Denial-of-Service Attacks in Bloom-Filter-Based Forwarding

Markku Antikainen, Tuomas Aura, and Mikko Särelä

***Abstract*—Bloom-filter-based forwarding has been suggested to solve several fundamental problems in the current Internet, such as routing-table growth, multicast scalability issues, and denial-of-service (DoS) attacks by botnets. The proposed protocols are source-routed and include the delivery tree encoded as a Bloom filter in each packet. The network nodes forward packets based on this in-packet information without consulting routing tables and without storing per-flow state. We show that these protocols have critical vulnerabilities and make several false security assumptions. In particular, we present DoS attacks against broad classes of Bloom-filter-based protocols and conclude that the protocols are not ready for deployment on open networks. The results also help us understand the limitations and design options for Bloom-filter forwarding.**

***Index Terms*—Multicast, network protocols, network-level security and protection.**

I. Introduction

IN-PACKET Bloom-filter-based forwarding has been proposed as a solution for several problems in the current Internet, including routing-table growth and scalable multicast [1], denial-of-service (DoS) resistant forwarding [2], and information-centric network design [3], [4]. Its use has also been proposed for data centers [5], [6] and as an enhancement for Multi-Protocol Label Switching (MPLS) [7]. There are several implementations, including one for *NetFPGA* [3], [6], [8], [9].

The idea behind in-packet Bloom-filter-based forwarding is relatively simple: The delivery tree is stored in the packet header as a set of *forwarding-hop identifiers* (FHIDs), which can be either nodes, links, or in–out interface pairs on the delivery tree. The set of the FHIDs in the delivery tree is encoded as a Bloom filter [10] data structure, which enables efficient testing of set membership. Network nodes forward packets by checking which potential FHIDs, e.g., outgoing links, are in the Bloom filter.

Manuscript received May 17, 2012; revised March 10, 2013; accepted August 06, 2013; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor I. Keslassy.

M. Antikainen and T. Aura are with the School of Science, Aalto University, Espoo 00076, Finland (e-mail: markku.antikainen@aalto.fi; tuomas.aura@aalto.fi).

M. Särelä is with the School of Electrical Engineering, Aalto University, Espoo 00076, Finland (e-mail: mikko.sarela@aalto.fi).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TNET.2013.2281614

In comparison to standard IP routing-table lookup or solutions that use Bloom filters to encode routing tables [11] or destination addresses [12], [13], in-packet Bloom-filter-based forwarding requires little computation at the routers. It can also scale to much larger numbers of multicast groups than the current IP multicast protocols because the network nodes do not need to store any per-group or per-flow state. These properties make it attractive to use the same protocol for both unicast and multicast.

Existing Bloom-filter-based protocols have proposed three security mechanisms: 1) limiting the number of items stored in the Bloom filters; 2) centralizing Bloom filter computation and making forwarding-hop identifiers secret; 3) using cryptographically computed per-flow forwarding-hop identifiers.

However, perhaps surprisingly nobody until today has thoroughly evaluated the security of these proposals. In this paper, we start this work by analyzing the denial-of-service resistance of the existing Bloom-filter-based forwarding architectures. Many variations of such protocols have been proposed in the literature (see Section II). Since no single protocol has yet been standardized, the analysis in this paper aims to cover all the different variants of encoding the delivery tree in to the in-packet Bloom filter. To do this, we have created a unified connectivity-graph model that makes it possible to analyze different protocol variants with a single model.

We evaluate the effectiveness of the security mechanisms against denial-of-service attacks using three different attacks. First, we show that static link-identifiers can be reverse-engineered. This was hypothesized in [2]. However, our paper is the first to show this is actually the case. Second, we show that distributed packet-flooding attacks can get around the proposed security mechanisms in existing literature. Third, some protocol variants are vulnerable to attacks that prevent nodes from leaving a multicast group. These attacks show that most of the abstract security claims presented in the literature do not hold under detailed analysis.

The attacks are distributed. That is, they require the attacker to have access to a botnet consisting of at least hundreds of compromised computers. This is a reasonable assumption because real DoS attacks on the current Internet are commonly launched from botnets. It is also estimated that a large portion (16%–25%) of Internet hosts belong to botnets [14]. Moreover, the abstract security claims made about the Bloom-filter-based protocols cover distributed denial-of-service attacks.

