Adversarial VNet Embeddings: A Threat for ISPs?

Yvonne-Anne Pignolet[1], Stefan Schmid[2], Gilles Tredan[3]
[1] ABB Corporate Research, Switzerland; [2] T-Labs & TU Berlin, Germany; [3] CNRS-LAAS, France

***Abstract*—This paper demonstrates that virtual networks that are dynamically embedded on a given resource network may constitute a security threat as properties of the infrastructure—typically a business secret—are disclosed. We initiate the study of this new problem and introduce the notion of *request complexity* which captures the number of virtual network embedding requests needed to fully disclose the infrastructure topology. We derive lower bounds and present algorithms achieving an asymptotically optimal request complexity for the important class of tree and cactus graphs (complexity $\Theta(n)$) as well as arbitrary graphs (complexity $\Theta(n^2)$).**

I. Introduction

An Internet Service Provider's (ISP) network infrastructure and its properties often constitute a critical business secret, not only because the smarter investment of a given resource budget yields a competitive advantage, but also because the discovery of, e.g., bottlenecks, may be exploited for attacks or bad publicity. Hence, providers around the world are often reluctant to open the infrastructure to novel technologies and applications that might lead to information leaks.

This paper raises the question whether today's virtualization trend in the Internet, and especially the *network virtualization* [2] paradigm, can be used to obtain information about the infrastructure. Network virtualization is a novel networking paradigm which allows customers to request *virtual networks (VNets)* on demand. VNets can come in different flavors and with different specifications, and can also provide QoS guarantees such as a minimal bandwidth (given that corresponding traffic shaping and reservation policies are in place). Although VNets appear as dedicated and "real" networks to their users, several VNets can be *embedded* (i.e., realized) over the same infrastructure network (referred to as the *substrate network*); network virtualization technology therefore enables the reuse of substrate resources.

Basically a VNet defines a set of virtual nodes (e.g., virtual machines) interconnected via virtual links according to the specified VNet topology over a substrate network. In this paper we consider VNet requests which do not impose any location constraints on where the virtual nodes are mapped to. One can take advantage of this flexibility in the VNet specification to optimize the VNet embedding. The set of virtual nodes and virtual links forming the VNet can hence be realized on arbitrary substrate nodes and paths, respectively.

Contribution. This paper initiates the study of a new problem, the discovery of a substrate topology through repeated VNet embedding requests. From a theoretical side, our model differs from existing topology discovery problems as the requests come in the form of entire *graphs* instead of paths only.

To measure how quickly a topology can be disclosed, we pursue an algorithmic approach and propose request strategies that reveal the substrate topology. These algorithms are evaluated on their *request complexity* which counts the number of VNet requests issued. Each request has the following simple form: *Is graph G (the VNet or "guest graph") embeddable in graph H (the substrate or "host graph")?*

We show that the request complexity of trees or even of cactus graphs is $\Theta(n)$, while arbitrary graphs have a request complexity of $\Theta(n^2)$, where n is the number of nodes in the substrate.

Related Work. For a survey on network virtualization, the reader is referred to [2]. While there is a large body of literature on how to embed VNets efficiently (e.g., [3], [7], [9]), we focus on the orthogonal question of what can be learned about a provider's infrastructure by such embeddings in this paper. For example, in the context of clouds, it has been shown that VM location information has security implications and that VM locations has been exploited for collocation attacks. [6]

Traceroute based topology inference [4] and *network tomography* [1] problems differ from our model as there, the exploration is inherently path-based.

II. VNet Embedding

We first introduce the VNet topology discovery problem and subsequently describe our algorithmic approach.

VNet Embedding. Our formal setting consists of two entities: a *customer* (the "adversary") that issues virtual network (VNet) requests and a *provider* that performs the access control and the embedding of VNets. We model the virtual network requests as simple, undirected, weighted graphs $G = (V, E, w)$ (the *guest graph*) where V denotes the virtual nodes and E denotes the virtual edges connecting nodes in V; the weight function w specifies capacity requirements, i.e., $w(v)$ denotes the resource *demand* for node $v \in V$ (e.g., computation or storage), and $w(e)$ denotes the demand for edge $e \in E$ (e.g., bandwidth).

