section 5, we prove that with the setting of μ, the drawn noises can still offer DP, even if they may be truncated to 0.

Secure equality test (Subroutine 2: secTest). We now present how S_{123} efficiently and securely perform secure equality tests on two encrypted one-hot vectors $[\![\mathbf{a}]\!]$ and $[\![\mathbf{b}]\!]$, outputting secret-shared $\omega = 1$ if $\mathbf{a} = \mathbf{b}$, and $\omega = 0$ otherwise. We use a bitwise AND operation to be carried out initially on $[\![\mathbf{a}]\!]$ and $[\![\mathbf{b}]\!]$, after which the XOR operation is utilized on the result of the AND operation to produce $[\![\omega]\!]$, i.e.,

$$[\![\omega]\!] = \bigoplus_{i=1}^{\rho} [\![\mathbf{a}[i]]\!] \otimes [\![\mathbf{b}[i]]\!]. \tag{3}$$

The solution is correct because there is only one "1" in both $\mathbf{a}$ and $\mathbf{b}$, and $\omega = 1$ only if the positions of "1" in $\mathbf{a}$ and $\mathbf{b}$ are identical, i.e., $\mathbf{a} = \mathbf{b}$. As Eq. 3 involves a "dot product" calculation, the re-sharing operation at the end of each $\otimes$ operation is no longer necessary. Specifically, S_{123} can locally perform $\oplus$ (line 4) over the 3-out-of-3 additive secret sharing of $\mathbf{a}[i] \otimes \mathbf{b}[i], i \in [\rho]$ (line 2) to produce the 3-out-of-3 additive secret sharing of ω. As a result, Eq. 3 enables an efficient secure equality test *without* the need for online communication among the parties and offline preparation.

S_{123} then reveal the test result to determine whether an encrypted edge is a matched edge. As S_{123} remain oblivious to the new locations of true edges and the frequency of true edge labels in the snapshot, the revelation of test results does not reveal the access pattern or the frequency of edge labels.

Candidate partial matches construction (Subroutine 3: Constr). After obliviously fetching matched edges for each

Subroutine 4 Secure Candidate Partial Matches Filtering

Input: $[\![\mathbb{Q}_d]\!]$'s candidate partial matches $[\![\mathcal{CP}_d]\!]$ and $[\![\mathbf{M}_d]\!]$.
Output: Encrypted partial match set $[\![\mathcal{P}_d]\!]$.

// S_{123} perform the following:
1: $[\![\mathcal{P}_d]\!] \leftarrow \emptyset$. //Initialize the partial match set.
2: **for all** $[\![P_l]\!] \in [\![\mathcal{CP}_d]\!]$ **do**
3: Construct $[\![P_l]\!]$'s encrypted EAM $[\![\mathbf{M}_l]\!]$ by Eq. 4.
4: Compute $[\![\omega]\!]$ based on $[\![\mathbf{M}_d]\!]$ and $[\![\mathbf{M}_l]\!]$ by Eq. 5.
5: Safely open $[\![\omega]\!]$; if $\omega = 0$, $[\![\mathcal{P}_d]\!]$.add($[\![P_l]\!]$).
6: **end for**
7: **return** Encrypted partial match set $[\![\mathcal{P}_d]\!]$.

edge label of $[\![\mathbb{Q}_d]\!]$, S_{123} perform step (2): constructing candidate partial matches by the edges from different matched edge sets that obey the timing order constraints of $[\![\mathbb{Q}_d]\!]$. As the edges in $[\![\hat{\mathbb{G}}_t]\!]$ are arranged based on their timestamps, S_{123} can determine the timing orders of all matched edges. Since $\mathbb{Q}_d$ is a TC-subquery pattern, S_{123} can trivially construct candidate partial matches by the matched edges based on their clear timing orders.

More specifically, S_{123} first construct the set $[\![\mathcal{C}]\!]$ (line 2), which includes all combinations of edges from different $[\![\mathcal{E}_y]\!], y \in \mathbf{y}_d$. S_{123} then examine the timing orders of edges within each combination $[\![P_l]\!] \in [\![\mathcal{C}]\!]$ (lines 3-7). Specifically, given $[\![P_l]\!]$, if $[\![e_1]\!] \prec \cdots \prec [\![e_{\kappa_d}]\!]$ holds, $[\![P_l]\!]$ is a candidate.

Example 7. *Consider $\mathbb{G}_9$ in Fig. 2 and $\mathbb{Q}_1$ in Fig. 4 as an example (without considering encryption and dummy edges for simplicity). $\mathbb{Q}_1$ consists of three edges $\{\sigma_1, \sigma_4, \sigma_2\}$, where $\sigma_1 \prec \sigma_4 \prec \sigma_2$. In $\mathbb{G}_9$, σ_1's matched edges (with label "005") are $\mathcal{E}_1 = \{e_1, e_6\}$; σ_4's matched edge (with label "004") is $\mathcal{E}_4 = \{e_5\}$; σ_2's matched edge (with label "006") is $\mathcal{E}_2 = \{e_7\}$. Therefore, all combinations of edges from $\mathcal{E}_1, \mathcal{E}_4, \mathcal{E}_2$ are $\mathcal{C} = \{P_1 = \{e_1, e_5, e_7\}, P_2 = \{e_6, e_5, e_7\}\}$. Since $e_6 \prec e_5 \prec e_7$ does not hold, P_2 is not a candidate. The final candidate partial match is $P_1 = \{e_1, e_5, e_7\}$ as $e_1 \prec e_5 \prec e_7$ holds.*

4.3.2 Secure Candidate Partial Matches Filtering

Design rationale. The candidate partial matches are extracted by considering only the edge labels and timing order constraints. We now introduce the method (Subroutine 4: secFlt) for securely filtering out the candidates whose structures are inconsistent with that of $[\![\mathbb{Q}_d]\!]$, to obtain the encrypted partial matches. As per Proposition 1, the structural consistency checking on subgraphs can be simplified to comparing their EAMs. Given that S_{123} have obtained the encrypted EAM $[\![\mathbf{M}_d]\!]$ of $[\![\mathbb{Q}_d]\!]$ during the **Pre-processing** phase, we next introduce how S_{123} obliviously construct the encrypted EAM for each encrypted candidate partial match. After that, we will detail how S_{123} securely compare the encrypted EAMs.

Oblivious encrypted EAMs construction. To construct the encrypted EAM $[\![\mathbf{M}_l]\!]$ for an encrypted candidate partial match, S_{123} need to perform secure equality test secTest over

Algorithm 1 Secure Partial Matches Detection

Input: A snapshot $[\![\mathbb{G}_t]\!]$ and $[\![\mathbb{Q}_d]\!] = ([\![\mathcal{L}_d]\!], [\![\mathbf{M}_d]\!])$.
Output: The partial match set $[\![\mathcal{P}_d]\!]$ with respect to $[\![\mathbb{Q}_d]\!]$.

1: $[\![\hat{\mathbb{G}}_t]\!] \leftarrow \mathsf{olivPad}([\![\mathbb{G}_t]\!])$. //Subroutine 1.
2: **for all** $[\![\mathbf{l}_y]\!] \in [\![\mathcal{L}_d]\!]$ **do**
3: $[\![\mathcal{E}_y]\!] \leftarrow \emptyset$. // Initialize the matched edge set.
4: **for all** $[\![e_x]\!] \in [\![\hat{\mathbb{G}}_t]\!]$ **do**
5: $[\![\omega]\!] \leftarrow \mathsf{secTest}([\![\mathbf{l}_x]\!], [\![\mathbf{l}_y]\!])$. //$[\![\mathbf{l}_x]\!]$ in $[\![e_x]\!]$; Subrou. 2.
6: Safely open $[\![\omega]\!]$; if $\omega = 1$, $[\![\mathcal{E}_y]\!]$.add($[\![e_x]\!]$).
7: **end for**
8: **end for**
9: $[\![\mathcal{CP}_d]\!] \leftarrow \mathsf{Constr}(\{[\![\mathcal{E}_y]\!]\})$. //Subroutine 3.
10: $[\![\mathcal{P}_d]\!] \leftarrow \mathsf{secFlt}([\![\mathcal{CP}_d]\!], [\![\mathbf{M}_d]\!])$. //Subroutine 4.
11: **return** The partial match set $[\![\mathcal{P}_d]\!]$ with respect to $[\![\mathbb{Q}_d]\!]$.