Our work shows that Bloom-filter-based forwarding needs further improvements on security before deployment on open networks. All the existing protocol variants have weaknesses that allow an attacker with a botnet to send traffic to a target

node and, sometimes, to prevent the victim from unsubscribing traffic flows. However, it should be noted that the security mechanisms proposed in the literature do increase the cost of DoS attacks and, thus, they may be useful in combination with further security solutions.

The rest of the paper is structured as follows. Section II gives an overview of Bloom-filter forwarding and previous work on DoS resistance. Section III presents a unified connectivity-graph model for analyzing different protocol variants. This model is then used to show that an attacker can reverse-engineer the link identifiers, which should be secret. Section IV presents injection attacks in which a single legitimate route to a target node is exploited by an entire botnet. Section V explains how a malicious subscriber can prevent others from unsubscribing a data flow. The implications of the attacks on the security and design of Bloom-filter-based protocols are discussed in Section VI. Finally, Section VII concludes the paper.

II. Bloom-Filter Forwarding

This section gives an overview of the Bloom-filter-based forwarding proposals. Section II-A provides background information on Bloom filters, and Section II-B surveys the proposed protocols. (For readers not previously familiar with the concepts, a brief tutorial is provided in Appendix A.) Section II-C discusses the previous work on denial-of-service attacks and countermeasures in this family of protocols.

A. Bloom Filters

Bloom filter [10] is a probabilistic space-efficient data structure for storing sets. Its basic operations are *membership testing* and *element addition*, but not element removal. Bloom filter is implemented as a bit array with fixed length m and a small number k of hash functions that map data items to indexes $[0, m-1]$ in the bit array. A data item is added to the set by computing the k hash functions on it and setting the corresponding k bits in the bit array to 1. Membership is tested similarly by computing the k hash functions on the data item and checking whether the corresponding k bits in the array are all 1. The operations take constant time but can, nevertheless, be computationally relatively expensive if cryptographic hash functions are used.

Bloom filters are probabilistic in the sense that the membership testing may return *false positive* results. The probability of false positives grows with the number of elements n added to the set because the percentage of 1-bits in the bit array grows. For this reason, it is necessary to limit the number of elements that are stored in the filter. For example, if we use $k = 7$ hash functions, a filter of length m can store $n = m/10$ data items with false positive rate of approximately 0.8%. (For details about Bloom filters and the false positive rate, we refer the reader, e.g., to Mullin [15].)

Because of their space efficiency, Bloom filters are typically used for storing large sets or when the memory usage needs to be minimized. In this paper, the filters have to fit into network packet headers. Therefore, we consider unusually short filters of 256–1024 bits, which can store roughly 20–100 data items.

B. Bloom-Filter-Based Multicast Forwarding

Three methods have been proposed for Bloom-filter-based multicast forwarding: 1) using Bloom filters in the multicast routers to reduce the space required for storing the multicast forwarding table [11]; 2) source routing where the multicast delivery-tree is encoded as a Bloom filter in the packet headers (e.g., [1] and [3]); and 3) storing the list of receivers in the packet header as a Bloom filter [12], [13].

1) Encoding Multicast-Group Addresses Into Bloom Filter: One of the first papers to suggest Bloom filters for multicast forwarding is by Grönvall [11]. In this work, each outgoing interface of a multicast router has a Bloom filter that encodes the multicast-group addresses reachable through that interface. False positives are acceptable because they only cause the multicast packets to be forwarded to some unnecessary paths.

2) Bloom-Filter-Based Source Routing: Several proposed future-internet protocols encode the source tree or route in an in-packet Bloom filter. In these proposals, effectively, each link is given an m-bit *link identifier* in which k pseudorandomly chosen bits are set. The function and inputs used to choose the bits vary in different proposals.

The Bloom filter F for the multicast tree is the bitwise OR of all link identifiers in the tree. The membership of a link identifier l is tested by checking the equality $F \wedge l = l$, where $\wedge$ is the bitwise AND of the bit arrays. Thus, the forwarding nodes only need to compute simple bitwise logical operations to process each packet. (See Appendix A for an example.)

The Free Riding Multicast (FRM) [1] was the first to propose implementing interdomain multicast using this method. It uses BGP to propagate membership information. The border router at the source computes the spanning tree for the multicast group and encodes the tree into the packet header. The link identifiers are encoded by hashing the pair of autonomous system numbers (ASNs). There are a few basic problems with FRM: It assumes that all BGP routers have an up-to-date information of the current state of interdomain routing and, as the ASNs are public information, it provides no security against malicious users. Accidental forwarding loops may also be created.

