NetHide: Secure and Practical Network Topology Obfuscation

Roland Meier and Petar Tsankov, *ETH Zurich;* Vincent Lenders, *armasuisse;* Laurent Vanbever and Martin Vechev, *ETH Zurich*

https://www.usenix.org/conference/usenixsecurity18/presentation/meier

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-939133-04-5

NetHide: Secure and Practical Network Topology Obfuscation

Roland Meier*, Petar Tsankov*, Vincent Lenders$^\diamond$, Laurent Vanbever*, Martin Vechev*
* *ETH Zürich* $^\diamond$*armasuisse*
nethide.ethz.ch

Abstract

Simple path tracing tools such as `traceroute` allow malicious users to infer network topologies remotely and use that knowledge to craft advanced denial-of-service (DoS) attacks such as Link-Flooding Attacks (LFAs). Yet, despite the risk, most network operators still allow path tracing as it is an essential network debugging tool.

In this paper, we present NetHide, a network topology obfuscation framework that mitigates LFAs while preserving the practicality of path tracing tools. The key idea behind NetHide is to formulate network obfuscation as a multi-objective optimization problem that allows for a flexible tradeoff between security (encoded as hard constraints) and usability (encoded as soft constraints). While solving this problem exactly is hard, we show that NetHide can obfuscate topologies at scale by only considering a subset of the candidate solutions and without reducing obfuscation quality. In practice, NetHide obfuscates the topology by intercepting and modifying path tracing probes directly in the data plane. We show that this process can be done at line-rate, in a stateless fashion, by leveraging the latest generation of programmable network devices.

We fully implemented NetHide and evaluated it on realistic topologies. Our results show that NetHide is able to obfuscate large topologies ($>$ 150 nodes) while preserving near-perfect debugging capabilities. In particular, we show that operators can still precisely trace back $>$ 90% of link failures despite obfuscation.

1 Introduction

Botnet-driven Distributed Denial-of-Service (DDoS) attacks constitute one of today's major Internet threats [1, 2, 5, 10]. Such attacks can be divided in two categories depending on whether they target end-hosts and services (volume-based attacks) or the network infrastructure itself (link-flooding attacks, LFAs).

Volume-based attacks are the simplest and work by sending massive amounts of data to selected targets. Recent examples include the 1.2 Tbps DDoS attack against Dyn's DNS service [6] in October 2016 and the 1.35 Tbps DDoS attack against GitHub in February 2018 [8]. While impressive, these attacks can be mitigated today by diverting the incoming traffic through large CDN infrastructures [23]. As an illustration, CloudFlare's infrastructure can now mitigate volume-based attacks reaching Terabits per second [18].

Link-flooding attacks (LFAs) [26, 38] are more sophisticated and work by having a botnet generate low-rate flows between pairs of bots or towards public services such that all of these flows cross a given set of network links or nodes, degrading (or even preventing) the connectivity for *all* services using them. LFAs are much harder to detect as: *(i)* traffic volumes are relatively small (10 Gbps or 40 Gbps attacks are enough to kill most Internet links [31]); and *(ii)* attack flows are indistinguishable from legitimate traffic. Representative examples include the Spamhaus attack which flooded selected Internet eXchange Point (IXP) links in Europe and Asia [4, 7, 12].

Unlike volume-based attacks, performing an LFA requires the attacker to know the topology *and* the forwarding behavior of the targeted network. Without this knowledge, an attacker can only "guess" which flows share a common link, considerably reducing the attack's efficiency. As an illustration, our simulations indicate that congesting an *arbitrary* link without knowing the topology requires 5 times more flows, while congesting a *specific* link is order of magnitudes more difficult.

Nowadays, attackers can easily acquire topology knowledge by running path tracing tools such as `traceroute` [17]. In fact, previous studies have shown that entire topologies can be precisely mapped with `traceroute` provided enough vantage points are used [37], a requirement easily met by using large-scale measurement platforms (e.g., RIPE Atlas [16]).

Existing works Existing LFA countermeasures either work *reactively* or *proactively*. Reactive measures dynamically adapt how traffic is being forwarded [25, 33] or have networks collaborating to detect malicious flows [31]. Proactive measures work by obfuscating the network topology so as to prevent attackers from discovering potential targets [28, 39, 40]. The problem with reactive countermeasures is the relative lack of incentives to deploy them: collaborative detection is only useful with a significant amount of participating networks, while dynamic traffic adaptation conflicts with traffic engineering objectives. In contrast, proactive approaches can protect each network individually without impacting normal traffic forwarding. Yet, they considerably lower the usefulness of path tracing tools [28, 39] such as `traceroute` which is the prevalent tool for debugging networks [24, 27, 37]. Further, they also provide poor obfuscation which can be easily broken with a small number of brute-force attacks [39, 40].

Problem statement Given the limitations of existing techniques, a fundamental question remains open: *is it possible to obfuscate a network topology so as to mitigate attackers from performing link flooding attacks while, at the same time, preserving the usefulness of path tracing tools?*

Key challenges Answering this question is challenging for at least three reasons:

1. The topology must be obfuscated with respect to any possible attacker location: attackers can be located anywhere and their tracing traffic is often indistinguishable from legitimate user requests.
2. The obfuscation logic should not be invertible and should scale to large topologies.
3. The obfuscation logic needs to be able to intercept and modify tracing traffic at line-rate. To preserve the troubleshooting-ability of network operators, tracing traffic should still flow across the correct physical links such that, for example, link failures in the physical topology are visible in the obfuscated one.

NetHide We present NetHide, a novel network obfuscation approach which addresses the above challenges. NetHide consists of two main components: *(i)* a usability-preserving and scalable obfuscation algorithm; and *(ii)* a runtime system, which modifies tracing traffic directly in the data plane.

The key technical insight behind NetHide is to formulate the network obfuscation task as a multi-objective optimization problem that allows for a flexible trade-off between security (encoded as hard constraints) and usability (soft constraints). We introduce two metrics to quantify the usability of an obfuscated topology: *accuracy* and *utility*. Intuitively, the accuracy measures the similarity between the path along which a flow is routed in the physical topology with the path that NetHide presents in the virtual topology. The utility captures how physical events (e.g., link failures or congestion) in the physical topology are represented in the virtual topology. To scale, we show that considering only a few randomly selected candidate topologies, and optimizing over those, is enough to find secure solutions with near-optimal accuracy and utility.

We fully implemented NetHide and evaluated it on realistic topologies. We show that NetHide is able to obfuscate large topologies (> 150 nodes) with marginal impact on usability. In fact, we show in a case study that NetHide allows to precisely detect the vast majority ($> 90\,\%$) of link failures. We also show that NetHide is useful when partially deployed: 40 % of programmable devices allow to protect up to 60 % of the flows.

Contributions Our main contributions are:

- A novel formulation of the network obfuscation problem in a way that preserves the usefulness of existing debugging tools (§3).
- An encoding of the obfuscation task as a linear optimization problem together with a random sampling technique to ensure scalability (§4).
- An end-to-end implementation of our approach, including an online packet modification runtime (§5).
- An evaluation of NetHide on representative network topologies. We show that NetHide can obfuscate topologies of large networks in a reasonable amount of time. The obfuscation has little impact on benign users and mitigates realistic attacker strategies (§6).

2 Model

We now present our network and attacker models and formulate the precise problem we address.

2.1 Network model

We consider layer 3 (IP) networks operated by a single authority, such as an Internet service provider or an enterprise. Traffic at this layer is routed according to the destination IP address. We assume that routing is deterministic, meaning that the traffic is sent along a single path between each pair of nodes. While this assumption does not hold for networks relying on probabilistic load-balancing mechanisms (e.g., ECMP [15]), it makes our attacker more powerful as all paths are assumed to be persistent and therefore easier to learn.

To deploy NetHide, we assume that some of the routers are programmable in a way that allows them to: *(i)* match

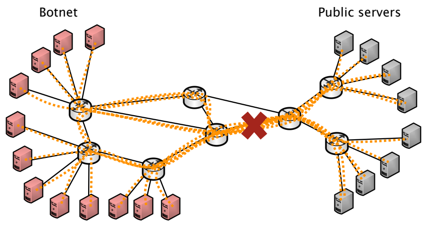


Figure 1: Link Flooding Attacks (LFAs) work by routing many legitimate low-volume flows over the same set of physical links in order to cause congestion. LFAs assume that the attacker can discover the network topology, usually using `traceroute`-like tracing.

Network components			
(Nodes)	N	$\subseteq$	$\mathcal{N} = \{n_1, \ldots, n_N\}$
(Links)	L	$\subseteq$	$N \times N$
(Forwarding tree)	T_n	$=$	(N, L_n), tree rooted at n
(Forwarding trees)	T	$=$	$\bigcup_{n \in N} T_n$
(Flows)	F	$\subseteq$	$N \times N$
Network topologies			
(Physical)	P	$=$	(N, L, T)
(Virtual)	V	$=$	(N', L', T')
			$N \subseteq N'$
Metrics			
(Flows per link)	$f(T,l)$	$=$	$\{(s,d) \in F \mid l \in T_d\}$
(Flow density)	$fd(T,l)$	$=$	$\lvert f(T,l) \rvert$
(Capacity)	c	$:$	$L \to \mathbb{N}$
(Accuracy)	acc	$:$	$((s,d), P, V) \mapsto [0,1]$
(Utility)	$util$	$:$	$((s,d), P, V) \mapsto [0,1]$

Figure 2: NetHide notation and metrics

on arbitrary IP Time-to-Live (TTL) values; *(ii)* change the source and destination addresses of packets (e.g., UDP packets for `traceroute`) depending on the original destination address and the TTL; and *(iii)* restore the original source and destination addresses when replies (e.g., ICMP packets) to modified packets arrive. Our implementation uses the P4 programming language [14], which fulfills the above criteria. Yet, NetHide could also be implemented on top of existing router firmware.