Similarly, the infrastructure network is given as a weighted undirected graph $H = (V, E, w)$ (the so-called *host graph* or *substrate*) as well, where V denotes the set of substrate nodes, E is the set of substrate links, and w is a capacity function describing the available resources on a given node or edge. Without loss of generality, we assume that there are no parallel edges or self-loops neither in VNet requests nor in the substrate, and that H is connected.

In this paper we assume that besides the resource demands, the VNet requests do not impose any mapping restrictions, i.e., a virtual node can be mapped to *any* substrate node, and we assume that a virtual link connecting two substrate nodes can be mapped to an entire (but single) *path* on the substrate as long as the demanded capacity is available. These assumptions are typical for virtual networks. [2]

A virtual link which is mapped to more than one substrate link however can entail certain costs at the *relay nodes*, the substrate nodes which do not constitute endpoints of the virtual link and merely serve for forwarding. For example, this cost may represent a header lookup cost and may depend on the packet rate of the communication. However, depending on the application, the cost can also be more complex, for instance in case of a VNet which requires additional functionality at the backbone routers, or to implement an intrusion detection system. We model these kinds of costs with a parameter $\epsilon > 0$ (per link). Moreover, we also allow multiple virtual nodes to be mapped to the same substrate node if the node capacity allows it; we assume that if two virtual nodes are mapped to the same substrate node, the cost of a virtual link between them is zero. With these definitions, we can formalize VNet embeddings.

Definition 1 (Embedding π, Relation $\mapsto$)**.** *An* embedding *of a graph $A = (V_A, E_A, w_A)$ to a graph $B = (V_B, E_B, w_B)$ is a mapping $\pi : A \to B$ where every node of A is mapped to exactly one node of B, and every edge of A is mapped to a path of B. That is, π consists of a node $\pi_V : V_A \to V_B$ and an edge mapping $\pi_E : E_A \to P_B$, where P_B denotes the set of paths. We will refer to the set of virtual nodes embedded on a node $v_B \in V_B$ by $\pi_V^{-1}(v_B)$; similarly, $\pi_E^{-1}(e_B)$ describes the set of virtual links passing through $e_B \in E_B$ and $\pi_E^{-1}(v_B)$ describes the virtual links passing through $v_B \in V_B$ with v_B serving as a relay node.*

To be valid, the embedding π has to fulfill the following properties: (i) Each node $v_A \in V_A$ is mapped to exactly one node $v_B \in V_B$ (but given sufficient capacities, v_B can host multiple nodes from V_A). (ii) Links are mapped consistently, i.e., for two nodes $v_A, v'_A \in V_A$, if $e_A = \{v_A, v'_A\} \in E_A$ then e_A is mapped to a single (possibly empty and undirected) path in B connecting nodes $\pi(v_A)$ and $\pi(v'_A)$. A link e_A cannot split into multiple paths. (iii) The capacities of substrate nodes are not exceeded: $\forall v_B \in V_B$: $\sum_{u \in \pi_V^{-1}(v_B)} w(u) + \epsilon \cdot |\pi_E^{-1}(v_B)| \leq w(v_B)$. (iv) The capacities in E_B are respected as well, i.e., $\forall e_B \in E_B$: $\sum_{e \in \pi_E^{-1}(e_B)} w(e) \leq w(e_B)$.

If there exists such a valid embedding mapping π, we say that graph A can be embedded in B, denoted by $A \mapsto B$. Hence, $\mapsto$ denotes the VNet embedding relation*.*

Definition 2 (Embedding Cost)**.** *The cost associated with an embedding π is denoted by $Cost(\pi) = \sum_{v_A \in V_A} w(v_A) + \sum_{e_A \in E_A} w(e_A) \cdot |\pi^{-1}(e_A)| + \epsilon \cdot \sum_{v_B \in V_B} |\pi_E^{-1}(v_B)|$.*

The provider is flexible where to embed a VNet as long as a valid mapping is chosen.