the endpoint IDs of each pair of edges $[\![e_i]\!], [\![e_j]\!]$ in the candidate partial match. Specifically, given $[\![e_i]\!]$'s endpoint IDs $([\![\mathbf{sid}_i]\!], [\![\mathbf{eid}_i]\!])$ and $[\![e_j]\!]$'s endpoint IDs $([\![\mathbf{sid}_j]\!], [\![\mathbf{eid}_j]\!])$, S_{123} perform:

$$[\![b_1]\!] = \mathsf{secTest}([\![\mathbf{sid}_i]\!], [\![\mathbf{sid}_j]\!]); [\![b_2]\!] = \mathsf{secTest}([\![\mathbf{eid}_i]\!], [\![\mathbf{eid}_j]\!]);$$
$$[\![b_3]\!] = \mathsf{secTest}([\![\mathbf{eid}_i]\!], [\![\mathbf{sid}_j]\!]); [\![b_4]\!] = \mathsf{secTest}([\![\mathbf{sid}_i]\!], [\![\mathbf{eid}_j]\!]);$$
$$[\![\mathbf{M}_l[i,j]]\!] = [\![b_1]\!] || [\![b_2]\!] || [\![b_3]\!] || [\![b_4]\!], i, j \in [\kappa_d], i \neq j, \quad (4)$$

where $[\![\mathbf{M}_l[i,i]]\!] = [\![0]\!] || [\![0]\!] || [\![0]\!] || [\![0]\!], i \in [\kappa_d]$; "||" denotes concatenation. To enable the subsequent computation in RSS domain, a re-sharing operation is required for each test result.

Secure structural consistency checking. Note that the edges of $[\![\mathbb{Q}_d]\!]$ have a one-to-one matching relationship with the edges in the encrypted candidate partial match. Therefore, based on Proposition 1, to check their structural consistency, S_{123} can evaluate whether $\mathbf{M}_d = \mathbf{M}_l$ holds. Specifically, S_{123} compute Eq. 1 on $[\![\mathbf{M}_d]\!]$ and $[\![\mathbf{M}_l]\!]$ in the RSS domain:

$$[\![\omega]\!] = \bigvee_{i=1}^{\kappa_d} \bigvee_{j=1}^{\kappa_d} \bigvee_{s=1}^{4} [\![\mathbf{M}_d[i,j][s]]\!] \oplus [\![\mathbf{M}_l[i,j][s]]\!], \quad (5)$$

where $\omega = 1$ indicates $\mathbf{M}_d \neq \mathbf{M}_l$ and $\omega = 0$ indicates $\mathbf{M}_d = \mathbf{M}_l$. Finally, S_{123} open $[\![\omega]\!]$ to determine whether adding the candidate partial match into the partial match set (line 5).

Algorithm 1 provides a complete construction for secure partial matches detection.

4.4 Secure Partial Matches Compatibility Checking

After obtaining the encrypted partial match sets, S_{123} then combine partial matches from different sets into the final encrypted detection result $[\![\mathcal{R}]\!]$. However, simply combining partial matches can lead to incorrect matches because the timing orders of edges and the structures of partial matches from different sets may not be compatible with each other. Hence, in this section we introduce how S_{123} securely check

the compatibility among encrypted partial matches to produce the final detection result $[\![\mathcal{R}]\!]$ (given in Algorithm 2). The process involves examining all possible combinations of partial matches (named as *candidate matches*). Specifically, given an encrypted candidate match $[\![R]\!]$, S_{123} first check if the timing orders of its edges adhere to the timing order constraints of $[\![\mathbb{Q}]\!]$ (Section 4.4.1). S_{123} then check whether the structure of $[\![R]\!]$ is consistent with that of $[\![\mathbb{Q}]\!]$ (Section 4.4.2). If the combination passes these checks, S_{123} add $[\![R]\!]$ into $[\![\mathcal{R}]\!]$.

4.4.1 Secure Timing Orders Compatibility Checking

Recall from the end of Section 4.3.1 that the edges in each partial match already obey the timing order constraints of the corresponding TC-subquery pattern. Therefore, S_{123} only need to check whether the timing orders of edges from different partial matches in $[\![R]\!]$ obey the encrypted compatibility order constraints. Specifically, given each encrypted compatibility order constraint $[\![o_c]\!] = (y_1, y_2, [\![\theta]\!])$, S_{123} first retrieve the two edges (denoted as $[\![e_1]\!]$ and $[\![e_2]\!]$) from $[\![R]\!]$ that match the edges σ_{y_1} and σ_{y_2} of $\mathbb{Q}$, respectively. Since S_{123} have access to the mappings between the edges in $[\![R]\!]$ and the edges in $[\![\mathbb{Q}]\!]$, the retrieval can be done easily.

S_{123} then check whether the timing orders of $[\![e_1]\!]$ and $[\![e_2]\!]$ comply with $[\![o_c]\!]$. The checking process is formalized as $[\![\varphi_c]\!] = ([\![e_2]\!] \overset{?}{\prec} [\![e_1]\!]) \oplus [\![\theta]\!]$ (line 6 of Algorithm 2), where the expression $([\![e_2]\!] \overset{?}{\prec} [\![e_1]\!])$ evaluates to 1 if $[\![e_1]\!]$ comes after $[\![e_2]\!]$ in $[\![\hat{\mathbb{G}}_t]\!]$, and 0 otherwise. Since S_{123} have access to the rankings of $[\![e_1]\!]$ and $[\![e_2]\!]$ in $[\![\hat{\mathbb{G}}_t]\!]$, this expression $([\![e_2]\!] \overset{?}{\prec} [\![e_1]\!])$ can be evaluated in plaintext. Here, $\varphi_c = 1$ indicates that the timing order between $[\![e_1]\!]$ and $[\![e_2]\!]$ complies with $[\![o_c]\!]$, while $\varphi_c = 0$ indicates the opposite. Since the secret-shared "$\oplus$" operation does not require online communication and offline preparation, the checking process proceeds *without* the need for online communication and offline preparation. Finally, S_{123} combine $[\![\varphi_c]\!]$ for all compatibility order constraints to obtain the check result φ (line 8 of Algorithm 2), where $\varphi = 1$ means that the timing orders of edges in $[\![R]\!]$ comply with all timing order constraints of $[\![\mathbb{Q}]\!]$, and $\varphi = 0$ means the opposite.

Example 8. *Recalling Fig. 2, Fig. 4, and Example 7, the partial match for $\mathbb{Q}_1 = \{\sigma_1, \sigma_4, \sigma_2\}$ is $\{e_1, e_5, e_7\}$, the partial matches for $\mathbb{Q}_2 = \{\sigma_3\}$ are $\{e_4\}$ and $\{e_9\}$, and the compatibility order constraint between $\mathbb{Q}_1$ and $\mathbb{Q}_2$ is $(3, 2, \theta = 1)$, i.e., $\sigma_3 \prec \sigma_2$. Firstly, we examine the compatibility of timing orders of $\{e_1, e_5, e_7\} \cup \{e_4\}$, where e_4 matches σ_3 and e_7 matches σ_2. Since $(e_7 \overset{?}{\prec} e_4) = 0$, $\varphi = 0 \oplus 1 = 1$. This result indicates that $e_4 \prec e_7 \approx \sigma_3 \prec \sigma_2$, namely the timing orders of $\{e_1, e_5, e_7\} \cup \{e_4\}$ comply with the timing order constraints of $\mathbb{Q}$. This conclusion is further supported by Fig. 2. Next, we evaluate $\{e_1, e_5, e_7\} \cup \{e_9\}$. In this case, e_9 matches σ_3 and e_7 matches σ_2. With $(e_7 \overset{?}{\prec} e_9) = 1$, we have $\varphi = 1 \oplus 1 = 0$. This result indicates that the timing orders of $\{e_1, e_5, e_7\} \cup \{e_9\}$ do not comply with the timing order constraints of $\mathbb{Q}$.*

Algorithm 2 Secure Partial Matches Compatibility Checking

Input: $\{[\![o_c]\!]\}_{c\in[C]}$ and $[\![\mathbf{M}]\!]$; partial matches $[\![\mathcal{P}_d]\!], d \in [D]$.
Output: The encrypted detection result set $[\![\mathcal{R}]\!]$.