The Line Speed Publish/Subscribe Inter-networking (LIPSIN) protocol [3] makes use of Bloom filters to implement multicast with *sender access control* for a publish–subscribe internetworking architecture that decouples the sender and receiver from each other [16]. The link identifiers are secret, and the path filters are computed by a *topology manager*, an external third party responsible also for the access control. Implementing such a logically centralized topology manager is possible in the intradomain case because it is akin to the path computation element (PCE) in MPLS [17]. However, creating a centralized topology manager that can be deployed in the distributed interdomain environment is much harder because it would require network operators to relinquish control over routing policy. This is currently an unresolved problem.

LANES [18] is another Bloom-filter forwarding protocol for a future publish–subscribe Internet architecture. Unlike FRM or LIPSIN, it uses in-packet Bloom filters only for unicast. Multicast is implemented by stateful routers at branching points.

Bloomcast [4], [19], [20] is an interdomain multicast protocol that uses IP unicast for multicast signaling. The subscriber nodes send *join* messages to the publisher over IP. The Bloom filter for the path is collected into a hop-by-hop header in the join packet. That is, the join packet starts with an empty filter from the subscriber and, at each hop on the way to the publisher, a link identifier is added to the filter with bitwise OR. This *distributed filter discovery* is an alternative to the centralized topology manager used in LIPSIN.

In addition to the already mentioned protocols, Rotherberg *et al.* [6] propose using in-packet Bloom filters in switching for data centers, while Zahemszky *et al.* [7] present Multi-Protocol Stateless Switching (MPSS), which combines Bloom-filter forwarding and MPLS. These protocols are intended for use within a single administrative domain and are thus not particularly sensitive to security vulnerabilities. Nevertheless, the broad range of Bloom-filter protocols proposed in the literature means that it is important to understand their security properties and limitations.

3) Encoding Receiver Set Into Bloom Filter: Application-oriented multicast (AOM and DOM) [12], [13] encodes the recipient address prefixes into a Bloom filter in the packet. Each router compares the Bloom filter against a list of Bloom filters in its routing table, which is updated whenever a router sees a multicast join message. When the router discovers a match showing that one or more receivers are reachable through a given interface, it reconstructs the Bloom filter with only those receivers, hence deleting any excess 1-bits from the filter.

This approach has some advantages and some drawbacks for security. One advantage is that it is more difficult for the attacker to add bits to the Bloom filter because the filter is recomputed after every hop. Also, since the routers hold state for the recipient domains, it becomes more difficult to forge a Bloom filter that causes packets to go to a chosen target.

A major drawback is that each router needs to compare the packet to all the destination Bloom filters in its routing table. BGP routing tables may contain hundreds of thousands of entries, and hence the process is computationally expensive, leaving routers potentially vulnerable to resource exhaustion attacks. The attacker only needs to make sure that the router contains many destination entries and receives many multicast packets. Both are easily achieved if the attacker controls a botnet.

Our analysis is focused on the source-routed protocols where routers do not hold large routing tables and do not need to perform expensive computation.

C. Known DoS Problems and Solutions

Multicast enables the sender to reach a large number of receivers even though it only sends each packet once. The use of Bloom filters creates a probabilistic element in packet forwarding; packets may be forwarded over links they were not intended as well as over the intended links. (However, false negatives, i.e., packets not forwarded over intended links, are not possible.) These two—traffic amplification and potential for false positives—create potential security problems. First, multicast protocols are prone to *injection attacks* in which the attacker tries to send unauthorized packets into the multicast delivery tree. Second, false positives may cause packets to loop or hop from one branch of the multicast tree to another, causing traffic to be amplified in the whole downstream multicast tree.

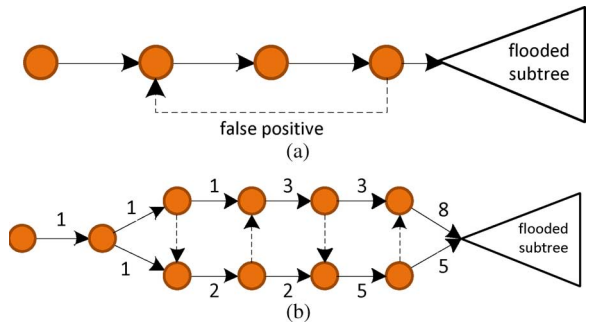


Fig. 1. Forwarding anomalies [19]. (a) Forwarding loop. (b) Repeated flow duplication.