Request Complexity. In order to define the complexity of substrate topology discovery, we assume the perspective of a customer (an "adversary") that seeks to disclose the (fixed) infrastructure topology of a provider with a minimal number of requests. These requests (and the answers to them) are the only means of obtaining information. As a performance measure, we introduce the notion of *request complexity*, i.e., the number of VNet requests which have to be issued until a given network is fully discovered, i.e., all nodes, edges and capacities are known to the adversary.

We are interested in algorithms that "guess" the target topology H (the host graph) among the set $\mathcal{H}$ of possible substrate topologies allowed by the model. Concretely, we assume that given a VNet request G (a guest graph), the substrate provider always responds with *an honest (binary) reply R* informing the customer whether the requested VNet G is embeddedable on the substrate H. In the following, we will use the notation `request`(G, H) to denote such an embedding request of G to H, and the provider will answer with the binary information whether G is embeddable in H (short: $G \mapsto H$). Based on this reply, the customer may then decide to ask the provider to embed the corresponding VNet G on H, or it may not embed it and continue asking for other

VNets (i.e., the customer does not pay for requests). Note that considering binary replies results in a worst case approach: we assume that very little information is leaked from the provider. Another plausible model is the valued reply model, where the provider returns Cost(`request`(G, H)).

Let ALG be an algorithm that issues a series of request requests $G_1, \ldots, G_t$ each consisting of a request graph to reveal H. The *request complexity* to infer the topology is measured in the number of requests t (in the worst case) until ALG issues a request G_t which is isomorphic to H and *terminates* (i.e., ALG knows that $H = G_t$ and does not issue any further requests).

In order to focus on the topological aspects, we assume that the substrate graph elements in H all have a constant capacity of one unit. Moreover, if not state otherwise, we assume that the requested nodes and links have a demand of one unit as well. In fact, our general lower bounds show that given the constant capacities of H, using alternative demands does not make the discovery faster. Clearly, with an VNet request approach only the parts of the substrate that are not occupied by other customers can be discovered. If simultaneously other customers' requests are fulfilled that inferrable part of the substrate changes.

III. Adversarial Topology Discovery

This section presents tight request complexity bounds for the discovery of three graph classes.

A. Trees

First, note that even if a topology discovery algorithm ALG is initially not aware that the substrate $H \in \mathcal{H}$ can only have a tree structure, it can discover the absence of cycles in the topology with a single request: If ALG asks for a triangle network (i.e., a complete graph K_3 consisting of three virtual nodes) with unit virtual node and link capacities, it can be embedded if and only if H contains a cycle. Once it is known that the set of possible infrastructure topologies (or host graphs) $\mathcal{H}$ is restricted to trees, the algorithm described in this section can be used to discover them. Moreover, if $H \in \mathcal{H}$ contains cycles, our algorithm computes a spanning tree of H.

The tree discovery algorithm TREE (see Algorithm 1 for the formal listing) described in the following is based on the idea of incrementally growing the request graph by adding *longest chains* (i.e., "branches" of the tree). Intuitively, such a longest chain of virtual nodes will serve as an "anchor" for extending further branches in future requests: since the chain is maximal and no more nodes can be embedded, the number of virtual nodes along the chain must equal the number of substrate nodes on the corresponding substrate path. The endpoints of the chain thus cannot have any additional neighbors and must be tree leafs (we will call these nodes *explored*), and we can recursively explore the longest branches of the so-called *pending nodes* discovered along the chain.