1: $[\![\mathcal{R}]\!] \leftarrow \emptyset$. // `Initialize detection result set.`
2: $[\![\mathcal{C}]\!] \leftarrow \{[\![R]\!] = \bigcup_{d\in[D]} [\![P_d]\!] | \{[\![P_d]\!] \in [\![\mathcal{P}_d]\!], d \in [D]\}\}$.
3: **for all** $[\![R]\!] \in [\![\mathcal{C}]\!]$ **do**
4: **for all** $[\![o_c]\!] \in \{[\![o_c]\!]\}_{c\in[C]}$ **do**
5: Retrieve edges $[\![e_1]\!]$ and $[\![e_2]\!]$ from $[\![R]\!]$ that match the edges σ_{y_1} and σ_{y_2} of $[\![\mathbb{Q}]\!]$, respectively.
6: $[\![\varphi_c]\!] \leftarrow ([\![e_2]\!] \overset{?}{\prec} [\![e_1]\!]) \oplus [\![\theta]\!]$.
7: **end for**
8: $[\![\varphi]\!] \leftarrow \bigotimes_{c=1}^{C} [\![\varphi_c]\!]$. //`Check the timing orders.`
9: Construct $[\![R]\!]$'s encrypted EAM $[\![\mathbf{M}_R]\!]$ by Eq. 4.
10: Compute $[\![\omega]\!]$ based on $[\![\mathbf{M}]\!]$ and $[\![\mathbf{M}_R]\!]$ by Eq. 5.
11: $[\![\chi]\!] \leftarrow [\![\varphi]\!] \otimes [\![\neg\omega]\!]$. //`Aggregate the results.`
12: Safely open $[\![\chi]\!]$, if $\chi = 1$, $[\![\mathcal{R}]\!]$.add($[\![R]\!]$).
13: **end for**
14: **return** The encrypted detection result set $[\![\mathcal{R}]\!]$.

4.4.2 Secure Structural Compatibility Checking

S_{123} then need to check whether the structure of $[\![R]\!]$ is consistent with that of $[\![\mathbb{Q}]\!]$. To check the structural consistency, similar to Section 4.3.2, S_{123} first construct $[\![R]\!]$'s encrypted EAM $[\![\mathbf{M}_R]\!]$ by Eq. 4 (line 9 of Algorithm 2), and then evaluate Eq. 5 over $[\![\mathbf{M}_R]\!]$ and $[\![\mathbb{Q}]\!]$'s encrypted EAM $[\![\mathbf{M}]\!]$ to output $[\![\omega]\!]$ (line 10 of Algorithm 2). Here, $\omega = 0$ indicates that the structures of $[\![R]\!]$ and $[\![\mathbb{Q}]\!]$ are consistent, while $\omega = 1$ indicates the opposite. Note that S_{123} need to construct $[\![\mathbf{M}_R]\!]$ based on the mappings between edges in $[\![R]\!]$ and edges in $[\![\mathbb{Q}]\!]$ to ensure that evaluating Eq. 5 on $[\![\mathbf{M}_R]\!]$ and $[\![\mathbf{M}]\!]$ outputs the correct result. Additionally, evaluating Eq. 5 on the EAM of matches containing dummy edges will always output $\omega = 1$, as dummy edges are not connected to any true edges. In other words, the dummy edges will not compromise the detection accuracy (as analyzed in Appendix E). We use the following example to illustrate the process.

Example 9. *Recalling the combination $\{e_1, e_5, e_7\} \cup \{e_4\}$ from Example 8, the edges e_1, e_5, e_7, and e_4 match edges σ_1, σ_4, σ_2, and σ_3 of $\mathbb{Q}$, respectively. Therefore, in the EAM of $\{e_1, e_5, e_7\} \cup \{e_4\}$, e_1 corresponds to the first row/column, e_5 corresponds to the fourth row/column, e_7 corresponds to the second row/column, and e_4 corresponds to the third row/column. Based on the structure of $\mathbb{G}_9$ in Fig. 2, the EAM of $\{e_1, e_5, e_7\} \cup \{e_4\}$ is* $\begin{bmatrix} 0000 & 0100 & 0000 & 0010 \\ 0100 & 0000 & 1000 & 0010 \\ 0000 & 1000 & 0000 & 0100 \\ 0001 & 0001 & 0100 & 0000 \end{bmatrix}$, *which matches the EAM of $\mathbb{Q}$ (as shown in Example 3).*

S_{123} AND $[\![\varphi]\!]$ and $[\![\neg\omega]\!]$ to obtain the check result (line 11 of Algorithm 2). Then S_{123} open $[\![\chi]\!]$, where $\chi = 1$ indicates that $\varphi = 1$ and $\omega = 0$, i.e., $[\![R]\!]$ is a time-constrained match of $[\![\mathbb{Q}]\!]$, while $\chi = 0$ indicates the opposite. Here, "$\neg$" represents the "NOT" operation in the binary domain, which can be realized by having two of S_{123} locally flip the share they jointly hold. Finally, if $[\![\mathcal{R}]\!]$ is not empty, S_{123} return it to FE, which decrypts $[\![\mathcal{R}]\!]$ and alerts the regulator.

5 Privacy Analysis

Proposition 2. *GraphGuard provides $(2\varepsilon, 2\delta)$-edge DP for the frequency of edge labels in each snapshot based on Definition 5.*

We prove Proposition 2 in Appendix F.

Remark. Proposition 2 indicates that the strength of DP protection for the frequency of edge labels in each snapshot is tunable and dependent on the privacy parameters ε and δ. Note that to our best knowledge, there is no deterministic theoretical formulas for determining the number of repeated queries needed to exploit the DP leakages for data reconstruction.

6 Security Analysis

We use the simulation paradigm [28] to analyze the security of GraphGuard. Establishing security under the simulation paradigm requiring formalizing two worlds: the real world, where the protocol is executed by honest parties, and an ideal world, where an ideal functionality $\mathcal{F}$ receives inputs from the parties and directly outputs the result to the relevant parties. We consider a probabilistic polynomial time (PPT) adversary $\mathcal{A}$ who *statically* corrupts one of S_{123}. We need to define a PPT simulator $\mathcal{S}$ who only accesses the leakage of $\mathcal{F}$ as claimed by our protocols, and "simulates" messages that resemble what the honest parties send to the corrupted server in the real world. If $\mathcal{A}$ cannot distinguish between the two worlds, then we consider our protocols to be secure.

Ideal functionality $\mathcal{F}$. Recall that the system of GraphGuard consists of the users, the FE dedicatedly maintained by the regulator, and the cloud servers. As FE does not have input and output, and solely serves as a proxy facilitating interaction between the users and the regulator with the cloud servers, we do not explicitly reference it in $\mathcal{F}$.

1. Setup: The regulator sends the public parameters Para, and the query pattern $\mathbb{Q}$ to $\mathcal{F}$, where Para consists of the spaces of vertex ID and edge label, and the window size. $\mathcal{F}$ executes the setup process with Para and $\mathbb{Q}$.

2. Append: A user sends a newly generated edge $e_x = (sid_x, eid_x, l_x, t_x)$ to $\mathcal{F}$, where sid_x and eid_x are the IDs of e_x's two endpoints, l_x is e_x's label, and t_x is e_x's timestamp. $\mathcal{F}$ appends the edge e_x to the current graph.

3. Detection: $\mathcal{F}$ performs time-constrained pattern detection for $\mathbb{Q}$ on each snapshot of the streaming graph $\{e_x\}$.

4. Output: If any time-constrained match(es) $\mathcal{R}$ is detected, $\mathcal{F}$ outputs $\mathcal{R}$ to the regulator. Otherwise, $\mathcal{F}$ outputs nothing (i.e., $\perp$) to the regulator.

Note that $\mathcal{F}$ outputs nothing to the servers and the users. However, we allow $\mathcal{F}$ to leak $\mathcal{L}(\mathcal{F}(\mathbb{Q})) = (\{t_x\}, \mathsf{Num}, \{\hat{f}\})$ to the servers. Here, $\mathsf{Num} = (|\mathbb{Q}|, |\mathbb{G}|, |\mathcal{CP}|, |\mathcal{P}|, |\mathcal{R}|)$, where $|\mathbb{Q}|$ is the number of edges in $\mathbb{Q}$, $|\mathbb{G}|$ is the number of edges in the streaming graph, $|\mathcal{CP}|$ is the number of candidate partial matches, $|\mathcal{P}|$ is the number of partial matches, and $|\mathcal{R}|$ is the number of detection results. $\{\hat{f}\}$ are the noisy frequencies of edge labels in each snapshot.