1) Injection Attacks and Z-Formation: Rothenberg *et al.* [2] identify the potential for injection attacks in Bloom-filter based forwarding and suggest, without giving many details, that an attacker could analyze correlations between filters and thus derive new filters that enable the injection attack. To our knowledge, we are the first to show that such attacks are, indeed, possible.

Their proposed solution, called *Z-formation*, is to compute the link identifiers dynamically on per-packet basis. The Bloom-filter hash functions are implemented as a cryptographic hash that takes its arguments from a *flow identifier* in the packet, a periodically changing secret key local to each forwarding node, and the incoming and outgoing interface identifiers on the intended path of the packet. This way, the Bloom filter stores more routing context than just the links on the route. The purpose is to increase the difficulty of reverse-engineering the secret link identifiers or combining the filters of different flows (which otherwise could be done by computing their bitwise OR). Moreover, the time dependence of the identifiers means that the path filter will need to be refreshed periodically, at which point misbehaving senders can be denied the capability to send to the path.

2) Forwarding Anomalies and Bit Permutations: DoS attack may also result from *forwarding anomalies* caused by the false positives. The obvious case is a packet with an *all-ones filter*: The packet would be forwarded to everywhere and loop around any cycles in the network. For this reason, the forwarding nodes drop packets in which the fill factor ρ, i.e., the percentage of 1-bits, exceeds a threshold $\rho_{\max}$. The maximum fill factor is usually 50%.

Särelä *et al.* [4], [19] point out that false positives can also cause unintentional *forwarding loops* if a packet is accidentally forwarded back to an earlier node in the multicast tree, and false positives between two branches of the multicast tree can cause *flow duplication*. Fig. 1 illustrates these situations. The latter can happen even in valley-free routing, which prevents loops. While Särelä *et al.* focus on accidental routing anomalies, we are concerned about malicious senders causing similar problems intentionally.

To solve these problems, Särelä *et al.* suggest *permuting the Bloom filter* at each router. The permutation is simply a shuffle of the filter bits. The bit permutations are described in more

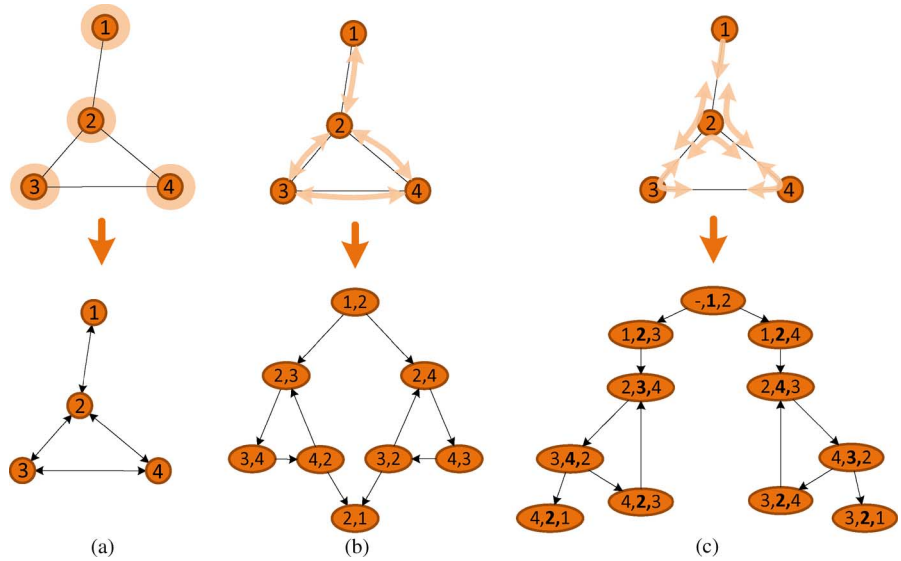


Fig. 2. Different FHID types and connectivity graphs for the same network. (a) FHID = node. (b) FHID = link, i.e., node pair. (c) FHID = in–out interface pair.

detail in Appendix A. Each node has a fixed, pseudorandomly chosen permutation, which is applied to the filters in all packets that travel through the node. This is sufficient to protect against loops and flow duplication because the Bloom filter becomes a function of the path that the packet has already traversed.

An alternative to the permutations would be to include a hop counter or the already traversed path of the packet as an input to the Bloom filter hash functions, i.e., the z-Formation. The permutations have almost the same effect, but they are faster to compute than the z-Formation and they have practical compositionality properties.