More concretely, TREE first discovers the overall longest (cycle-free) chain of nodes in the substrate tree by performing binary search on the length of the maximal embeddable path. This is achieved by requesting, in request R_i, a VNet of 2^i linearly connected virtual nodes (of unit node and link capacities); in Algorithm 1, we refer to a single virtual link connecting two virtual nodes by a *chain* C, and a sequence of j chains by C^j. The first time a path of the double length 2^i is not embeddable, TREE asks for the longest embeddable chain with 2^{i-1} to $2^i - 1$ virtual nodes; and so on. Once the longest chain is found, its end nodes are considered *explored* (they cannot have any additional neighbors due to the longest chain property), and all remaining virtual nodes along the longest chain are considered *pending* (set $\mathcal{P}$): their tree branches still need to be explored. TREE then picks an arbitrary pending node v and seeks to attach a maximal chain ("branch") analogously to the procedure above, except for that the node at the chain's origin is left pending until no more branches can be added. The scheme is repeated recursively until there are no pending nodes left. Formally, in Algorithm 1, we write GvC to denote that a chain C is added to an already discovered graph G at the virtual node v.

Algorithm 1 Tree Discovery: TREE

```
1: $G := \{\{v\}, \emptyset\}$ /* current request graph */
2: $\mathcal{P} := \{v\}$ /* pending set of unexplored nodes*/
3: **while** $\mathcal{P} \neq \emptyset$ **do**
4:     choose $v \in \mathcal{P}$, $S :=$*exploreSequence*$(v)$
5:     **if** $S \neq \emptyset$ **then**
6:         $G := GvS$, add all nodes of $S$ to $\mathcal{P}$
7:     **else**
8:         remove $v$ from $\mathcal{P}$
```

```
*exploreSequence*$(v)$
1: $S := \emptyset$
2: **if** request$(GvC, H)$ **then**
3:     find max $j$ s.t. $GvC^j \mapsto H$ (binary search)
4:     $S := C^j$
5: **return** $S$
```

Theorem 1. *Algorithm* TREE *is correct and has a request complexity of* $\Theta(n)$*, where* n *is the number of substrate nodes. This is asymptotically optimal.*

Proof: Correctness: Since the substrate network is connected, each node can be reached by a path from any other node. As the algorithm explores each path attached to a discovered node until no more nodes can be added, every node is eventually found. Since a tree is cycle-free, this also implies that the set of discovered edges is complete. *Complexity (upper bound)*: We observe that our algorithm has the property that at time t, it always ask for a VNet which is a strict super graph of any embeddable graph asked at time $t' < t$ (positive answer from the provider). Moreover, due to the exponential binary search construction, TREE issues $O(\log \ell)$ requests to discover a chain consisting of ℓ links. The cost of exploring a path can be distributed among its constituting links, thus we have an accounting scheme which shows that the amortized cost per link is constant: As there are at most $n-1$ links in a tree, the total number of requests due to the link discovery is linear in n as well. In order to account for requests at nodes that do not have any unexplored neighbors and lead to marking a node explored (at most one request per node), $O(n)$ requests need to be added. *Complexity (lower bound)*: The lower bound follows from the cardinality of the set of non-isomorphic trees, which is in the order of $2.96^n/n^{5/2}$ [5]. Since any discovery algorithm can only obtain a binary information for each request issued, a request cuts the remaining search space in (at most) half. Therefore, the request complexity of any algorithm is at least $\Omega(\log(2.96^n/n^{5/2})) = \Omega(n)$. ■

Observe that TREE has the nice property that if H is not a tree, TREE computes a spanning tree of H by extending maximal (cycle-free) branches from the nodes.

Corollary 1. TREE *determines a spanning tree of any graph with request complexity* $\Theta(n)$.

B. Arbitrary Graphs

Let us now turn to the general problem of inferring arbitrary substrate topologies. First note that even if the total number of substrate nodes is known, the adversary cannot simply compute the substrate edges by testing each virtual link between the node pairs: the fact that the corresponding virtual link can be embedded does not imply that a corresponding substrate link exists, because the virtual link might be mapped across an entire substrate *path*. Nevertheless, we will show in the following that a request complexity of $O(n^2)$ can be achieved; this is asymptotically optimal.