***Definition* 9.** *Let $\prod$ denote the protocol for privacy-preserving time-constrained graph pattern detection, wherein the regulator provides the query pattern $\mathbb{Q}$ as input. Let $\mathcal{A}$ be an adversary who statically corrupts one of the servers S_{123}, and let $\mathsf{View}_{\mathsf{Real}}^{\prod(\mathbb{Q})}$ be the view of the corrupted server during the protocol run. In the ideal world, a simulator $\mathcal{S}$ generates a simulated view $\mathsf{View}_{\mathsf{Ideal}}^{\mathcal{S},\mathcal{L}(\mathcal{F}(\mathbb{Q}))}$ given only the leakage of $\mathcal{F}$. We say that $\prod$ is secure, if there exists a PPT simulator $\mathcal{S}$ such that $\mathsf{View}_{\mathsf{Ideal}}^{\mathcal{S},\mathcal{L}(\mathcal{F}(\mathbb{Q}))}$ is indistinguishable from $\mathsf{View}_{\mathsf{Real}}^{\prod(\mathbb{Q})}$.*

Proposition 3. *Based on Definition 9, GraphGuard securely realizes $\mathcal{F}$ with the leakage $\mathcal{L}(\mathcal{F}(\mathbb{Q}))$.*

We prove Proposition 3 in Appendix G. In addition, we explicitly analyze how GraphGuard hides search access patterns in Appendix H. Here, the search pattern implies whether a new query pattern has been issued before, while the access pattern reveals which vertices/edges of the streaming graph are matched with that of the query pattern.

Remark. While the focus of GraphGuard is on privacy protection for the streaming graph formed by edges contributed by users, another concern upon practical deployment might be collusion between any of the semi-honest servers and some users. In such case, the privacy of the edges generated by honest users still holds. This is because in GraphGuard users only generate edges for the streaming graph and do not receive any messages during the protocol execution. Meanwhile, the nature of secret sharing ensures that any of the servers alone learns no information about the data being secret-shared.

7 Performance Evaluation

7.1 Setup

We make a prototype implementation of GraphGuard using a combination of Python and C++. All experiments are conducted on a workstation equipped with 24 Intel Xeon Gold 6240R CPU cores, a NVIDIA RTX A6000 GPU, 128 GB of RAM, and 2 TB of external SSD storage, running Ubuntu

20.04.3 LTS. The server-side communication on the workstation is emulated by the loopback filesystem with 10 ms network delay. Furthermore, we utilize a MacBook Air with 8 GB of RAM to act as the FE to encrypt and upload the query patterns and graph edges to the workstation. δ is set to 10^{-5}.

GPU-based protocol instantiation. We note that the design of GraphGuard lends itself well to parallelization, allowing us to leverage the power of GPUs for efficient parallel processing and performance enhancement. In GraphGuard, all private information from the streaming graph and query pattern is encoded as one-hot vectors and encrypted by binary RSS. The primary operations in GraphGuard involve addition and multiplication of (one-hot) vectors. To facilitate efficient processing on the GPU, we store the ciphertext of the streaming graph and query patterns as bit-strings using the Numpy library. We implement GraphGuard by utilizing optimized NVIDIA's Python-based CUDA kernels. We empirically observe that the GPU-based instantiation of secTest can perform *one million* secure equality tests on secret-shared one-hot vectors with a length of 10000, in just 1 s. We compare secTest with two commonly used secure equality test protocols: the bit decomposition-based protocol [24] and the distributed point function (DPF)-based protocol [9]. Using the same GPU hardware, performing one million secure equality tests on secret-shared values in the ring $\mathbb{Z}_{2^{16}}$ via the bit decomposition-based protocol [24] takes about 14 s[3], while the DPF-based protocol [9] takes about 3 s[4]. Besides, our secTest securely produces the secret-shared equality test result without requiring any online communication or offline preparation. In contrast, the bit decomposition-based protocol [24] needs 154.5 MB of online communication, while the DPF-based protocol [9] requires the cloud servers to receive offline one million pairs of DPF keys (with a size of about 520 MB) and needs 3.8 MB of online communication.

Datasets. We conduct experiments mainly using three real-world graph datasets: (1) MOOC user action: https://snap.stanford.edu/data/act-mooc.html. This dataset is a small and dense network, consisting of 7,143 vertices and 411,749 temporal edges. (2) Reddit hyperlink network: https://snap.stanford.edu/data/soc-RedditHyperlinks.html. This dataset is a medium size and sparse network containing 55,863 vertices and 858,490 temporal edges. (3) com-DBLP: https://snap.stanford.edu/data/com-DBLP.html. This dataset is a large and sparse network containing 317,080 vertices and 1,049,866 edges.

Query pattern sets. We generate each query pattern randomly, considering the statistics of the corresponding dataset. The process involves randomly generating a query pattern of size $|\mathbb{Q}|$, while considering the average degree and edge label distribution of the dataset. In addition, we randomly assign timing order constraint for each pair of edges in the query pattern with a probability of $1/|\mathbb{Q}|$. For each experiment, we use 10 random query patterns.

[3]We use the public source code from [24].

[4]We implement the protocol of [9], with the security parameter $\lambda = 128$.

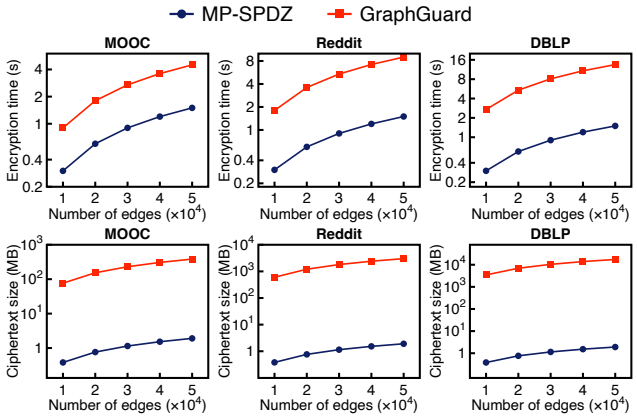


Figure 6: Running time and output ciphertext size due to encrypting streaming graphs, varying the number of edges.

Baseline. To our best knowledge there is no prior work solving the same problem targeted by us. Hence, we have to construct a baseline simply using generic MPC framework for direct secure computation of time-constrained graph pattern detection over streaming graphs. Under this we are able to fairly demonstrate the effectiveness of our custom designs. We construct our baseline using the generic and popular MPC framework MP-SPDZ [24], as it has been widely used for constructing baselines in recent works [16,36,41] and is well documented. In the baseline, identical to GraphGuard, we adopt the edge list structure [32] to represent the streaming graph. In addition, we use the same system model as GraphGuard, i.e., three servers with an honest majority and operating under a semi-honest threat model. We use the following basic secure protocols of MP-SPDZ: get_random$(\cdot)$, add$(\cdot)$, mul$(\cdot)$, reveal_to$(\cdot)$, secure_shuffle$(\cdot)$, LessThanZero$(\cdot)$, equal$(\cdot)$ to construct the baseline. It is noted that since the baseline directly secures the computation required by plaintext graph pattern detection, it needs to work with a large ring $\mathbb{Z}_{2^{32}}$, as opposed to $\mathbb{Z}_2$ used in GraphGuard due to our custom design of data encoding method and secure components. In addition, to provide the baseline with an advantage, we use the computational capabilities of GPUs for parallel processing.