2.2 Attacker model

We assume an attacker who controls a set of hosts (e.g., a botnet) that can inject traffic in the network. The attacker's goal is to perform a *Link Flooding Attack* (LFA) such as Coremelt [38] or Crossfire [26]. The objective of these attacks is to isolate a network segment by congesting one or more links. The attacker aims to congest links by creating low-volume flows from many different sources (bots) to many destinations (public servers or other bots) such that all these flows cross the targeted links (illustrated in Fig. 1). An attacker's *budget* limits the number of flows she can run and we quantify the attacker's strength based on her budget. Because the additional traffic is low-volume, it is hard to separate it from legitimate (also low-volume) traffic. This makes detecting and mitigating LFA attacks a hard problem [41].

To mount an efficient and stealthy LFA, the attacker must know enough (source, destination) pairs that communicate via the targeted link(s). Otherwise, she would have to create so many flows that she no longer remains efficient. Similarly to [26, 38], we assume the attacker has no prior knowledge of the network topology. However, the attacker can learn the network topology using `traceroute`-like tracing techniques [17]. `traceroute` works by sending a series of packets (probes) to the destination with increasing TTL values. In response to these probes, each router along the path to the destination sends an ICMP time exceeded message. More specifically, `traceroute` leverages the fact that TTL values are decremented by one at each router, and that the first router to see a TTL value of 0 sends a response to the source of the probe. For example, a packet with TTL value of 3 sent from A to B will cause the third router along the path from A to B to send an ICMP time exceeded message to A. By aggregating paths between many host pairs, it is possible to determine the topology and the forwarding behavior of the network [37]. We remark that in addition to revealing forwarding paths, `traceroute`-like probes also disclose the Round-Trip Time (RTT), i.e., the time difference between the moment a probe is sent and the corresponding ICMP time exceeded message is received, which can be used as a side-channel to gain intuition about the feasibility of a (potentially obfuscated) path returned by `traceroute`.

Finally, we assume that the attacker knows everything about the deployed protection mechanisms in the network (including the ones presented in this paper) except their secret inputs and random decisions following Kerckhoff's principle [34].

2.3 Notation

We depict our notation and definitions in Fig. 2. We model a *network topology* as a graph with nodes $N \subseteq \mathcal{N}$, where $\mathcal{N}$ is the set of all possible nodes, and links $L \subseteq N \times N$. A *node* in the graph corresponds to a router in the network and a *link* corresponds to an (undirected) connection between two routers. NetHide allows to extend a topology with virtual nodes, i.e., nodes $v \in \mathcal{N} \setminus N$.

Given a node n, we use a tree $T_n = (N, L_n)$ rooted at n to model how packets are forwarded to n. We refer to this tree as a *forwarding tree*. For simplicity, we write $l \in T_n$ to denote that the link l is contained in the forwarding tree T_n, i.e., $T_n = (N, L_n)$ with $l \in L_n$. We use T to denote the set of all forwarding trees.

A *flow* $(s,d) \in F$ is a pair of a source node s and destination node d. Note that the budget of the strongest attacker is given by the total number $|F|$ of possible flows. We use $T_{s \to d}$ to refer to the path from source node s to destination node d according to the forwarding tree T_d. In the style of [26], we define the *flow density fd* for a link $l \in L$ as the number of flows that are routed via this link (in any direction). The maximum flow density that a link can handle without congestion is denoted by the link's *capacity* c. A topology (N, L, T) is *secure* if the flow density for any link in the topology does not exceed its capacity, i.e., $\forall l \in L : fd(T,l) \leq c(l)$. Note that no attacker (with any budget) can attack a secure topology as all links have enough capacity to handle the total number of flows from all the (source, destination) pairs in F.

2.4 Problem statement

We address the following *network obfuscation problem*: Given a physical topology P, the goal is to compute an obfuscated (virtual) topology V such that V is secure and is as similar as possible to P. In other words, the goal is to deceive the attacker with a virtual topology V. For the similarity between the physical topology P and the obfuscated topology V, we refer to §3 where we present metrics which represent the *accuracy* of paths reported by `traceroute` and the *utility* of link failures in P being closely represented in V.

We remark on a few important points. First, if P is secure, then the obfuscation problem should return P since we require that V is as similar as possible to P. Second, for any network and any attacker, the problem has a trivial solution since we can always come up with a network that has an exclusive routing path for each (source, destination) pair. However, for non-trivial notions of similarity, it is challenging to discover an obfuscated network V that similar to P.

3 NetHide

We now illustrate how NetHide can compute a secure and yet usable (i.e., "debuggable") obfuscated topology on a simple example depicted in Fig. 3. Specifically, we consider the task of obfuscating a network with 6 routers: $A, \ldots, F$ in which the core link (C,D) acts as bottleneck and is therefore a potential target for an LFA.

Inputs NetHide takes four inputs: *(i)* the physical network topology graph; *(ii)* a specification of the forwarding behavior (a forwarding tree for each destination according to the physical topology and incorporating potential link weights); *(iii)* the capacity c of each link (how many flows can cross each link before congesting it); along with *(iv)* the set of attack flows F to protect against. If the position of the attacker(s) is not known (the default), we define F to be the set of all possible flows between all (source,destination) pairs.

Given these inputs, NetHide produces an obfuscated virtual topology V which: *(i)* prevents the attacker(s) from determining a set of flows to congest any link; while *(ii)* still allowing non-malicious users to perform network diagnosis. A key insight behind NetHide is to formulate this task as a multi-objective optimization problem that allows for a flexible tradeoff between security (encoded as hard constraints) and usability (encoded as soft constraints) of the virtual topology. The key challenge here is that the number of obfuscated topologies grows exponentially with the network size, making simple exhaustive solutions unusable. To scale, NetHide only considers a subset of candidate solutions amongst which it selects a usable one. Perhaps surprisingly, we show that this process leads to desirable solutions.

Pre-selecting a set of secure candidate topologies NetHide first computes a random set of obfuscated topologies. In addition to enabling NetHide to scale, this random selection also acts as a secret which makes it significantly harder to invert the obfuscation algorithm.

NetHide obfuscates network topologies along two dimensions: *(i)* it modifies the topology graph (i.e., it adds or removes links); and *(ii)* it modifies the forwarding behavior (i.e., how flows are routed along the graph). For instance, in Fig. 3, the two shown candidate solutions V_1 and V_2 both contain two virtual links used to "route" flows from A to E and from B to F.

Selecting a usable obfuscated topology While there exist many secure candidate topologies, they differ in terms of *usability*, i.e., their perceived usefulness for benign users. In NetHide, we capture the usability of a virtual topology in terms of its *accuracy* and *utility*.

The *accuracy* measures the logical similarity of the paths reported when using `traceroute` against the original and against the obfuscated topology. Intuitively, a virtual topology with high accuracy enables network operators to diagnose routing issues such as sub-optimal routing. Conversely, tracing highly inaccurate topologies is likely to report bogus information such as traffic jumping between geographically distant points for no apparent reason. As illustration, V_2 is more accurate than V_1 in Fig. 3 as the reported paths have more links and routers in common with the physical topology.

The *utility* metric measures the physical similarity between the paths actually taken by the tracing packets in the physical and the virtual topology. Intuitively, utility

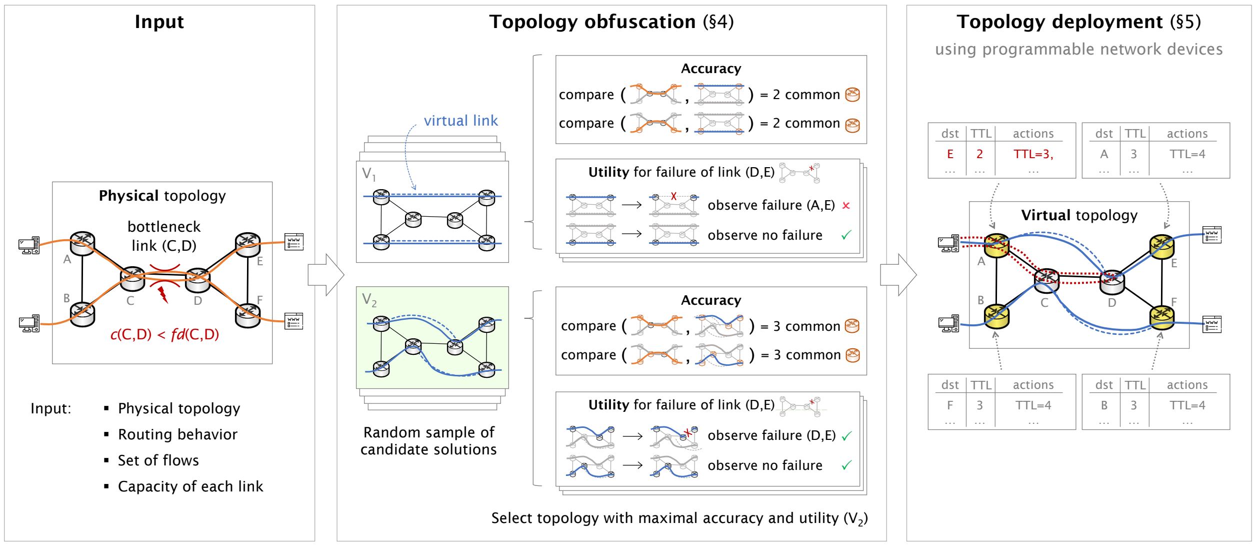


Figure 3: NetHide operates in two steps: *(i)* computing a secure *and* usable virtual topology; and *(ii)* deploying the obfuscated topology in the physical network.

captures how well events such as link failures or congestion in the physical topology are observable in the virtual topology. For instance, we illustrate that V_2 has a higher utility than V_1 in Fig. 3 by considering the failure of the link (D,E). Indeed, a non-malicious user would observe the failure of (D,E) (which is not obfuscated) when tracing V_2 while it would observe the failure of link (A,E) instead of (D,E) when tracing V_1.