The main idea of our algorithm GEN is to build upon the TREE algorithm to first find a spanning tree (see Corollary 1). This spanning tree (consisting of pending nodes only) "reserves" the resources on the substrate nodes, such that they cannot serve as relay nodes for virtual links passing through them. Subsequently, we try to extend the spanning tree with additional edges. An arbitrary pending node u is chosen, and we try to add an edge to any other pending node v in the spanning tree. After looping over all pending nodes and adding the corresponding links, u is marked *explored*. GEN terminates when no more pending nodes are left. The lower bound is a consequence of the number of unlabelled nodes.

Theorem 2. *A general graph can be discovered with request complexity* $\Theta(n^2)$. *This is asymptotically tight.*

C. Cactus Graphs

So far we presented an asymptotically optimal tree discovery algorithm with request complexity $\Theta(n)$, and an optimal algorithm for general graphs with request complexity $\Theta(n^2)$. This raises the question whether a linear request complexity can only be achieved in acyclic graphs. At least our approach of first computing a spanning tree seems to be problematic when applied to cyclic graphs, as there are instances where $\Omega(n^2)$ requests are unavoidable to find just one additional cyclic edge. We will now show that it is still possible to infer more general topologies without sacrificing efficiency.

The *cactus graph* is a particularly interesting topology in the context of the Internet. (For example, the topologies collected in experiments such as *Rocketfuel* are often sparse but contain certain cycles along the backbone, and thus resemble the cactus graph [8].) Formally, every edge in the cactus graph belongs to at most one 2-connected component, i.e., the cactus graph does not contain any diamond graph shaped minors.

The underlying idea of our cactus discovery algorithm CACTUS is that in contrast to a tree where nodes are origins of simple paths (i.e., branches), a cactus node can be the origin of several sub-cactus graphs consisting of 1- and 2-connected components. That is, the resulting graph when collapsing one or several 2-connected components to a single node is a cactus as well, or even a tree if all components are collapsed. We use these properties to extend our tree algorithm to include cycle requests in addition to chains when exploring a pending virtual node. Concretely, instead of using longest chains as "anchor points" for extending an existing topology, we search, in each possible direction from a pending cactus node v for a maximal sequences of cycles (short: Y) and chains (short: C). Once such a sequence (or "*motif*") is found, our algorithm CACTUS attempts to discover the detailed structure of the chain/cycle sequence by inserting as many nodes on the chains and cycles as possible.

See Algorithm 2 for the formal listing. We use graph

Algorithm 2 Cactus Discovery: CACTUS

```
1: $G := \{\{v\}, \emptyset\}$ /* current request graph */
2: $\mathcal{P} := \{v\}$ /* pending set of unexplored nodes*/
3: while $\mathcal{P} \neq \emptyset$ do
4:     choose $v \in \mathcal{P}$, $S :=$ exploreSequence($v$)
5:     if $S \neq \emptyset$ then
6:         $G := GvS$, add all nodes of $S$ to $\mathcal{P}$
7:         for all $e \in S$ do edgeExpansion($e$)
8:     else
9:         remove $v$ from $\mathcal{P}$
```

exploreSequence(v)

```
1: $S := \emptyset$, $P'' := \emptyset$
2: if $GvYCY \mapsto H$ then
3:     find max $j$ s.t. $GvY^jCY \mapsto H$
4:     $S := Y^jCY$, $P' := \{C\}$
5:     while $P' \neq \emptyset$ do
6:         for all $C_i \in P'$ do
7:             $A := prefix(C_i, S)$, $B := postfix(C_i, S)$;
8:             if $GvACYCB \mapsto H$ then
9:                 find max $j, k$ s.t. $GvAC(Y^jC)^kB \mapsto H$
10:                for $l := 1, \ldots, k$ do
11:                    $P'' := P'' \cup \{C_l\}$
12:                $S := AC(Y^jC)^kB$
13:        $P' := P''$, $P'' := \emptyset$
14: if request($GvSY, H$) then
15:     find max $j$ s.t. $GvSY^j \mapsto H$
16:     $S := SY^j$
17: if request($GvSC, H$) then
18:     $S := SC$
19: return $S$
```

edgeExpansion(e)

```
1: let $u, v$ be the endpoints of edge $e$, remove $e$ from $G$
2: find max $j$ s.t. $GvC^ju \mapsto H$
3: $G := GvC^ju$, add newly discovered nodes to $\mathcal{P}$
```

grammar notation for the iterative exploration process.