7.2 Evaluation on Encryption

Fig. 6 compares the cost of encrypting streaming graphs between GraphGuard and the baseline. Since GraphGuard utilizes custom data encoding and encryption mechanisms (e.g., one-hot encoding), it results in higher computational cost for encryption and larger ciphertext size compared to the baseline. However, as will be shown shortly, our custom design allows GraphGuard to support significantly faster online graph pattern detection. Additionally, we point out that in GraphGuard the servers only need to use the encrypted edges within the sliding window for secure pattern detection, following the processing logic in plaintext domain (e.g., [27, 35]). Hence,

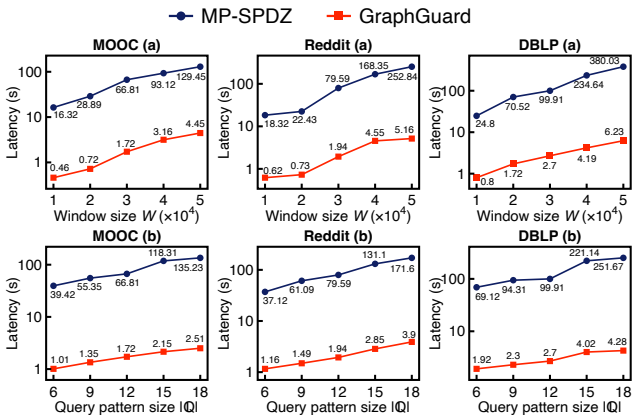


Figure 7: Query latency of GraphGuard and the baseline on different datasets with (a) $|\mathbb{Q}| = 12$ and $W \in [1 \times 10^4, 5 \times 10^4]$; (b) $W = 3 \times 10^4$ and $|\mathbb{Q}| \in \{6, 9, 12, 15, 18\}$.

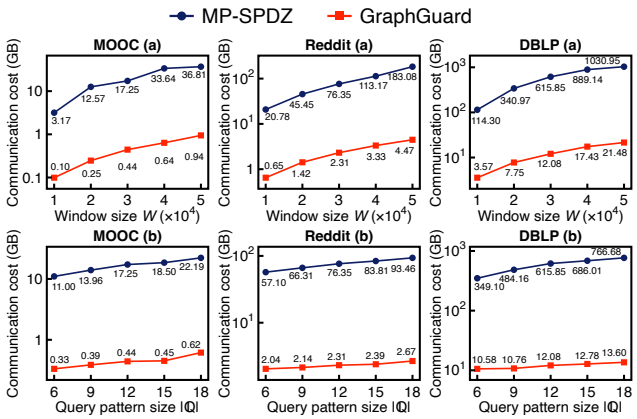


Figure 8: Communication cost of GraphGuard and the baseline on different datasets with (a) $|\mathbb{Q}| = 12$ and $W \in [1 \times 10^4, 5 \times 10^4]$; (b) $W = 3 \times 10^4$ and $|\mathbb{Q}| \in \{6, 9, 12, 15, 18\}$.

		secFet	secFlt	secCHK
MOOC	MP-SPDZ	66.8	54.33	8.32
	GraphGuard	**2.19**	**1.96**	**0.3**
Reddit	MP-SPDZ	131.75	114.58	6.51
	GraphGuard	**2.61**	**2.09**	**0.46**
DBLP	MP-SPDZ	221.91	130.36	27.76
	GraphGuard	**3.26**	**2.23**	**0.74**

Table 1: Breakdown of query latency for the comparison between GraphGuard and the baseline on different datasets (with $|\mathbb{Q}| = 12$ and $W = 5 \times 10^4$).

		secFet	secFlt	secCHK
MOOC	MP-SPDZ	18.4	15.33	3.08
	GraphGuard	**0.45**	**0.37**	**0.12**
Reddit	MP-SPDZ	81.48	73.45	28.15
	GraphGuard	**1.94**	**1.76**	**0.76**
DBLP	MP-SPDZ	502.31	405.19	123.45
	GraphGuard	**10.3**	**7.77**	**3.41**

Table 2: Breakdown of communication cost for the comparison between GraphGuard and the baseline on different datasets (with $|\mathbb{Q}| = 12$ and $W = 5 \times 10^4$).

in such case the memory cost is relatively small compared to maintaining the complete graph, e.g., not exceeding 1 GB on the MOOC dataset with a window size of 5×10^4.

7.3 Evaluation on Query Latency

Fig. 7 shows the comparison of query latency between GraphGuard and the baseline under different window sizes W and query pattern sizes $|\mathbb{Q}|$. Note that we do not vary the privacy budget ε in this comparison ($\varepsilon = 0.6$ for the results of GraphGuard) since the baseline is fully built on RSS technique. The results clearly demonstrate that GraphGuard consistently outperforms the baseline, achieving a substantial speedup ranging from 29× to 60×. Additionally, the query latency gap between GraphGuard and the baseline increases significantly as W and $|\mathbb{Q}|$ increase. The results show the superior scalability of GraphGuard compared to the baseline.

To better demonstrate the advantages of GraphGuard over the baseline, we provide a breakdown of the query latency performance in Table 1. Here, we divide the overall secure processing pipeline into three common subroutines: secure candidate partial matches fetching (denoted as secFet), secure candidate partial matches filtering (denoted as secFlt), and secure partial matches compatibility checking (denoted as secCHK). Note that secFet and secFlt together support secure partial matches detection. We can observe that GraphGuard consistently outperforms the baseline in all subroutines. The advantage of GraphGuard in secFet stems from our oblivious dummy edges padding protocol, which enables the secure revelation of equality test results on encrypted edge labels, thereby significantly reducing the subsequent unnecessary overhead on unmatched edges. In secFlt and secCHK, it is our proposed data structure EAM that enables S_{123} to efficiently and securely check the structural consistency between encrypted candidate matches and the query pattern.

7.4 Evaluation on the Server-Side Communication Cost

Fig. 8 shows the comparison of the server-side communication cost between GraphGuard and the baseline under different window sizes W and query pattern sizes $|\mathbb{Q}|$. The results demonstrate that GraphGuard consistently outperforms the baseline, achieving substantial communication cost savings ranging from 96% to 98%. Furthermore, the communication cost savings of GraphGuard compared to the baseline increase significantly as the values of W and $|\mathbb{Q}|$ increase.

In Table 2, we provide a breakdown of the communication cost for more detailed comparison between GraphGuard and

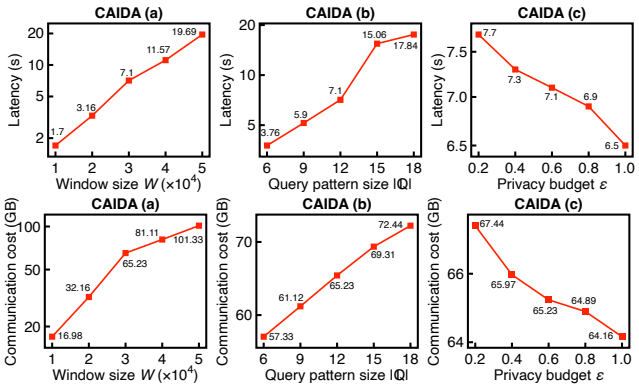


Figure 9: Query latency and communication cost of GraphGuard on the CAIDA dataset with (a) $\varepsilon = 0.6$, $|\mathbb{Q}| = 12$, and $W \in [1 \times 10^4, 5 \times 10^4]$; (b) $\varepsilon = 0.6$, $W = 3 \times 10^4$, and $|\mathbb{Q}| \in \{6, 9, 12, 15, 18\}$; and (c) $W = 3 \times 10^4$, $|\mathbb{Q}| = 12$, and $\varepsilon \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$.

the baseline. We can observe that GraphGuard outperforms the baseline in all subroutines. The advantage of GraphGuard in secFet and secFlt primarily comes from our efficient secure equality test protocol, which eliminates the need for online communication among the parties. In secCHK, the advantage of GraphGuard largely comes from our specialized secure edge temporal order comparison method, which also eliminates the need for online communication.

7.5 Scalability of GraphGuard

To demonstrate the scalability of GraphGuard, we further conduct experiments on a much larger dataset "CAIDA Anonymized Internet Traces 2015" obtained from https://catalog.caida.org/dataset/passive_2015_pcap. It contains 445,440,480 communication records (edges) involving 2,601,005 different IP addresses (vertices), with each edge associated with a timestamp indicating the communication time. This dataset is also the largest real-world streaming graph dataset used in the state-of-the-art plaintext work [27]. We model the dataset as an edge-labeled streaming graph, as described in [27]. Fig. 9 shows the results. Note that with the same snapshot size, a larger number of vertices in the dataset will result in higher query latency due to the impact on the length of the one-hot encoded vertex ID. The results demonstrate GraphGuard's scalability: on an encrypted snapshot of size of 5×10^4, the query latency only increases from 6.23 seconds on the DBLP dataset (comprising 317,080 vertices) to 19.69 seconds on the CAIDA dataset.