Analogous to the link identifiers, permutations can be kept secret and they can depend on the incoming or outgoing interface or their combination. It is, however, not practical to change the permutations frequently. First, the problem of creating fast and efficient key-dependent pseudo-random bit permutations either with software [21] or hardware [22], [23] is a well-known difficult problem in the field of cryptography. Also, static bit permutations can be implemented in hardware simply by crossing wires, which makes them attractive for the designers of forwarding hardware. Second, changing the bit permutations is not easy to do gradually. If a forwarding node uses both the new and old permutation simultaneously, it will have to create two copies of every packet, one processed with the old and one with the new bit permutation. There is no similar problem with the updating of FHIDs.

III. Reverse-Engineering Attack

We will now start with the security analysis of Bloom-filter forwarding. In Section III-A, we explain how different variants of the link identifiers can be modeled in a uniform way. The model will be used in all the following sections including Section III-B, in which we show that an attacker with a botnet is able to reverse-engineer the supposedly secret link identifiers.

A. Connectivity Graph

As seen in Section II-C, the data items stored in the Bloom filter do not need to be just link identifiers. They could equally identify the *nodes* on the path, the *links* on the path, or the *incoming–outgoing interface pairs* on the path. Moreover, the identifiers could be *symmetric*, i.e., the same regardless of the direction in which the packets travel, or they could be *directional*, i.e., different in the upstream and downstream directions. All these variants provide sufficient information for the forwarding nodes to pass the packet onto the next-hop node. To avoid confusion, we will use the term *FHID* to denote all these different types of identifiers. This paper, however, focuses on directional identifiers because they give stronger sender access control. (Symmetric identifiers would be suitable for groups in which all members are allowed to send.)

In order to analyze networks with different identifier types, we define the concept *connectivity graph*. The vertices in this directed graph represent forwarding-hop identifiers, and there is an edge from one identifier to another if a packet can traverse the second hop directly after the first one. Fig. 2 illustrates the connectivity graphs for the same network topology with three different identifier types. Using the forwarding-hop identifiers in the figure, the path from 1 to 4 via 2 and 3 would be encoded as the set of: (a) nodes $\{2, 3, 4\}$; (b) links $\{\langle 1, 2\rangle, \langle 2, 3\rangle, \langle 3, 4\rangle\}$; and (c) in–out interface pairs $\{\langle -, 1, 2\rangle, \langle 1, 2, 3\rangle, \langle 2, 3, 4\rangle\}$.

Depending on the FHIDs, the resulting connectivity graphs can differ greatly in size. Table I shows the graph sizes for two quite different network topologies. The AS graph is a power-law network in which the core nodes can have thousands of neighbors. In the tree topology, the node degrees are almost uniform. As we will see below, the size of the connectivity graph can be an indicator for the difficulty of many attacks presented in this paper: The more FHIDs there are, the harder it is for an attacker to reverse-engineer them or to combine them into useful

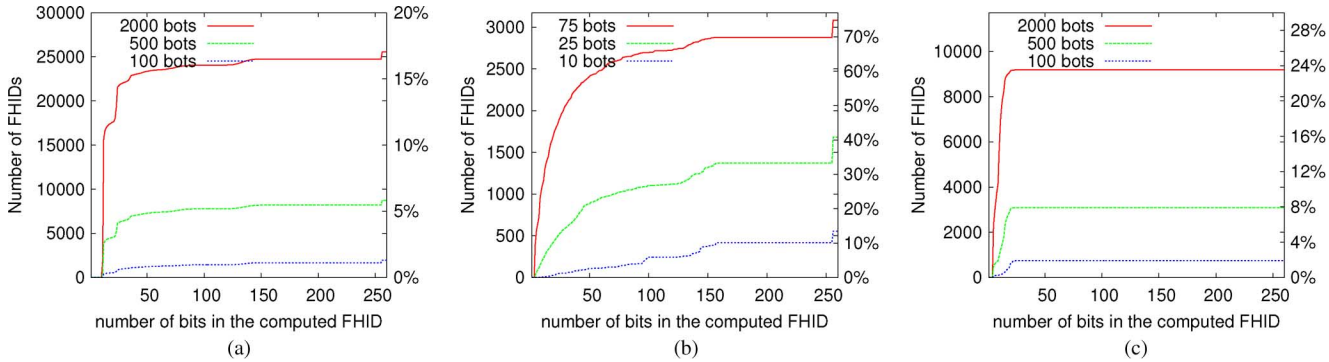


Fig. 3. CDF of accuracy for reverse-engineered link identifiers (for the topologies, see Appendix B). (a) AS graph. (b) 32×32 grid graph. (c) 5-ary tree of depth 7.