Given V_1, V_2 and the fact that V_2 has higher accuracy and utility, NetHide deploys V_2.

Deploying the obfuscated topology NetHide obfuscates the topology at runtime by modifying tracing packets (i.e., IP packets whose TTL expires somewhere in the network). NetHide intercepts and processes such packets without impact on the network performance, directly in the data plane, by leveraging programmable network devices. Specifically, NetHide intercepts and possibly alters tracing packets at the edge of the network before sending them to the pretended destination in the physical network. That way, NetHide ensures that tracing packets traverse the corresponding physical links, and preserves the utility of `traceroute`-like tools. Observe that any alteration of tracing packets is reverted before they leave the network, which makes NetHide transparent. In contrast, simpler approaches which answer to tracing packets at the network edge or from a central controller (e.g., [28, 39]) render network debugging tools unusable.

Consider again Fig. 3 (right). If router A receives a packet towards E with TTL=2, this packet needs to expire at router D according to the virtual topology. Since the link between A and D does not exist physically, the packet needs to be sent to D via C, and it would thus expire at C. To prevent this and to ensure that the packet expires at D, NetHide increases the TTL by 1. Observe that, in addition to ensure the utility (see above), making the intended router answer to the probe also ensures that the measured round trip times are realistic (cf. §5).

4 Generating secure topologies

In this section, we first explain how to phrase the task of obfuscating a network topology as an optimization problem. We then present our implementation which consists of roughly 2000 lines of Python code and uses the Gurobi ILP solver [9].

4.1 Optimization problem

Given a topology $P = (N, L, T)$, a set of flows F, and capacities c, the network obfuscation problem is to generate a virtual topology $V = (N', L', T')$ such that: *(i)* V is secure; and *(ii)* the *accuracy* and *utility* metrics are jointly maximized; we define these metrics shortly.

NetHide generates V by modifying P in three ways: *(i)* NetHide can add *virtual nodes* to the topology graph that do not exist in P; *(ii)* NetHide adds *virtual links* to connect physical or virtual nodes in V; and *(iii)* NetHide can modify the *forwarding trees* for all nodes in V.

We show the constraints that encode the security and the objective function that captures the closeness in terms of accuracy and utility in Fig. 4 and explain them below.

Security constraints The main constraint is the *security* (C1) imposed on V. This being a hard constraint (as opposed to be part of the objective function) means that if

Objective function

$$\max_V \sum_{f\in F} \left(w_{acc}\cdot acc(f,P,V) + w_{util}\cdot util(f,P,V)\right)$$
where $w_{acc}\in[0,1]$, $w_{util}\in[0,1]$, $w_{acc}+w_{util}=1$

Hard Constraints

(Security)	$\forall l \in L' : fd(V,l) \le c(l)$	(C1)
(Complete)	$n\in N \Rightarrow n\in N'$	(C2)
(Reach)	$\forall n\in N' : \lvert\{T_n \mid T_n\in T'\}\rvert = 1$	(C3)
	$\forall T\in T' : \forall l\in T \;:\; l\in L'$	(C4)
	$(n,n')\in L' \Rightarrow \{n,n'\}\in N'$	(C5)

Figure 4: NetHide optimization problem. NetHide finds a virtual topology that is secure and has maximum accuracy compared with the physical topology.

NetHide finds a virtual topology V, then V is secure with respect to the attacker model and the capacities.

To ensure that the virtual topology V is valid, NetHide incorporates additional constraints capturing that: (C2) all physical nodes in N are also contained in the virtual topology with nodes N'; (C3) there is exactly one virtual forwarding tree for each node; and (C4-5) links and nodes in the virtual forwarding trees are contained in N'.

Objective function The objective of NetHide is to find a virtual topology that maximizes the overall accuracy (cf. §4.2) and utility (cf. §4.3). As shown in Fig. 4, we define the overall accuracy and utility as a weighted sum of the accuracy and utility values of all flows in the network.

4.2 Accuracy metric

The accuracy metric is a function that maps two paths for a given flow to a value $v\in[0,1]$. In our case, this value captures the similarity between a path $T_{s\to d}$ in P for a given flow (s,d) and the (virtual) path $T'_{s\to d}$ for the same flow (s,d) in V. Formally, given a flow (s,d), the accuracy is defined as:

$$acc((s,d),P,V) = 1 - \frac{LD(T_{s\to d}, T'_{s\to d})}{|T_{s\to d}| + |T'_{s\to d}|}$$

Where $LD(T_{s\to d}, T'_{s\to d})$ is Levenshtein distance [32] and $|T_{s\to d}|$ denotes the length of the path from s to d.

The overall accuracy of a topology (as referred to in §6) is defined as the average accuracy over all flows in F:

$$A_{avg}(P,V) = avg_{(s,d)\in F}\; acc((s,d),P,V)$$

We point out that the accuracy metric in NetHide can also be computed by any other function to precisely represent the network operator's needs.

```
Input: Flow $(s,d)\in F$,
       physical topology $P=(N,L,T)$,
       virtual topology $V=(N',L',T')$
Output: utility $u\in[0,1]$
for $n \in T'_{s\to d}$ do
    $C \leftarrow T_{s\to n}\cap T'_{s\to d}[0:n]$    // common links
    $u_n \leftarrow \frac{1}{2}\left(\frac{|C|}{|T_{s\to n}|} + \frac{|C|}{|T'_{s\to d}[0:n]|}\right)$    // utility
$u \leftarrow \frac{1}{|T'_{s\to d}|}\sum_{n\in T'_{s\to d}} u_n$    // average
```

Algorithm 1: Utility metric. It incorporates the likelihood that a failure in the physical topology P is visible in the virtual topology V and that a failure in V actually exists in P. Note that we treat $T_{s\to d}$ as a set of links.

4.3 Utility metric

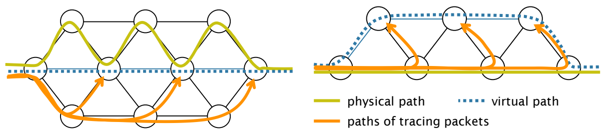
While the accuracy measures the similarity between the physical and virtual paths for a given flow, the utility measures the representation of physical events, such as link failures. For our implementation, we design the utility metric such that it computes the probability that a link failure in the physical path is observed in the virtual path and the probability that a failure reported in the virtual path is indeed occurring in the physical path.

Algorithm 1 describes the computation of our utility metric for a given flow (s,d). In the algorithm, given a virtual path $T'_{s\to d} = s\to n_1\to\cdots\to n_k\to d$, we write $T'_{s\to d}[0:n_i]$ to denote the prefix path $s\to n_1\to\cdots\to n_i$. NetHide computes the overall utility by taking the average utility computed over all flows:

$$U_{avg} = avg_{(s,d)\in F}\; util((s,d),P,V)$$

As with accuracy, a network operator is free to implement a custom utility metric.

In most cases, the accuracy and utility are strongly linked together (we show this in §6). However, as illustrated in Fig. 5, there exist cases where the accuracy is high and the utility low or vice-versa.



(a) high accuracy, low utility (b) low accuracy, high utility

Figure 5: High accuracy does not always imply high utility (and vice-versa). In Fig. 5a, the physical and virtual paths are similar but the tracing packets do not cross the physical links. In Fig. 5b, the physical and virtual paths are dissimilar but the tracing packets do cross the physical links.

4.4 Scalability

To obfuscate topologies with maximal accuracy and utility, a naive approach would consider all possible changes to P, which is infeasible even for small topologies.

NetHide significantly reduces the number of candidate solutions in order to ensure reasonable runtime while providing close-to-optimal accuracy and utility. The key insight is that NetHide pre-computes a set of *forwarding trees* for each node and later computes V as the optimal combination of them. Thanks to the reduction from modeling individual links or paths to forwarding trees, NetHide only considers *valid* combinations of paths (i.e., paths that form a tree rooted at n, $\forall n \in N'$).

For computing the forwarding trees, NetHide builds a complete graph G with all nodes from V, that is $G = (V, E)$ where $V = N'$ and $E = N' \times N'$, and assigns each edge the same weight $w(e) = 1\ \forall e \in E$. Then, NetHide uses Dijkstra's algorithm [21] to compute forwarding trees towards each node $n \in N'$. That is, a set of paths where the paths form a tree which is rooted at n. This is repeated until the specified number of forwarding trees per node is obtained while the weights are randomly chosen $w(e) \sim \mathit{Uniform}(1, 10)$ for each iteration.

As NetHide pre-computes a fixed number of forwarding trees per node, the ILP solver later only needs to find an optimal combination of $\mathscr{O}(|N'|)$ forwarding trees instead of $\mathscr{O}(|N'|^2)$ links and $\mathscr{O}(|N'|^{|N'|})$ forwarding trees.