8 Related Work

Recently, significant efforts have been made towards developing techniques for searching encrypted outsourced graphs. Some works [17, 19] focus on privacy-preserving shortest path search, which aims to identify a path between two given vertices in an encrypted graph that minimizes the sum of the edge weights in the path. Some works [4, 7] focus on privacy-preserving breadth-first search, which aims to identify a graph structure from an encrypted graph that satisfies a given property. Other works [39, 42] focus on privacy-preserving subgraph matching, with the objective of identifying subgraphs isomorphic to a given small query graph from a large encrypted graph. However, it is important to note that the works [39, 42] focus on subgraph matching over *static* graphs, instead of time-constrained pattern detection over streaming graphs targeted by this paper, which is much more complicated as it additionally handles streaming graphs and considers the timing orders of the edges in graphs. In independent work, a design SGPM [21] is claimed to support secure time-constrained graph pattern matching. But it indeed addresses a *different and simplified* problem. In its scenario, the cloud holds each encrypted edge of a streaming graph independently *without considering/protecting the graph structure*, the client provides an encrypted *time constraint* as the query (instead of a query graph pattern), and the cloud obtains and returns the *plaintext* edge IDs as the query result.

In short, no previous studies have explored the same problem of privacy-preserving outsourcing of time-constrained pattern detection over streaming graphs as we do in this paper.

9 Conclusion

We present GraphGuard, the first system that enables privacy-preserving outsourcing of time-constrained pattern detection over streaming graphs. Through a synergistic approach bridging graph modeling, lightweight secret sharing, edge DP, and data encoding and padding, GraphGuard enables cloud-empowered time-constrained pattern detection over streaming graph while preserving the privacy of the streaming graph, the query patterns, and the detection results. The evaluation results on several real-world datasets demonstrate that compared with the generic MPC baseline, GraphGuard achieves up to 60× improvement in query latency and up to 98% savings in communication cost. We believe that our initial research effort lays a good foundation for advancing research on privacy-preserving query processing on dynamic graphs.

Acknowledgments

We sincerely thank the shepherd and the anonymous reviewers for their constructive and invaluable feedback. This work was supported in part by Guangdong Basic and Applied Basic Research Foundation under Grant 2023A1515010714, and in part by Shenzhen Science and Technology Program under Grant JCYJ20220531095416037 and Grant JCYJ20230807094411024.

References

[1] Airbnb on AWS. Aws case study: Airbnb. https://aws.amazon.com/solutions/case-studies/airbnb/?nc1=h_ls, 2018.

[2] Apple and Google. Exposure notification privacy-preserving analytics white paper. https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf, 2021.

[3] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *Proc. of ACM CCS*, 2016.

[4] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure graph analysis at scale. In *Proc. of ACM CCS*, 2021.

[5] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. Efficient secure three-party sorting with applications to data analysis and heavy hitters. In *Proc. of ACM CCS*, 2022.

[6] James Bell, Adria Gascon, Badih Ghazi, Ravi Kumar, Pasin Manurangsi, Mariana Raykova, and Phillipp Schoppmann. Distributed, private, sparse histograms in the two-server model. In *Proc. of ACM CCS*, 2022.

[7] Marina Blanton, Aaron Steele, and Mehrdad Aliasgari. Data-oblivious graph algorithms for secure computation and outsourcing. In *Proc. of ACM AsiaCCS*, 2013.

[8] Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In *Proc. of EUROCRYPT*, 2021.

[9] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proc. of ACM CCS*, 2016.

[10] Lennart Braun, Mahak Pancholi, Rahul Rachuri, and Mark Simkin. Ramen: Souper fast three-party computation for RAM programs. In *Proc. of ACM CCS*, 2023.

[11] Sutanay Choudhury, Lawrence B. Holder, George Chin Jr., Khushbu Agarwal, and John Feo. A selectivity based approach to continuous pattern detection in streaming graphs. In *Proc. of EDBT*, 2015.

[12] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence*, 26(10):1367–1372, 2004.

[13] Max Curran, Xiao Liang, Himanshu Gupta, Omkant Pandey, and Samir R Das. ProCSA: Protecting privacy in crowdsourced spectrum allocation. In *Proc. of ESORICS*, 2019.

[14] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Søren B. Lassen, Philip Pronin, Sriram Sankar, Guanghao Shen, Gintaras Woss, Chao Yang, and Ning Zhang. Unicorn: A system for searching the social graph. *Proceedings of the VLDB Endowment*, 6(11):1150–1161, 2013.

[15] Reza Curtmola, Juan A. Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: improved definitions and efficient constructions. In *Proc. of ACM CCS*, 2006.

[16] Emma Dauterman, Mayank Rathee, Raluca Ada Popa, and Ion Stoica. Waldo: A private time-series database from function secret sharing. In *Proc. of IEEE S&P*, 2022.

[17] Minxin Du, Shuangke Wu, Qian Wang, Dian Chen, Peipei Jiang, and Aziz Mohaisen. Graphshield: Dynamic large graphs for secure queries with forward privacy. *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[18] Cynthia Dwork. Differential privacy. In *Proc. of ICALP*, 2006.

[19] Esha Ghosh, Seny Kamara, and Roberto Tamassia. Efficient graph encryption scheme for shortest path queries. In *Proc. of ACM AsiaCCS*, 2021.

[20] Xi He, Ashwin Machanavajjhala, Cheryl J. Flynn, and Divesh Srivastava. Composing differential privacy and secure computation: A case study on scaling private record linkage. In *Proc. of ACM CCS*, 2017.

[21] Jinjing Huang, Wei Chen, Zhixu Li, Pengpeng Zhao, Weiqing Wang, Hongzhi Yin, and Lei Zhao. SGPM: a privacy protected approach of time-constrained graph pattern matching in cloud. *World Wide Web*, 23(1):519–547, 2020.

[22] Jacob Imola, Takao Murakami, and Kamalika Chaudhuri. Differentially private triangle and 4-cycle counting in the shuffle model. In *Proc. of ACM CCS*, 2022.

[23] Peipei Jiang, Qian Wang, Muqi Huang, Cong Wang, Qi Li, Chao Shen, and Kui Ren. Building in-the-cloud network functions: Security and privacy challenges. *Proceedings of the IEEE*, 109(12):1888–1919, 2021.

[24] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proc. of ACM CCS*, 2020.

[25] Evgenios M. Kornaropoulos, Charalampos Papamanthou, and Roberto Tamassia. The state of the uniform: Attacks on encrypted databases beyond the uniform query distribution. In *Proc. of IEEE S&P*, 2020.

[26] Shangqi Lai, Xingliang Yuan, Shifeng Sun, Joseph K. Liu, Yuhong Liu, and Dongxi Liu. GraphSE^2: An encrypted graph database for privacy-preserving social search. In *Proc. of ACM AsiaCCS*, 2019.

[27] Youhuan Li, Lei Zou, M. Tamer Özsu, and Dongyan Zhao. Space-efficient subgraph search over streaming graph with timing order constraint. *IEEE Transactions on Knowledge and Data Engineering*, 34(9):4453–4467, 2022.

[28] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*, pages 277–346. 2017.

[29] Brendan D McKay and Adolfo Piperno. Practical graph isomorphism, ii. *Journal of symbolic computation*, 60:94–112, 2014.

[30] Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In *Proc. of ACM CCS*, 2018.

[31] Mozilla Security Blog. Next steps in privacy-preserving Telemetry with Prio. https://blog.mozilla.org/security/2019/06/06/next-steps-in-privacy-preserving-telemetry-with-prio/, 2019.

[32] Kameshwar Munagala and Abhiram Ranade. I/O-complexity of graph algorithms. In *Proc. of SODA*, 1999.

[33] Takashi Nishide and Kazuo Ohta. Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In *proc. of PKC*, 2007.

[34] PIXNET on AWS. Aws case study: PIXNET. https://aws.amazon.com/solutions/case-studies/pixnet/, 2014.

[35] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 11(12):1876–1888, 2018.

[36] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. SiRnn: A math library for secure RNN inference. In *Proc. of IEEE S&P*, 2021.

[37] Zihao Shan, Kui Ren, Marina Blanton, and Cong Wang. Practical secure computation outsourcing: A survey. *ACM Computing Surveys*, 51(2):1–40, 2018.