Definition 3. *Let C denote a* Chain *($V = \{u, v\}, E = \{\{u, v\}\}$), i.e., a virtual edge that maps to a path on the substrate, and let Y denote a* cYcle *($V = \{u, v, w\}, E = \{\{u, v\}, \{u, w\}, \{w, v\}\}$), i.e., a virtual triangle that maps to a cycle in the substrate. Two chains, cycles or a chain and a cycle can be appended to each other, by merging one of their virtual nodes (due to symmetry it does not matter which nodes). This operation can be repeated to form an arbitrary* sequence of chains and cycles *S. By Y^j or C^j we refer to the* concatenation *of j cycles or chains respectively. Given a graph G containing nodes u, v, the notation $GvS(u)$ denotes a graph where the node $v \in G$ is merged with the first node in S, i.e., the edges of S are added to the set of G's edges and the corresponding nodes to the set of G's nodes; the node $u \in G$ is optional, and if it is stated, it means that in addition, node u is connected to the last node in S. Given a sequence S and a particular chain C belonging to this sequence then $prefix(C, S)$ and $postfix(C, S)$ denote the subsequences of S before and after this chain C.*

Theorem 3. CACTUS *discovers any cactus topology with request complexity $\Theta(n)$. This is tight.*

Proof sketch: The algorithm terminates as soon as all edges incident to nodes found so far (i.e., *pending* nodes) have been discovered. Consequently, we need to show that all nodes and all their adjacent edges are detected in order to prove correctness (i.e., there is a bijection between the edges in G and in H and thus it is not possible that a virtual edge connects two nodes that are not adjacent in the (sub)cactus graphs). Note that the algorithm maintains the invariant that $GvS \mapsto H$ at all times. As a consequence we can analyze the properties of S and thereby deduce properties of the substrate. For a sequence S discovered in $exploreSequence(v)$ the following properties hold: (i) no more Ys can be inserted (replace a Y by a YY or a C by a CYC), (ii) no chain can be inserted between two cycles (replace YY with YCY) and (iii) no C can be replaced by a Y. Thus the discovered sequence S cannot be extended with more cycles or chains between cycles. Based on these invariants it remains to show that the steps of the algorithm discover all nodes which are part of this sequence. For the complexity, we can again use an amortization scheme that assigns request costs to edges: an edge is assigned the cost of the requests where it is identified for the first time ■

REFERENCES

[1] A. Anandkumar, A. Hassidim, and J. Kelner. Topology discovery of sparse random graphs with few participants. In *Proc. SIGMETRICS*, 2011.

[2] M. K. Chowdhury and R. Boutaba. A survey of network virtualization. *Elsevier Computer Networks*, 54(5), 2010.

[3] G. Even, M. Medina, G. Schaffrath, and S. Schmid. Competitive and deterministic embeddings of virtual networks. In *Proc. ICDCN*, 2012.

[4] Y. A. Pignolet, G. Tredan, and S. Schmid. Misleading stars: What cannot be measured in the internet? In *Proc. DISC*, 2011.

[5] J. M. Plotkin and J. W. Rosenthal. How to obtain an asymptotic expansion of a sequence from an analytic identity satisfied by its generating function. *J. Austral. Math. Soc. Ser. A*, 1994.

[6] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proc. 16th ACM CCS*, pages 199–212, 2009.

[7] G. Schaffrath, S. Schmid, and A. Feldmann. Optimizing long-lived cloudnets with migrations. In *Proc. IEEE/ACM UCC*, 2012.

[8] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring isp topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, 2004.

[9] S. Zhang, Z. Qian, J. Wu, and S. Lu. An opportunistic resource sharing and topology-aware mapping framework for virtual networks. In *Proc. IEEE INFOCOM*, 2012.