We point out that the reduction from individual links or paths to forwarding trees and the small number of considered forwarding trees does not affect the security of V as security is a hard constraint and thus, NetHide *never* produces a topology that is insecure. In fact, the small number of considered forwarding trees actually makes NetHide more secure because it makes it harder to determine P even for a powerful brute-force attacker that can run NetHide with every possible input.

4.5 Security

We now discuss the security provided by NetHide. We consider two distinct attacker strategies: *(i)* reconstructing the physical topology P from the virtual topology V; and *(ii)* choosing an attack based on the observed virtual topology V (without explicitly reconstructing P). We describe the two strategies below.

Reconstructing the physical topology If the attacker can reconstruct P, then she can check if P is insecure and select a link and a set of flows that congests that link. Reconstructing the physical topology is mitigated in two ways. First, the attacker cannot reconstruct P with certainty by simply observing the virtual topology V. NetHide's obfuscation function maps any physical topology that is secure to itself (i.e., to P). The obfuscation function is therefore not injective, which entails that NetHide guarantees opacity [35], a well-known security property stipulating that the attacker does not know the secret P.

Given that the attacker cannot reconstruct P with certainty, she may attempt to make an educated guess based on the observed V and her knowledge about NetHide's obfuscation function. Concretely, the attacker may perform exact Bayesian inference to discover the most likely topology T that was given as input to the obfuscation function. Exact inference is, however, highly non-trivial as NetHide's obfuscation function relies on a complex set of constraints. As an alternative, the attacker may attempt to approximately discover a topology T that was likely provided as input to NetHide. Estimating the likelihood that a topology T could produce V is, however, expensive because NetHide's obfuscation is highly randomized. That is, the estimation step would require a large number of samples, obtained by running T using the obfuscation function.

Choosing an attack In principle, even if the attacker cannot reconstruct P, she may still attempt to attack the network by selecting a set of flows and checking if these cause congestion or not. As a base case for this strategy, the attacker may randomly pick a set of flows. A more advanced attacker would leverage her knowledge about the observed topology to select the set of flows such that the likelihood of a successful attack is maximized.

In our evaluation, we consider three concrete strategies: *(i) random*, where the attacker selects the set of flows uniformly at random, *(ii) bottleneck+random*, where the attacker selects a link with the highest flow density and selects additional flows uniformly at random from the remaining set of flows, and *(iii) bottleneck+closeness*, where the attacker selects a link with the highest flow density and selects additional flows based on their distance to the link. Our results show that NetHide can mitigate these attacks even for powerful attackers (which can run many flows) and weak physical topologies (with small link capacities) while still providing high accuracy and utility (cf. §6.7). For example, NetHide provides 90 % accuracy and 72 % utility while limiting the probability of success to 1 % for an attacker which can run twice the required number of flows and follows the *bottleneck+random* strategy in a physical topology where 20 % of the links are insecure.

Finally, we remark that while our results indicate that NetHide successfully mitigates advanced attackers, providing a formal probabilistic guarantee on the success of the attacker is an interesting and challenging open problem. As part of our future work, we plan to formalize a class of attackers, which would allow us to formulate and prove a formal guarantee on that class.

5 NetHide topology deployment

In this section, we describe how NetHide deploys the virtual topology V on top of the physical topology P. For this, we first state the challenges NetHide needs to address. Then, we provide insights on the programming language and the architecture using which we implemented NetHide and describe the packet processing software as well as the controller in detail. In addition, we explain the design choices that make NetHide partially deployable and we discuss the impact of changes in the physical topology to the virtual topology.

5.1 Challenges

In the following, we explain the major challenges which need to be addressed by the design and the implementation of the NetHide topology deployment in order to provide high security, accuracy, utility and performance.

Reflecting physical events in virtual topology Maintaining the usefulness of network tracing and debugging tools is a major requirement for any network obfuscation scheme to be practical. As we explained in the previous sections, NetHide ensures that tracing V returns meaningful information by maximizing the utility metric. As a consequence, NetHide must assure that the data plane is acting in a way that corresponds to the utility metric.

The key idea to ensure high utility in NetHide is that the tracing packets are sent through the physical network as opposed to being answered at the edge or by a central controller. Answering to tracing packets from a single point is impractical as events in P (such as link failures) would not be visible.

Timing-based fingerprinting of devices Besides the IP address of each node in a path, tracing tools allow to determine the round trip time (RTT) between the source and each node in the path. This can potentially be used to identify obfuscated parts of a path.

While packets forwarding is usually done in hardware without noticeable delay, answering to an expired (TTL=0) IP packet involves the router control plane and causes a noticeable delay. Actually, our experiments show that the time it takes for a router to answer to an expired packet not only varies greatly, but is also *characteristic* for the device, making it possible to identify a device based on the distribution of its processing time.

NetHide makes RTT measurements realistic by ensuring that a packet that is supposedly answered by node n is effectively answered by n. As such, n will process the packet as any other packet with an expired TTL irrespective of whether or not obfuscation is in place and the measured RTT is the RTT between the source host and n.

Packet manipulations at line rate To avoid tampering with network performance, NetHide needs to parse and modify network packets at line-rate. In particular, it needs manipulate the TTL field in IP headers as well as the IP source and destination addresses. Since changing these fields leads to a changed checksum in the IP header, NetHide also needs to re-compute checksums.

While there are many architectures and devices where the NetHide runtime can operate, we decided to implement it in P4, which we introduce in the next section.

5.2 NetHide and P4

P4 [20] is a domain-specific programming language that allows programming the data plane of a network. It is designed to be both protocol- and target-independent meaning that it can process existing protocols (e.g., IP or UDP) as well as developer-defined protocols. P4 programs can be compiled to various targets (e.g., routers or switches) and executed in different hardware (e.g., CPUs, FPGAs or ASICs). Software targets (e.g., [13]) provide an environment to develop and test P4 programs while hardware targets (such as [3]) can run P4 programs at line rate.

A P4 program is composed of a parser, which parses a packet and extracts header data according to specified protocols, a set of match+action tables and a control program that specifies how these tables are applied to a packet before the (potentially modified) packet is sent to the output port. Besides table lookups, P4 also supports a limited set of operations such as simple arithmetic operations or computing hash functions and checksums.

For our implementation, we use P4_14 [14] and leverage P4's customizable header format to rewrite tracing packets at line rate without requiring to keep state (per packet, flow or host) at the devices.

5.3 Architecture

NetHide features a controller to translate V to configurations for programmable network devices, and a packet processing software that is running on network devices and modifies packets according to these configurations.

The device configuration is described as a set of match+action table entries that are queried upon arrival of a packet (Fig. 6). The entries are installed when V is deployed the first time and when it changes. At other times, NetHide devices act autonomously.

We describe the packet processing software as well as the controller in the following two sections.

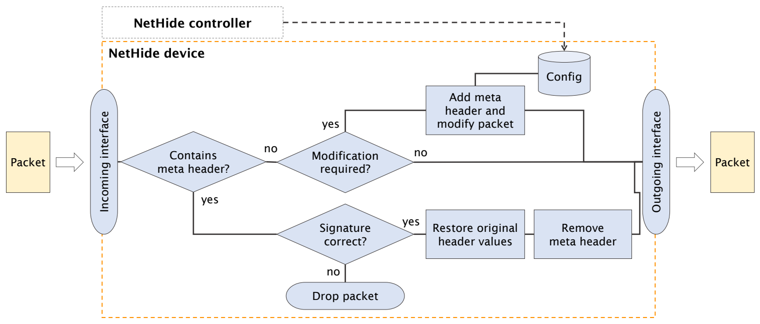


Figure 6: NetHide topology deployment architecture overview. A controller generates the configuration entries which are later used by the packet processing software running in NetHide devices.

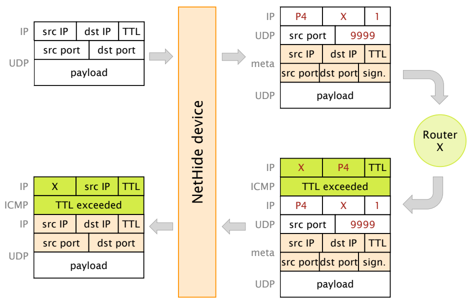


Figure 7: NetHide devices encode state information into packets in order to avoid maintaining state in the devices.

5.4 Packet processing software

The packet processing software is running in the data plane of a network device and typically performs tasks such as routing table lookups and forwarding packets to an outgoing interface. For NetHide, we extend it with functionality to modify packets such that the behavior for a network user is consistent with V. In the following paragraphs, we explain the processing shown in Fig. 6.

Identifying potential tracing packets Upon receiving a new packet, a NetHide device first checks whether it is a response to a packet that was modified by NetHide (cf. below). If not, it checks whether the packet's virtual path is different from the physical path and it thus needs to be modified. Even though we often use `traceroute` packets as examples, NetHide does not need to distinguish between `traceroute` (or other tracing traffic) and productive network traffic. Instead, it purely relies on the TTL value, the source and destination of a packet and—if needed—it obfuscates traffic of all applications.

Encoding the virtual topology If a packet needs to be modified, NetHide queries the match+action table which returns the required changes for the packet. Changes can include modifications of the destination address and/or the TTL value. If the packet's TTL is high enough that it can cross the egress router, NetHide does not need to modify addresses. However, if the virtual path for this packet has a different length than the physical path, the TTL needs to be incremented or decremented by the difference of the virtual and the physical path length.