[38] Sijun Tan, Brian Knott, Yuan Tian, and David J Wu. CryptGPU: Fast privacy-preserving machine learning on the GPU. In *Proc. of IEEE S&P*, 2021.

[39] Songlei Wang, Yifeng Zheng, Xiaohua Jia, Hejiao Huang, and Cong Wang. OblivGM: Oblivious attributed subgraph matching as a cloud service. *IEEE Transactions on Information Forensics and Security*, 17:3582–3596, 2022.

[40] Zuan Wang, Xiaofeng Ding, Hai Jin, and Pan Zhou. Efficient secure and verifiable location-based skyline queries over encrypted data. *Proceedings of the VLDB Endowment*, 15(9):1822–1834, 2022.

[41] Jean-Luc Watson, Sameer Wagh, and Raluca Ada Popa. Piranha: A GPU platform for secure computation. In *Proc. of USENIX Security*, 2022.

[42] Lyu Xu, Byron Choi, Yun Peng, Jianliang Xu, and Sourav S Bhowmick. A framework for privacy preserving localized graph pattern query processing. *Proceedings of the ACM on Management of Data*, 1(2):1–27, 2023.

[43] Quan Yuan, Zhikun Zhang, Linkang Du, Min Chen, Peng Cheng, and Mingyang Sun. PrivGraph: Differentially private graph data publication by exploiting community information. In *Proc. of USENIX Security*, 2023.

[44] Xinyi Zhang, Qichen Wang, Cheng Xu, Yun Peng, and Jianliang Xu. FedKNN: Secure federated k-nearest neighbor search. In *Proc. of ACM SIGMOD*, 2024.

A The Underlying Plaintext Algorithm

Algorithm 3 gives the plaintext graph pattern detection process underlying the security design of GraphGuard, which is built on the framework of the state-of-the-art scheme [27].

B Computational Benefit from Outsourcing

To demonstrate the computational benefit of outsourcing, we compare the running time of the regulator in GraphGuard with that under the case of local graph pattern detection over plaintext streaming graph. Here, we construct the plaintext baseline based on the state-of-the-art work [27]. We use the CAIDA dataset, with varying query pattern size $|\mathbb{Q}| \in \{6, 9, 12, 15, 18\}$ and a window size of 5×10^4. The time taken by the decomposition and encryption of a query pattern in GraphGuard varies in $\{0.29, 0.44, 0.72, 0.83, 1.27\}$ seconds accordingly. Note that such cost is *one-off*. Besides, when a new edge emerges, the local encryption time is not more than 1 millisecond on average. In comparison, in the non-outsourcing setting, the time requires by local detection

Algorithm 3 Time-Constrained Graph Pattern Detection

Input: A snapshot $\mathbb{G}_t$, TC-subquery patterns $\{\mathbb{Q}_d\}_{d\in[D]}$, and compatibility order constraints $\{o_c\}_{c\in[C]}$.
Output: The detection result set $\mathcal{R}$.

for each TC-subquery pattern $\mathbb{Q}_d, d \in [D]$ **do**
  for each edge label l_y in $\mathbb{Q}_d$ **do**
    $\mathcal{E}_y \leftarrow \emptyset$.// `Initialize` l_y`'s matched edge set.`
    for each edge e_x in the snapshot $\mathbb{G}_t$ **do**
      If e_x's label is identical to l_y, then $\mathcal{E}_y$.add(e_x).
    end for
  end for
  $\mathcal{CP}_d \leftarrow \emptyset$. // `Initialize the candidate partial match set.`
  for each set of edges from different $\mathcal{E}_y$ **do**
    If the temporal order of the edges complies with the timing order constraints specified by $\mathbb{Q}_d$, the subgraph P_l formed by them is added to $\mathcal{CP}_d$.
  end for
  $\mathcal{P}_d \leftarrow \emptyset$. // `Initialize the partial match set.`
  for each candidate partial match $P_l \in \mathcal{CP}_d$ **do**
    If P_l's structure matches that of $\mathbb{Q}_d$, $\mathcal{P}_d$.add(P_l).
  end for
end for
$\mathcal{C} \leftarrow \emptyset$. // `Initialize the candidate match set.`
for each set of partial matches from $\mathcal{P}_d, d \in [D]$ **do**
  If the temporal order of the partial matches complies with $\{o_c\}$, the graph R formed by them is added to $\mathcal{C}$.
end for
$\mathcal{R} \leftarrow \emptyset$. // `Initialize the result set.`
for each candidate match $R \in \mathcal{C}$ **do**
  If R's structure matches that of $\mathbb{Q}$, then $\mathcal{R}$.add(R).
end for
return The detection results $\mathcal{R}$.

over a snapshot varies in $\{1.13, 1.43, 1.96, 2.34, 2.59\}$ seconds, when a new edge emerges and the window slides. Such local cost indeed will grow with the increase in (1) the window size, (2) the query pattern size, and (3) the number of timing order constraints in the query pattern. While the above results showcase the computational benefit from outsourcing, we emphasize that there are many other well-known benefits that boosts the trend of graph service outsourcing.

C Leakage Mitigation

To mitigate the leakage of edge timestamps, FE can encrypt the timestamps and upload encrypted new edges to S_{123} in a batch fashion. With this, the timestamps of new edges will be obfuscated in the view of the cloud servers. When the timestamps are encrypted, the snapshot can be obliviously retrieved as follows. Firstly, we can leverage the technique of public interval query on secret-shared values [8] to obliviously retrieve the encrypted edges whose timestamps fall within the sliding window. Then, S_{123} can obliviously sort the encrypted edges based on their encrypted timestamps, producing the encrypted snapshot as desired. Additionally, GraphGuard can be extended to protect the number of edges in the streaming graph. Specifically, we can have users generate dummy edges in the form of $(\llbracket -\mathbf{1} \rrbracket, \llbracket -\mathbf{1} \rrbracket, \llbracket \mathbf{l}_x \rrbracket, t_x)$, similar to what the cloud servers do during the oblivious dummy edges padding process (as described in Section 4.3.1). The tailored format of dummy edges ensures no accuracy degradation, as will be analyzed in Appendix E.

D Securely Handling Vertex/Edge Deletion

GraphGuard can be extended to securely handle vertex/edge deletion using Oblivious RAM (ORAM) techniques (e.g., [10]). In particular, the oblivious write operation can be utilized, which allows the servers to obliviously write a secret-shared value at a secret-shared index. To integrate the operation into GraphGuard for handling edge deletion, each edge needs to be assigned a unique ID which is encrypted under RSS. When the regulator intends to delete a specific edge with ID i, it can let the servers obliviously write -1 to the encrypted IDs of the endpoints of the edge with ID i. Since the vertices with ID -1 do not connect with any other true vertex, the updated edges will not contribute to any time-constrained match and can be regarded as deleted. Similarly, to delete a vertex with ID j, the regulator can let the servers obliviously write -1 to the encrypted IDs of the endpoints of the edges where one endpoint's ID is j. In other words, the operation deletes all edges related to the vertex with ID j, which is equivalent to deleting the vertex with ID j.

E Accuracy Analysis

We emphasize that GraphGuard does not compromise the accuracy of pattern detection. While S_1 and S_2 pad encrypted dummy edges in each snapshot to obscure the true edges, these dummy edges are specifically crafted in the form of $\{(-1,-1,l,t_x)\}$ (as described in Section 4.3.1). This ensures that the dummy edges are disconnected from any true edges, as none of the true edges have endpoints with an ID of -1. Therefore, the structure checking performed on any candidate match involving dummy edges by Eq. 5, will necessarily yield $\llbracket \omega \rrbracket = \llbracket 1 \rrbracket$. This is due to the fact that the dummy edges do not connect with any other true edges. Therefore, there will always be a discrepancy between the EAM of any subgraph containing dummy edges and the EAM of the query pattern. Therefore, the dummy edges do not compromise the accuracy.