TABLE I
CONNECTIVITY GRAPH SIZES IN TWO DIFFERENT NETWORK TOPOLOGIES

Topology	FHID type	Connectivity graph	
		Vertices	Edges
AS-topology, valley-free routing (33 508 nodes)	Nodes	33 508	150 002
	Directional links	150 002	44 662 198
	In-out interface pairs	44 662 198	1 041 147 480
5-ary tree of depth 7 (19 531 nodes)	Nodes	19 531	39 060
	Directional links	39 060	117 170
	In-out interface pairs	117 170	195 200

paths. Nevertheless, we will see that even a very large connectivity graph does not make a network entirely immune to DoS attacks.

B. Reverse-Engineering FHIDs

The secrecy of forwarding-hop identifiers is critical to the security of Bloom-filter forwarding. For this reason, we first investigate the difficulty of reverse-engineering the identifiers. If the attacker knows all the FHIDs, it can flood specific targets with packets. Moreover, even partial knowledge of the secret identifiers may enable the attacker to create routing loops or cause flow duplication intentionally.

We consider an attacker that controls a botnet. The bots are randomly distributed around the network, and they can subscribe to flows from each other. The attacker also knows the network topology and routing policy. This analysis applies directly, for example, to LIPSIN [3], but not to protocols where the FHIDs are flow-specific or where the filters are permuted.

In order to compute the approximate forwarding-hop identifiers, the attacker first obtains the path filters between all pairs of bots. (How it obtains them depends on how filter discovery is implemented in the particular system. That is, however, irrelevant to the attack.) Since the attacker knows the network topology and routing policy, it knows which connectivity graph vertices each path passes through. For each vertex, it computes the intersection (i.e., bitwise AND) of the filters of all the paths that pass through the vertex. This gives the vertex FHID with reasonable accuracy if even two paths cross there. Most of the computed FHIDs will have some extra bits set, but that does not prevent their use in filter creation.

The attack was simulated with three very different network topologies: the AS topology, planar grid graph, and a 5-ary tree of depth 7. In the simulation, the length of the filters was $m = 1024$, and the number of hash functions k was set to 12, 4, or 5 depending on the node degrees in the topology. (See Appendix B for details of the example topologies.) Bots were placed randomly in the network. After this, the path filters were calculated for the shortest paths between all pairs of bots. To make the attacks more difficult, all filters have been filled to the maximum fill factor $\rho = 50\%$ by setting random bits. Finally, approximations of all of the FHIDs were computed as the bitwise AND of the path filters that are known to include the FHID.

Fig. 3 shows the accuracy of the computed link identifiers for the three network topologies and different botnet sizes. The faster the cumulative distribution function (CDF) curve rises, the larger percentage of the computed link identifiers are known with high accuracy. In all cases, an attacker with a medium-to-large botnet could compute a significant portion (between 5% and 50%) of the link identifiers with reasonable accuracy. The resolved identifiers were typically at the core of the network where most paths cross.

Note that the attacker can use partially solved FHIDs just as well as fully computed ones. Path Bloom filters are created by computing the bitwise OR of the FHIDs, and the extra 1-bits in a Bloom filter simply result in some more false positives, which is not a problem for a DoS attacker. The unnecessary 1-bits matter only if the fill factor of the created path filter exceeds the 50% maximum, but the attacker can avoid using such long routes.

While the graphs are only presented for link identifiers, results for the other FHID types were similar. The node identifiers are the easiest to reverse-engineer, and the in–out interface-pair identifiers are more difficult due to the vast number of them. Some identifiers in the core network can nevertheless be computed with high accuracy.

The attacker is thus able to reverse-engineer most FHIDs at the core of the network, but fewer of those close to the network edge. This means that the attacker is able to create routing loops or amplifying routes in the core network. On the other hand, the reverse-engineering attack does not seem to enable the flooding of arbitrary edge nodes. Periodic updating of the identifiers can make the attack somewhat more difficult because the attacker will have to repeat the reverse-engineering regularly. We can nevertheless conclude that it is wrong in principle to assume that the identifiers can be kept secret.