If the packet has a low TTL value which will expire before the packet reaches its destination, NetHide needs to ensure that the packet expires at the correct node with respect to V. For this, NetHide modifies the destination address of the packet such that it is sent to the node that has to answer according to V. In addition, it sets the source address to the address of the NetHide device that handles the packet. Therefore, the modified packet is sent to the responding router and the answer comes back to the NetHide device. At this point, NetHide needs to restore the original source and destination addresses of the packet and forward the reply to the sender.

Rewriting tracing packets at line rate The devices that we use to deploy NetHide are able to modify network traffic at line rate without impacting latency and throughput. As described above, NetHide sometimes needs to modify the TTL value in production traffic (which does not impact latency or delay and is already done by routers today) and it needs to send tracing packets to different routers (which has an impact on the observed RTT; but only for tracing packets whose TTL expires before reaching the destination).

Rewriting tracing packets statelessly A naive way to be able to reconstruct the original source and destination addresses of a packet is to cache them in the device (which bears similarities with the operating mode of a NAT device—but the state would need to be maintained on a *per-packet* basis). Since this would quickly exceed the limited memory that is typically available in programmable network devices, NetHide follows a better strategy: instead of maintaining the state information in the device, it encodes it into the packets. More precisely, NetHide adds an additional header to the packet which contains the original (layer 2 and 3) source and destination addresses, the original TTL value as well as a signature (a hash value containing the additional header combined with a device-specific secret value) (cf. Fig. 7). This *meta header* is placed on top of the layer 3 payload and is thus contained in ICMP time exceeded replies.

Preventing packet injections Coming back to the first check when a packet arrives: if it contains a *meta header* and the signature is valid (i.e., corresponds to the device), NetHide restores the original source and destination addresses of the packet and removes the meta header before sending it to the outgoing interface.

5.5 NetHide controller

Below, we explain the key concepts of the NetHide controller which generates the configurations mentioned above.

Configuring the topology Being based on P4 devices, configuration entries are represented as entries in match+action tables which are queried by the packet processing program. NetHide's configuration entries are of the following form:

$$(\text{destination}, \text{TTL}) \mapsto$$
$$(\text{virtual destination IP}, \text{hops to virtual destination})$$

where the virtual destination IP can be unspecified if only the length of a path needs to be modified. P4 tables can match on IP addresses with *prefixes*, meaning that only one entry per prefix (e.g., `1.2.3.0/24`) is required. For example, the entry "`(1.2.3.0/24,1)` $\mapsto$ `(11.22.33.44,5)`" means that if the device sees a packet to `1.2.3.4` (or any other IP address in `1.2.3.0/24`) with `TTL=1`, it will send it to `11.22.33.44` and change the TTL-value to `5`.

Modifying packets distributedly NetHide selects one programmable device per flow which then handles all of the flow's packets. This device must be located before the first spoofed node, i.e., the first node in the virtual path that is different from the physical path.

While there is always one distinct device in charge of handling a certain flow, the same device is assigned to many different flows. To balance the load across devices, NetHide chooses one of the eligible devices at random (this does not impact the obfuscation). For more redundancy, multiple devices could be assigned to each flow.

Changing the topology on-the-fly Thanks to the separation between the packet processing software and the configuration table entries, V can be changed *on-the-fly* without interrupting the network.

5.6 Partial deployment

As deploying a system that needs to run on *all* devices is difficult, we design NetHide such that it can fully protect a network while being deployed on only a few devices. The key enabler for this is that NetHide only needs to modify packets at most at one point for each flow.

NetHide can obfuscate all traffic as soon as it has crossed at least one NetHide device. In the best case, in which NetHide is deployed at the network edge, it can protect the entire network. In the evaluation (§6), we show that even for the average case in which the NetHide devices are placed at random positions, a few devices are enough to protect a large share of the flows.

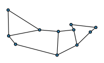
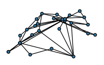
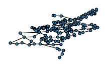
	Abilene	Switch	US Carrier
			
Nodes	11	42	158
Links	14	63	189
Max. flow density	35	390	11301
Avg. flow density	19	89	1587

Table 1: We evaluate NetHide based on three realistic topologies of different size.

5.7 Dealing with topology changes

NetHide sends tracing packets through P such that they expire at the correct node according to V. Changes in P can impact NetHide in two ways:

1. When links are *added* to P or the routing behavior changes: some flows may no longer traverse the device that was selected to obfuscate them. This can be addressed by installing configuration entries in multiple devices (which results in a trade-off between resource requirements and redundancy). Since V is secure in any case, there is no immediate need to react to changes in P. However, to provide maximum accuracy and utility, NetHide can compute a new V' based on P' and deploy it without interrupting the network.
2. When links are *removed* from P: this results in link failures in V and has no impact on the security of V. If the links are permanently removed, NetHide can compute and deploy a new virtual topology.

6 Evaluation

In this section, we show that NetHide: *(i)* obfuscates topologies while maintaining high accuracy and utility (§6.2, §6.3); *(ii)* computes obfuscated topologies in less than one hour, even when considering large networks (§6.4). Recall that this computation is done offline, once, and does not impact network performance at runtime; *(iii)* is resilient against timing attacks (§6.5); *(iv)* is effective even when partially deployed (§6.6); *(v)* mitigates realistic attacks (§6.7); and *(vi)* has little impact on debugging tools (§6.8).

6.1 Metrics and methodology

Metrics To be able to compare the results of our evaluation with different topologies, we use the average *flow density reduction factor*, which denotes the ratio between the flow density in the physical topology $P = (N, L, T)$ and in the virtual topology $V = (N', L', T')$:

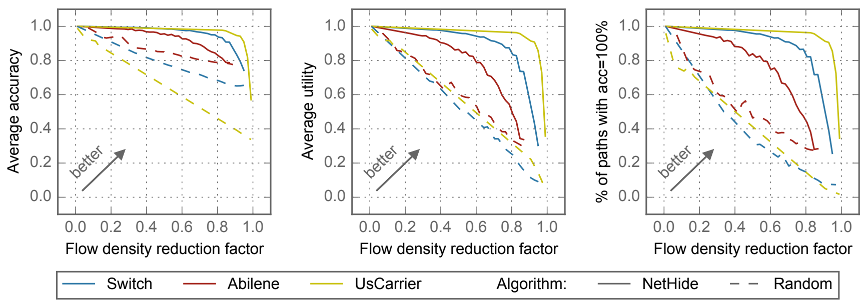


Figure 8: Accuracy and utility for different protection margins. NetHide achieves high accuracy (left plot) and utility (middle) and does not change most of the paths at all (right plot) while reducing the flow density by more than 75 %.

$$FR = 1 - \frac{\text{avg}_{l \in L'} fd(V,l)}{\text{avg}_{l \in L} fd(P,l)}$$

The flow density denotes the number of flows that are carried at each link (cf. §2.3). For example, $FR = 0.2$ means that the links in V carry 80% less flows than those in P (on average). For the accuracy and utility of V, we use A_{avg} and U_{avg} as defined in §4.

Datasets We consider three publicly available network topologies from [11]: a small (Abilene, the former US research network), a medium (Switch, the network connecting Swiss universities) and a large one (US Carrier, a commercial network in the US). Table 1 lists key metrics for the three topologies. For the forwarding behavior, we assume that traffic in P is routed along the shortest path or a randomly picked shortest path in case there are multiple shortest paths between two nodes.

Parameters We run all our experiments with the following parameters: All nodes in P can act as ingress and egress for malicious traffic (which is the worst case when an attacker is everywhere). We also assume that all links have the same capacity. Since tracing packets need to be answered by the correct node, NetHide only adds virtual links but no nodes (i.e., $N = N'$). We consider 100 forwarding trees per node. For the ILP solver, we specify a maximum relative gap of 2 %, which means that the optimal results can be at most 2 % better than the reported results (in terms of accuracy and utility, security is not affected). We run NetHide at least 5 times with each configuration and plot the average results.

6.2 Protection vs. accuracy and utility

In this experiment, we analyze the impact of the obfuscation on the accuracy and utility of V. For this, we run NetHide for link capacities c (the maximum flow density) varying between 10 % and 100 % of the maximum flow density listed in Table 1.

Fig. 8 depicts the accuracy (left) and utility (center) achieved by NetHide according to the flow density reduction factor. An ideal result is represented by a point in the upper right corner translating to a topology that is both highly obfuscated and provides high accuracy and utility. As baseline, we include the results of a naive obfuscation algorithm that computes V by adding links at random positions and routing traffic along a shortest path.

NetHide scores close to the optimal point especially for large topologies. We observe that the random algorithm can achieve high accuracy and utility (when adding few links) or high protection (when adding many links) but not both at the same time. Though, in a small area (very high flow density reduction in a small topology), the random algorithm can outperform NetHide. The reason is that such a low flow density is only achievable in an (almost) complete graph. While adding enough links randomly will eventually result in a complete graph, the small number of forwarding trees considered by NetHide does not always contain enough links to build a complete graph.

In Fig. 8 (right), we show the percentage of flows that do not need to be modified (i.e., have 100 % accuracy and utility) depending on the flow density reduction factor.

Fig. 8 (right) illustrates that NetHide can obfuscate a network without modifying most of its paths therefore preserving the usability of tracing tools. In the medium size topology, NetHide computes a virtual topology that lowers the average flow density by more than 80 % while keeping more than 80 % of the paths *identical*. This is significantly better than the random baseline where a flow density reduction by 80 % only preserves about 15 % of the paths. We observe that larger topologies generally exhibit better results than small ones. This is due to the fact that in bigger topologies, a small modification has less impact on average accuracy than in a small topology while still providing high obfuscation. Conversely, smaller topologies lead to worse results as a small number of changes can have a big impact.