F Proof of Proposition 2

We first prove that in the view of S_2, GraphGuard can provide (ε, δ)-edge DP for the frequency of edge labels. Let $\mathbb{G}$ and $\mathbb{G}'$ be two neighboring graphs that differ in only one edge, specifically the edge with label l. Let f_l and f'_l be the frequencies of edges labeled l in the snapshots $\mathbb{G}_t$ of $\mathbb{G}$ and $\mathbb{G}'_t$ of $\mathbb{G}'$, respectively. We have $|f_l - f'_l| \leq 1$ as $\mathbb{G}$ and $\mathbb{G}'$ are two neighboring

graphs. If the noise n_l drawn for edge label l by S_1 (i.e., line 3 of Subroutine 1) is *non-negative*, the probability to output the same noisy frequency $\hat{f}$ from $\mathbb{G}_t$ and $\mathbb{G}'_t$ is bounded by

$$\frac{Pr[\hat{f}-f_l]}{Pr[\hat{f}-f'_l]} = \frac{e^{\frac{-\varepsilon\cdot|\hat{f}-f_l-\mu|}{1}}}{e^{\frac{-\varepsilon\cdot|\hat{f}-f'_l-\mu|}{1}}} = e^{\frac{\varepsilon\cdot(|\hat{f}-f'_l-\mu|-|\hat{f}-f_l-\mu|)}{1}} \leq e^{\frac{\varepsilon|f'_l-f_l|}{1}} \leq e^{\varepsilon}.$$

The probability of drawing a *negative* noise from $Lap(\varepsilon,\delta,1)$ is [20]: $Pr[x<0] = \sum_{x=-1}^{-\infty} \frac{e^{\varepsilon}-1}{e^{\varepsilon}+1}\cdot e^{-\varepsilon\cdot(x-\mu)} = \frac{e^{-\mu\cdot\varepsilon}}{e^{\varepsilon}+1}$. With the setting for μ in Eq. 2, we have $Pr[x<0] = \frac{e^{-(-\frac{\ln[(e^{\varepsilon}+1)\cdot(1-\delta)]}{\varepsilon})\cdot\varepsilon}}{e^{\varepsilon}+1} = 1-\delta$. Hence, with $1-\delta$, the probability that S_2 views the same noisy frequency $\hat{f}$ from $\mathbb{G}_t,\mathbb{G}'_t$ is bounded by e^{ε}, which satisfies (ε,δ)-edge DP in Definition 5.

Since S_2 performs the same process as S_1, in the view of S_1, GraphGuard can also provide (ε,δ)-edge DP. Based on the composition theorem of DP [18], in the view of S_3, GraphGuard can provide $(2\varepsilon,2\delta)$-edge DP. Therefore, in the view of S_{123}, GraphGuard can provide $(2\varepsilon,2\delta)$-edge DP.

G Proof of Proposition 3

Since the roles of S_{123} in GraphGuard are symmetric, except that S_3 receives the secret shares of dummy edges twice, it suffices to prove the existence of simulator for S_3. Note that during the Setup and Append phases, S_3 receives only the secret shares of private values in the query pattern $\mathbb{Q}$ and the streaming graph $\{e_x\}$, along with the timestamps $\{t_x\}$. So we can trivially construct the simulator by invoking the RSS simulator. The Detection is realized by Algorithm 1 (denoted as secDET) and Algorithm 2 (denoted as secCHK). Since they are invoked in order and their inputs/outputs are secret shares, we analyze the existence of their simulators separately [13].

Simulator for secDET**.** secDET consists of four subcomponents: oblivious dummy edges padding olivPad, secure equality test secTest, candidate partial matches construction Constr, and secure candidate partial matches filtering secFlt. We analyze the existence of their simulators in turn.

-Simulator for olivPad**.** At the beginning of olivPad (Subroutine 1), S_3 holds $\{(\langle\mathbf{sid}_x\rangle_{\{3,1\}}, \langle\mathbf{eid}_x\rangle_{\{3,1\}}, \langle\mathbf{l}_x\rangle_{\{3,1\}}, t_x)\}, \varepsilon, \delta$. Later, S_3 receives secret shares of the dummy edges from S_1 and S_2 (lines 9 and 11), and the simulator can be trivially constructed by invoking the RSS simulator [3]. Then S_3 receives secret shares during the execution of oblivious sorting, and the simulator can be constructed by invoking the simulator of oblivious sorting [5].

-Simulator for secTest**.** Note that throughout the execution of secTest (Subroutine 2), S_3 does not receive any information. After each execution of secTest, S_3 receives the secret shares $\langle\omega\rangle_1$ and $\langle\omega\rangle_2$ from S_1 and S_2, respectively, to reveal ω (line 6 of Algorithm 1). Based on the security of RSS [3], $\langle\omega\rangle_1, \langle\omega\rangle_2$ are uniformly random in S_3's view. Additionally, due to the padding of a random number of encrypted dummy edges with random timestamps, the revealed bit-string $\{\omega\}$, in S_3's view, is also uniformly random given $\{\hat{f}\}$.

-Simulator for Constr**.** Note that in Constr's execution (Subroutine 3), S_3 does not receive any information from either S_1 or S_2. Therefore, the simulator exists.

-Simulator for secFlt**.** At the beginning of secFlt (Subroutine 4), S_3 holds $\langle\mathcal{CP}_d\rangle_{\{3,1\}}$ and $\langle\mathbf{M}_d\rangle_{\{3,1\}}$. Later, S_3 receives secret shares during the execution of Eqs. 4 and 5. Since these equations involve basic operations in the RSS domain, the simulator can be constructed by invoking the RSS simulator [3]. S_3 then receives the shares $\langle\omega\rangle_1, \langle\omega\rangle_2$ from S_1 and S_2, respectively, to reveal ω (line 5 of Subroutine 4). The simulator can simulate the revealed bit-string $\{\omega\}$ as follows. Firstly, the simulator holds the number of candidate partial matches $|\mathcal{CP}|$, which corresponds to the size of $\{\omega\}$. Since the candidate partial matches are produced from $[\![\hat{\mathbb{G}}_t]\!]$, which is produced from $[\![\mathbb{G}_t]\!]$ after oblivious dummy edges padding, the positions of 0s in $\{\omega\}$ are random. Therefore, the simulator can first generate an all-ones string with a length of $|\mathcal{CP}|$, and then randomly select $|\mathcal{P}|$ (i.e., the number of partial matches) positions in the string to replace the 1s with 0s.

Simulator for secCHK**.** At the beginning of secCHK, S_3 holds $\langle\mathcal{P}_d\rangle_{\{3,1\}}, d\in[D]$, $\langle\mathbf{M}_d\rangle_{\{3,1\}}$, and $\{(y_1,y_2,\langle\theta\rangle_{\{3,1\}})\}$. Later, S_3 receives shares in the execution of lines 8-12. Note that lines 8-11 involve basic operations in the RSS domain. S_3 then receives the shares $\langle\chi\rangle_1$ and $\langle\chi\rangle_2$ from S_1 and S_2, respectively, to reveal χ (line 12). The simulator can simulate $\{\chi\}$ by the same method as the simulator for secFlt described above, and inputting the number of detection results $|\mathcal{R}|$.

Since S_3 does not view any information during the Output phase, we do not need to simulate its view in this phase.

H Analysis of Hiding Search Access Patterns

Hiding the search pattern. Given an encrypted query pattern $[\![\mathbb{Q}]\!]$, $\mathsf{S}_\alpha, \alpha\in\{1,2,3\}$ only receives the secret shares $\langle\mathbf{M}\rangle_{\{\alpha,\alpha+1\}}, \langle\mathcal{L}\rangle_{\{\alpha,\alpha+1\}}, \langle\theta\rangle_{\{\alpha,\alpha+1\}}$. The security of RSS [3] guarantees that even encrypting the same value multiple times will result in different secret shares indistinguishable from uniformly random values. Therefore, S_{123} cannot determine whether a new query pattern has been issued before, except by knowing whether the decompositions $\mathcal{Y}$ have been used before. However, since FE has the freedom to assign indexes to the edges in the query pattern and decompose it, S_{123} cannot determine whether a new query pattern has been issued before based on $\mathcal{Y}$.

Hiding the access pattern. Recall that before S_{123} fetch the matched edges $\{[\![\mathcal{E}_y]\!]\}$ from each snapshot $[\![\mathbb{G}_t]\!]$, they first obliviously sort $[\![\mathbb{G}_t]\!]$ (after padding dummy edges) based on their timestamps. Since the timestamps of dummy edges are random, S_{123} cannot determine which edges in $[\![\mathbb{G}_t]\!]$ after padding and sorting are dummy. Hence, they cannot determine which edges in the original snapshot $[\![\mathbb{G}_t]\!]$ are matched edges, i.e., GraphGuard can hide the access pattern.