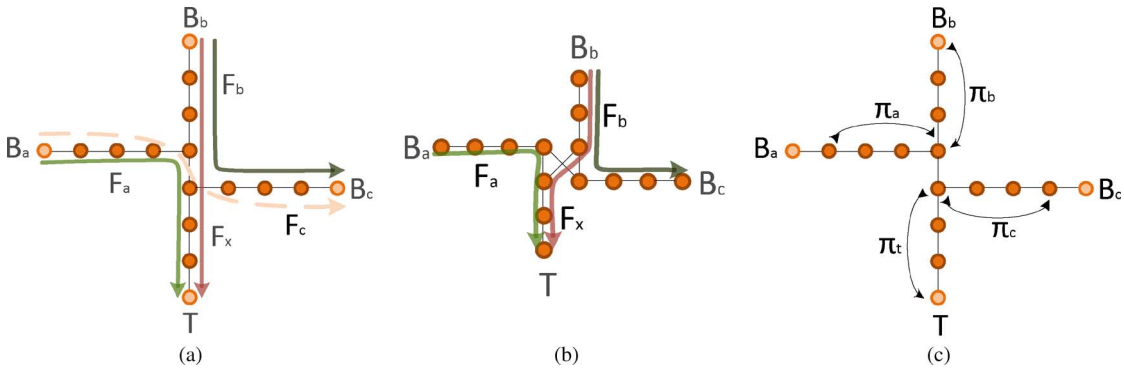


Fig. 4. Injection attack: F_a and F_b are combined into F_x for sending packets from B_b to the target. (a) Network topology. (b) Connectivity graph (FHID = in–out interface pair). (c) Path permutations.

As mentioned in Section II-C, there may be another secret parameter in the forwarding nodes: the filter permutation. These affect the reverse-engineering attack in two ways. First, they obscure the filter in such a way that the above process for computing FHIDs will not work in general. Second, there does not seem to be any straightforward way for a sparsely distributed botnet to reverse-engineer the individual permutations. For this reason, we turn next to another class of attacks that can deal with the permutations of the filters as well.

IV. Injection Attack

This section presents a form of injection attack against Bloom-filter-based multicast. We show that if the attacker has control over just one node that can send to a target, then a significant portion of the botnet will be able to send packets to the target. Section IV-A explains the basic principle of the attack, and the following sections show that it is possible under increasingly stringent security assumptions.

A. Basic Injection

Fig. 4(a) shows a simple injection attack. The target T has subscribed to a data flow from a compromised node, the bot B_a. These packets are sent with the path filter F_a in the packet header. The goal of the attacker is to enable another bot B_b to also send to the target T. If this succeeds with B_b, the attacker can try to do the same for all nodes in the botnet.

The attacker needs help from another suitably positioned node B_c. This node subscribes to a flow from B_b. The packets from B_b to B_c are sent with the path filter F_b. As seen in the figure, the path represented by F_a intersects with the path represented by F_b. Since the attacker controls the bots, it learns both of these filters. The attacker then computes the bitwise OR (i.e., union) of the path filters

$$F_x = F_a \vee F_b. \tag{1}$$

This filter can be used for sending packets from B_b to the target T.

In order for the attack to work, two conditions must be met. First, the two paths need to meet in such a way that a complete path from B_b to T is formed by their union. How much overlap between the paths is required depends on the FHID type. We can state the general requirement in terms of the connectivity graph: If the connectivity-graph vertices in F_a and F_b form together a continuous path from B_b to T, then B_b can send packets to the target with F_x. Second, the fill factor of F_x must not exceed the maximum 50%. That will usually not happen when F_a and F_b are unicast paths. The natural question to ask next is how often these conditions are met in practice. We will, however, defer this question to the end of Section IV-B, where we evaluate a more advanced version of the attack.

It must be noted that the packets sent by B_b with the filter F_x are not only forwarded to T but also to B_c. There are techniques for reducing such collateral damage, but we will not go into the details because the attacker does not necessarily care.

If the filters are created by a trusted topology manager, one potential countermeasure is to fill all filters up to the 50% maximum (by setting random bits as necessary). This prevents the attacker from combining filters with simple bitwise OR. The attacker can, however, remove the extra bits from the filter F_b by testing which ones are unnecessary for the packet to travel to B_c. Any transport-layer acknowledgment or application-layer response mechanism will also enable the attacker to remove the extra bits from F_a by testing which bits can be zeroed while still prompting a response from T. Thus, setting all filters to the maximum fill factor cannot be relied on as a security measure.