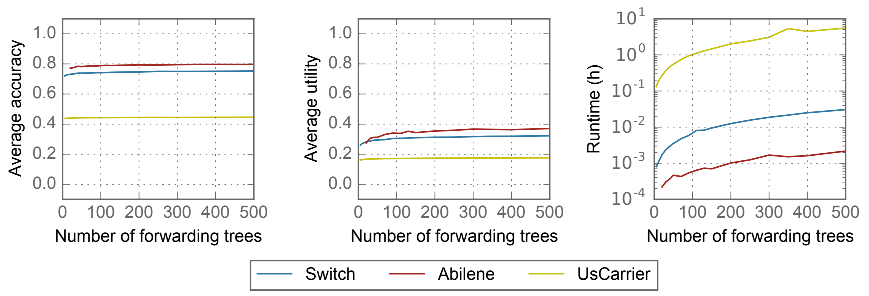


Figure 9: Accuracy, utility and runtime for different number of forwarding trees. Considering only a small number of forwarding trees per node does not significantly decrease the accuracy and utility of NetHide but drastically decreases the runtime. Thanks to this, NetHide can obfuscate large topologies (>150 nodes) in less than one hour.

6.3 Accuracy vs. utility

In Fig. 10, we analyze the impact of the accuracy weight (w_{acc} in Fig. 4) on the resulting accuracy and utility. We specify the capacity of each link to 10 % of the maximum flow density listed in Table 1 and observe that w_{acc} has a relatively small impact for our accuracy and utility metrics especially for large topologies. This confirms that a topology with a high accuracy typically also has a high utility. If the paths are similar (high accuracy), the packets are routed via the same links (high utility), too.

6.4 Search space reduction and runtime

In this experiment, we analyze the impact of the search space reduction—in terms of the number of forwarding trees per node—on the runtime of NetHide. As we explained in §4.4, NetHide considers only a small subset of forwarding trees to improve scalability. We again specify the capacity of each link to 10 % of the maximum flow density listed in Table 1 and run NetHide for a varying number of forwarding trees per node. The experiments were run in a VirtualBox VM running Ubuntu 16.04 with 20 Intel Xeon E5 CPU cores and 90 GB of memory.

In Fig. 9, we show that a small number of forwarding trees is enough to reach close-to-optimal results. While the runtime increases exponentially with the number of forwarding trees, the accuracy and utility do not noticeably improve above 100 forwarding trees per node.

The runtime of NetHide when considering 100 forwarding trees per node is within one hour, even for large topologies (Fig. 9). As the topology is computed offline (cf. §5.7), such a running time is reasonable.

6.5 Path length

In this experiment, we analyze the difference between the lengths of paths in P and V. Large differences between the length of the physical path and the virtual path can lead to unrealistic RTTs and leak information about the obfuscation (e.g., if the RTT is significantly different for two paths of the same length).

As the results in Fig. 11 show, virtual paths are shorter than physical paths (the ratio is ≤ 1)—intuitively because removing a node from a path has a smaller impact on our accuracy and utility metrics than adding one) and—for the medium and large topology—the virtual paths are less than 10 % shorter both on average and in the 10th percentile for a flow density reduction of 80 %.

The resulting small differences in path lengths support our assumption that timing information mainly leaks through the processing time at the last node and not through the propagation time (§5) as long as all links have roughly the same propagation delay.

6.6 Partial deployment

We now analyze the achievable protection if not all devices at the network edge are programmable. In NetHide, a flow can be obfuscated as long as it crosses a NetHide device before the first spoofed node (the first node that is different from the physical path). This is obviously the case if all edge routers are equipped with NetHide. Yet, as we show in Fig. 12, a small percentage of NetHide devices (e.g., 40 %) is enough to protect the majority (60 %) of flows even in the average case where the devices are placed at random locations and all nodes are considered as ingress and egress points of traffic (i.e., as edge nodes).

To obtain the results in Fig. 12, we set the maximum flow density to 10 % of the maximum value in Table 1 and vary the percentage of programmable nodes in V between 0 and 100 %. For each step, we compute the average amount of flows that can be protected for 100 different samples of programmable devices.

The percentage of obfuscated flows in Fig. 12 is normalized to only consider flows that need to be obfuscated. As we have shown in Fig. 8, the vast majority of flows does not need to be obfuscated at all.

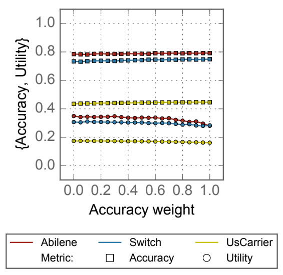


Figure 10: The accuracy weight has a small impact for our accuracy and utility metrics.

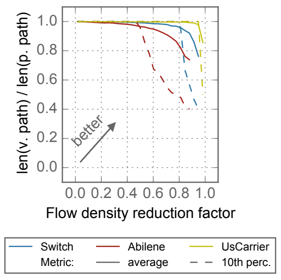


Figure 11: Reducing the flow density by 80 % changes path lengths by less than 20 %.

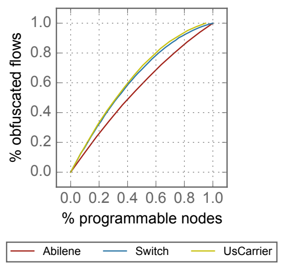


Figure 12: Partial deployment at random locations. 40 % NetHide devices allow to protect up to 60 % of the flows *that need obfuscation*

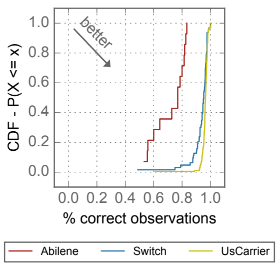


Figure 13: Link failures are correctly observed with high probability (e.g., for Switch: only 15 % of the failures appear in less than 90 % of the paths.)

As an alternative approach to partial deployment, NetHide can be extended to incorporate the number and/or locations of NetHide devices as a constraint or as an objective such as to compute virtual topologies that can be deployed without new devices or with as few programmable devices as possible.

6.7 Security

As we explained in §4.5, inferring the exact physical input topology from the virtual topology is difficult.

However, an attacker can try to attack V directly, without trying to determine P. Such an attacker is limited by the fact that she does not know P and by a maximum number (budget) of flows that she can create. Therefore, the key challenge for the attacker is to select the flows such that they result in a successful attack on P.

Besides the attacker's budget, her chances of success also depend on the robustness of P: If P is weak (i.e., the capacity of many links is exceeded), it either needs to be obfuscated more or attacks are more likely to succeed.

In this experiment, we simulate three feasible strategies for an attacker to select b flows:

- *Random*: Samples b flows uniformly at random from the set of all flows F.
- *Bottleneck+Random*: Identifies the link with the highest flow density in V (a "bottleneck" link l_b) and attacks by initiating all the $fd(l_b)$ flows that cross this link plus $(b - fd(l_b))$ random additional flows.
- *Bottleneck+Closeness*: Identifies the link l_b with the highest flow density in V and attacks by initiating all the $fd(l_b)$ flows that cross this link plus $(b - fd(l_b))$ nearby flows (according to the metric in Algorithm 2).

An attack is successful if running the selected set of flows in P exceeds any link's capacity (not necessarily the link that the attacker tried to attack).

In our simulations, we vary both the attacker's budget and the robustness of P (in terms of the link capacity). We vary the capacity such that between 10 % and 100 % of the links in P are secure (e.g., if 10 % of the links are secure, an attacker could directly attack 90 % of the links if there was no obfuscation). For each choice of the link capacity c in P, we vary the number of flows that the attacker can initiate between $b = c + 1$ (just enough to break a link) and $b = 4 \times (c + 1)$ (four times the number of flows that the most efficient attacker would need).

To obtain the simulation results in Fig. 14 and Fig. 15, we simulated 10k attempts (*Random* and *Bottleneck+Random*) and 1k attempts (*Bottleneck+Closeness*) for each virtual topology from §6.2 and each combination of the link capacity and attacker budget.

In Fig. 14 we compare the *Random* attacker with *Bottleneck+Random* and in Fig. 15 we compare *Random* with *Bottleneck+Closeness*. In the first row of each figure, we plot how much obfuscation (i.e., in terms of the flow density reduction factor) is required to make the attacker successful in < 1 % of her attempts. There, we observe that the *Random* attacker is (as expected) the least powerful because it requires less obfuscation to defend against it and that *Bottleneck+Closeness* is slightly more powerful than *Bottleneck+Random*. Considering the setting with the Abilene topology and the attacker with 2× budget: Mitigating this attacker requires no obfuscation when she follows the *Random* strategy, but 71 % (*Bottleneck+Random*) or 86 % (*Bottleneck+Closeness*) flow density reduction for the more sophisticated strategies.

The required flow density reduction naturally increases as the attacker's budget increases. In the right column where the attacker can run four times the number of required flows, even the *Random* attacker is successful because she can run so many flows (or even all possible flows in many cases) that it does not matter how the flows are selected.

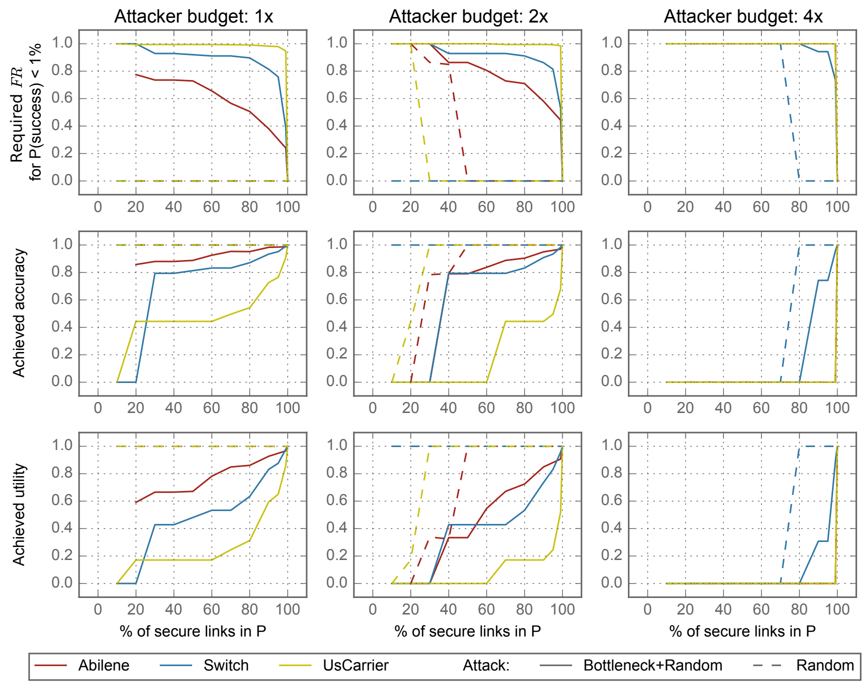


Figure 14: Attack simulations comparing the *Random* attacker with *Bottleneck+Random*. The plots show the required flow density reduction (FR) for making the attacker succeed with $Pr < 1\%$ (first row) and the obtained accuracy and utility (second and third row) depending on the link capacity of the physical topology (measured as the percentage of secure links in the x-axis). For example, defending the Switch topology with only 60 % secure links against *Bottleneck+Random* with 2× budget maintains 80 % accuracy.

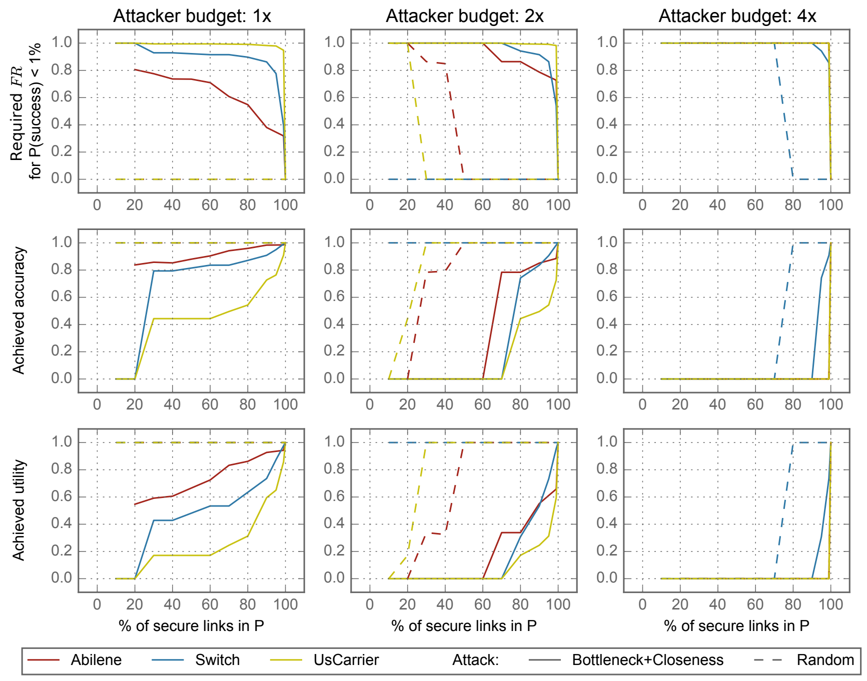


Figure 15: Attack simulations comparing the *Random* attacker with *Bottleneck+Closeness*. *Bottleneck+Closeness* is slightly more powerful than *Bottleneck+Random* (Fig. 14), which results in more obfuscation that is required.

```
Input:  Virtual topology $V = (N', L', T')$,
        Flow $(s,d) \in N' \times N'$,
        Flow path $T'_{s \to d}$
        Bottleneck link $(n_1, n_2) \in L'$
Output: Preference $p \in [0,1]$

if $(n_1 \in T'_{s \to d}) \wedge (n_2 \in T'_{s \to d})$ then
    $p \leftarrow 1/|$ links between $n_1$ and $n_2$ in $T'_{s \to d}|$
else if $(n_1 \in T'_{s \to d}) \wedge (n_2 \notin T'_{s \to d})$ then
    $n_a \leftarrow$ node after $n_1$ in $T'_{s \to d}$
    $n_b \leftarrow$ node before $n_1$ in $T'_{s \to d}$
    $p_a \leftarrow$ length of path from $n_2$ to $n_a$
    $p_b \leftarrow$ length of path from $n_2$ to $n_b$
    $p \leftarrow 1/\min(p_a, p_b)$
else if $(n_1 \notin T'_{s \to d}) \wedge (n_2 \in T'_{s \to d})$ then
    (see above with $n_1$ and $n_2$ flipped)
else
    $p \leftarrow 0$
```

Algorithm 2: Flow preference metric. Flows that contain the bottleneck link or at least one of the endpoints of the link are more promising to be useful in the attack.

The second and third row in the plots show the accuracy and utility that is preserved after obfuscating the topology. We observe there, that especially the Abilene and Switch topologies provide high accuracy and utility even if less than 50 % of the links in P are secure. Comparing Fig. 14 and Fig. 15 shows that since mitigating *Bottleneck+Closeness* requires more obfuscation, the achieved accuracy and utility is lower.

6.8 Case study: Link failure detection

We now show that NetHide preserves most of the usefulness of tracing tools by considering the problem of identifying link failures in obfuscated topologies. For our analysis, we use all three topologies and a flow density reduction factor of 50 %. Then, we simulate the impact of an individual failure for each link. That is, we analyze how a failing *physical* link is represented in V.

Failing a link can have different effects in V: Ideally, it is *correctly observed*, which means that the exact same link failure appears in V. But since V contains links that are not in P or vice-versa, a physical link failure can be observed as *multiple link failures* or as the *failing of another virtual link.*

In Fig. 13, we show that the vast majority of physical link failures is precisely reflected in the virtual topology. That is, NetHide allows users to use prevalent debugging tools to debug connectivity problems in the network. These results are a major advantage compared to competing approaches [28, 39] that do not send the tracing packets through the actual network.

7 Frequently asked questions

Below, we provide answers to some frequently asked questions and potential extensions of NetHide.

Can a topology be de-obfuscated by analyzing timing information? In NetHide, each probing packet is answered by the correct router and thus the processing time at the last node is realistic. Though, the propagation time can leak information in topologies where the propagation delay of some links is significantly higher than of others.

However, extracting information from the propagation time in geographically small networks is hard for three reasons: *(i)* it is impossible to measure propagation time separately. Instead, only the RTT is measurable; *(ii)* the RTT includes the unknown return path; and *(iii)* NetHide keeps path length differences are small. For topologies exhibiting larger delays, NetHide can be extended to consider link delays as an additional constraints.

The same arguments hold for analyzing queuing times or other time measurements. Moreover, delays often vary greatly in short time intervals, making it practically infeasible to perform enough simultaneous measurements.

Can a topology be de-obfuscated by analyzing link failures? Because some physical link failures are observed as multiple concurrent link failures in the virtual topology, an attacker can try to reconstruct the physical topology by observing link failures over a long timespan. However, this strategy is not promising for the following reasons: *(i)* most of the link failures are directly represented in the virtual topology (cf. §6.8). Observing them does not provide usable information for de-obfuscation; and *(ii)* analyzing link failures over time requires permanent tracing of the entire network between, which would make the attacker visible and is against the idea of LFAs.

Is NetHide compatible with link access control or VLANs? Not at the moment, but we can easily extend our model to support them. The required changes are: *(i)* link access control policies need to be part of the NetHide's input; *(ii)* the ILP needs additional constraints to respect different VLANs (i.e., model forwarding trees per VLAN); *(iii)* the output consists of VLAN-specific paths; and *(iv)*, the runtime additionally matches on the VLAN ID and applies the appropriate actions.

Does NetHide support load-balancing? Not at the moment, but after the following extensions: *(i)* instead of an exact path for each flow, we specify the *expected* load that a flow adds to each link (e.g., using max-min fair allocation as in [30]); *(ii)* the constraints regarding the flow density now constrain the *expected* flow density; *(iii)* the virtual topology can contain multiple parallel paths and probabilities with which each path is taken; and *(iv)* the runtime randomly selects one of the possible paths.

How close to the optimal is the solution computed by NetHide? Computing this distance is computationally infeasible as it requires to exhaustively enumerate all possible solutions (one of the cruxes behind NetHide security). Instead, we measure the distance between the virtual and the physical topology (§6.2) and show that the virtual topology is already very close (in terms of accuracy and utility) to the physical one. The optimal solution would therefore only do slightly better, while being much harder to compute.

Can NetHide be used with other metrics for computing the flow density? At present, NetHide requires a static metric such that the flow density can be computed before obfuscating the topology. For simplicity, we assume that the load which each flow imposes to the network is the same and all links have the same capacity. However, this assumption can easily be relaxed to allow specific loads and capacities for each flow and link (therefore requiring more knowledge or assumptions about the topology and the expected traffic).

8 Related work

Existing works on detecting and preventing LFAs can be broadly classified into reactive and proactive approaches. Reactive approaches only become active once a potential LFA is detected. As such, they do not prevent LFAs and only aim to limit their impact after the fact. CoDef [31] works on top of routing protocols and requires routers to collaborate to re-route traffic upon congestion. SPIFFY [25] temporarily increases the bandwidth for certain flows at a congested link. Assuming that benign hosts react differently than malicious ones, SPIFFY can tell them apart. Liaskos et al. describe a system [33] that continuously re-routes traffic such that it becomes unlikely that a benign host is persistently communicating via a congested link. Malicious hosts on the other hand are expected to adapt their behavior. Nyx [36] addresses the problem of LFAs in the context of multiple autonomous systems (ASes). It allows an AS to route traffic from and to another AS along a path that is not affected by an LFA.

On the other hand, proactive solutions—including NetHide—aim at preventing LFAs from happening and are typically based on obfuscation. HoneyNet [28] uses software-defined networks to create a virtual network topology to which it redirects `traceroute` packets. While this hides the topology from an attacker, it also makes `traceroute` unusable for benign purposes. Trassare et al. implemented topology obfuscation as a kernel module running on border routers [39]. The key idea is to identify the most critical node in the network and to find the ideal position to add an additional link that minimizes the centrality of this node. The border router replies to `traceroute` packets as if there was a link at the determined position. However, adding a single link has little impact on the security of a big network and even if the procedure would be repeated, an attacker could determine the virtual links with high probability. Further, `traceroute` becomes unusable for benign users as the replies come from the border router.

Linkbait [40] identifies potential target links of LFAs and tries to hide them from attackers. Hiding a target link is done by changing the routing of tracing packets from bots in such a way that the target link does not appear in the paths. As a prerequisite to only redirect traffic from bots, Linkbait describes a machine learning-based detection scheme that runs at a central controller which needs to analyze all traffic. Being based on re-routing of packets, Linkbait can only present paths that exist in the network. Therefore, a topology that does not have enough redundant paths cannot be protected. The paper does not discuss issues with an attacker that is aware of the protection scheme and sends tracing traffic that is likely to be misclassified and therefore not re-routed.

Other approaches that are related to LFAs but not particularly to our work are based on virtual networks [22], require changes in protocols or support from routers and end-hosts [19,29] or focus on the detection of LFAs [41].

9 Conclusion

We presented a new, usable approach for obfuscating network topologies. The core idea is to phrase the obfuscation task as a multi-objective optimization problem where security requirements are encoded as hard constraints and usability ones as soft constraints using the notions of accuracy and utility.

As a proof-of-concept, we built a system, called NetHide, which relies on an ILP solver and effective heuristics to compute compliant obfuscated topologies and on programmable network devices to capture and modify tracing traffic at line rate. Our evaluation on realistic topologies and simulated attacks shows that NetHide can obfuscate large topologies with marginal impact on usability, including in partial deployments.

Acknowledgements

We are grateful to the anonymous reviewers, Benjamin Bichsel, Rüdiger Birkner and Tobias Bühler for the constructive feedback and the insightful discussions. This work was partly supported by armasuisse Science and Technology (S+T) under the Zurich Information Security and Privacy Center (ZISC) grant.

References

[1] 3 in 4 DDoS attacks aimed at multiple vectors. https://www.enterpriseinnovation.net/article/3-4-ddos-attacks-aimed-multiple-vectors-512931178.

[2] Akamai q2 2017 state of the Internet. https://content.akamai.com/us-en-pg9565-q2-17-state-of-the-internet-security-report.html.

[3] Barefoot Tofino. https://barefootnetworks.com/products/product-brief-tofino/.

[4] Can a DDoS break the Internet? Sure... just not all of it. https://arstechnica.com/information-technology/2013/04/can-a-ddos-break-the-internet-sure-just-not-all-of-it/.

[5] DDoS attack threat cannot be ignored. http://www.computerweekly.com/feature/DDoS-attack-threat-cannot-be-ignored.

[6] Dyn Statement on 10/21/2016 DDoS Attack. https://dyn.com/blog/dyn-statement-on-10212016-ddos-attack/.

[7] Exclusive: Inside the ProtonMail siege: how two small companies fought off one of Europe's largest DDoS attacks. http://www.techrepublic.com/article/exclusive-inside-the-protonmail-siege-how-two-small-companies-fought-off-one-of-europes-largest-ddos/.

[8] Github survived the biggest DDoS attack ever recorded. https://www.wired.com/story/github-ddos-memcached/.

[9] Gurobi mathematical programming solver. http://www.gurobi.com/products/gurobi-optimizer.

[10] How to fight the new breed of DDoS attacks on data centers. http://www.datacenterknowledge.com/security/how-fight-new-breed-ddos-attacks-data-centers.

[11] The internet topology zoo. http://topology-zoo.org/.

[12] Message regarding the ProtonMail DDoS attacks. https://protonmail.com/blog/protonmail-ddos-attacks/.

[13] P4 behavioral model. https://github.com/p4lang/behavioral-model.

[14] The P4 language specification - version 1.0.4. https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf.

[15] RFC 2992 - analysis of an equal-cost multi-path algorithm. https://tools.ietf.org/html/rfc2992.

[16] RIPE atlas. https://atlas.ripe.net/.

[17] traceroute(8) - Linux manual page. http://man7.org/linux/man-pages/man8/traceroute.8.html.

[18] Unmetered mitigation: DDoS protection without limits. https://blog.cloudflare.com/unmetered-mitigation/.

[19] Basescu, C., Reischuk, R. M., Szalachowski, P., Perrig, A., Zhang, Y., Hsiao, H.-C., Kubota, A., and Urakawa, J. SIBRA - Scalable internet bandwidth reservation architecture. *arXiv preprint arXiv:1510.02696* (2015).

[20] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM CCR 44*, 3 (2014).

[21] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik 1* (1959).

[22] Gillani, F., Al-Shaer, E., Lo, S., Duan, Q., Ammar, M., and Zegura, E. Agile virtualized infrastructure to proactively defend against cyber attacks. *IEEE INFOCOM 2015*.

[23] Giotsas, V., Smaragdakis, G., Dietzel, C., Richter, P., Feldmann, A., and Berger, A. Inferring BGP blackholing activity in the Internet. *ACM IMC 2017*.

[24] Holterbach, T., Pelsser, C., Bush, R., and Vanbever, L. Quantifying interference between measurements on the RIPE atlas platform. *ACM IMC 2015*.

[25] Kang, M. S., Gligor, V. D., and Sekar, V. SPIFFY: Inducing cost-detectability tradeoffs for persistent link-flooding attacks. In *NDSS 2015*.

[26] Kang, M. S., Lee, S. B., and Gligor, V. D. The crossfire attack. *IEEE S&P 2013*.

[27] Katz-Bassett, E., John, J. P., Krishnamurthy, A., Wetherall, D., Anderson, T. E., and Chawathe, Y. Towards IP geolocation using delay and topology measurements. *ACM IMC 2006*.

[28] Kim, J., and Shin, S. Software-defined HoneyNet: Towards mitigating link flooding attacks. *IEEE/IFIP DSN-W 2017*.

[29] Kim, T. H.-J., Basescu, C., Jia, L., Lee, S. B., Hu, Y.-C., and Perrig, A. Lightweight source authentication and path validation. *ACM SIGCOMM 2014*.

[30] Kumar, P., Yuan, Y., Yu, C., Foster, N., Kleinberg, R., Lapukhov, P., Lim, C. L., and Soulé, R. Semi-oblivious traffic engineering: The road not taken. *USENIX NSDI 2018*.

[31] Lee, S. B., Kang, M. S., and Gligor, V. D. CoDef: Collaborative defense against large-scale link-flooding attacks. *ACM CoNEXT 2013*.

[32] Levenshtein, V. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady 10* (1966).

[33] Liaskos, C., Kotronis, V., and Dimitropoulos, X. A novel framework for modeling and mitigating distributed link flooding attacks. *IEEE INFOCOM 2016*.

[34] Petitcolas, F. A. P. *Kerckhoffs' Principle*. Springer US, 2011.

[35] Schoepe, D., and Sabelfeld, A. Understanding and enforcing opacity. *IEEE CSF 2015*.

[36] Smith, J. M., and Schuchard, M. Routing around congestion: Defeating DDoS attacks and adverse network conditions via reactive BGP routing. *IEEE S&P 2018*.

[37] Spring, N., Mahajan, R., and Wetherall, D. Measuring ISP topologies with Rocketfuel. *ACM SIGCOMM CCR 32*, 4 (2002).

[38] Studer, A., and Perrig, A. The coremelt attack. In *ESORICS 2009*, vol. 5789, Springer.

[39] Trassare, S. T., Beverly, R., and Alderson, D. A technique for network topology deception. In *IEEE MILCOM 2013*.

[40] Wang, Q., Xiao, F., Zhou, M., Wang, Z., Li, Q., and Li, Z. Linkbait: Active link obfuscation to thwart link-flooding attacks. *arXiv preprint arXiv:1703.09521*.

[41] Xue, L., Luo, X., Chan, E. W., and Zhan, X. Towards detecting target link flooding attack. *USENIX LISA 2014*.