Another, similar countermeasure would be to reduce the length of the Bloom filter so that the fill factor would be close to 50%. The length of the Bloom filter can be reduced by folding the filter and computing the bitwise OR of the overlapping parts: For example, halving a Bloom filter would happen by computing the bitwise OR of the first half of the filter with the second half. However, reducing the Bloom filter length does not prevent our attack because dynamic Bloom filter lengths would also make it possible for the attacker to expand the filters.

B. Injection With Permutations

The injection attack becomes more interesting if the forwarding nodes permute the Bloom filters. The secret permutations certainly make it impossible to combine the filters with (1). For example, when each router in Fig. 4 permutes the filter, B_b cannot use $F_a \vee F_b$ to send packets to T because the bits of F_a will be in a wrong order when they reach the point where the two paths meet. In order for them to be permuted into the

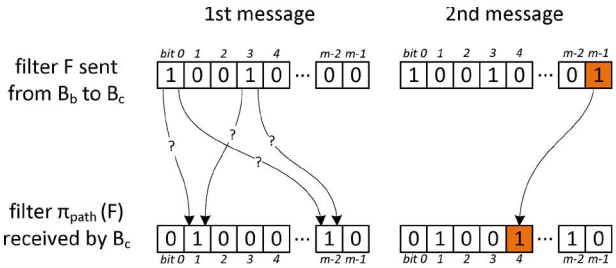


Fig. 5. Solving (partially) the composite permutation between two bots by turning 0-bits to 1-bits.

right locations, the filter would have to traverse the path from B_a to the meeting point.

However, while we cannot reverse-engineer the individual permutations, we can reason about the composite permutations over longer paths. Fig. 5 illustrates how the attacker discovers the composite permutation on the path between two bots. It sets one or more 0-bits in the filter to 1 at the sender and observes which bits change to 1 at the receiver. (Setting multiple bits at the same time can speed up the process as long as the fill factor of the filter remains under the maximum.) However, the attacker cannot learn how the 1-bits are mapped because zeroing any one of them would lead to the packet not arriving at the receiver. In order to discover the full permutation, the attacker needs to wait for the FHIDs in the network to be updated several times, so that the locations of the 1-bits change and all bit locations in the filter are zero at least once. Thus, the periodic update of the secret identifiers, which was intended as a security measure, actually helps this attack.

The average number of updates required to fully compute the composite permutation is logarithmic with the length of the filter. Experimentally, we found the number of updates to be between 5 and 20 for any reasonable filter. For a worst-case scenario for the attacker, consider a filter length $m = 1024$ and fill factor close to $\rho = 50\%$. In that case, the composite permutation between two bots is almost guaranteed to be known completely after 20 filter updates and at most 512 packets sent per update, i.e., about 10^4 packets in total. Usually, a fraction of this number of updates and packets is sufficient.

Let us now look at Fig. 4(c). We have given names to the composite permutations that the Bloom filter undergoes when the packet travels through specific path segments. With the procedure described above, the attacker can learn the composite permutations $\pi_c \circ \pi_a$ from B_a to B_c and $\pi_c \circ \pi_b$ from B_b to B_c as well as their inverses.

Remember that the goal of the attacker is to create a filter that can be used to send traffic from B_b to T. Such a filter F_x would need to satisfy the subset relations $F_b \subseteq F_x$ and $\pi_a(F_a) \subseteq \pi_b(F_x)$. (One filter is a subset of the other if all 1-bits in the first one are also set in the second one.) Satisfying the first relation makes a packet sent from B_b traverse to the meeting point of the paths. The second relation enables the packet to traverse from there to the target. Now, the attacker can compute the following filter from the composite permutations and filters that it already knows:

$$F_x = (\pi_c \circ \pi_b)^{-1}\left((\pi_c \circ \pi_a)(F_a)\right) \vee F_b. \tag{2}$$

It is clear that F_x satisfies the first subset relation. It also satisfies the second relation

$$\begin{aligned} & F_x \supseteq (\pi_c \circ \pi_b)^{-1}\left((\pi_c \circ \pi_a)(F_a)\right) && \text{(3a)} \\ \Rightarrow \quad & \pi_b(F_x) \supseteq \pi_b\left((\pi_c \circ \pi_b)^{-1}\left((\pi_c \circ \pi_a)(F_a)\right)\right) && \text{(3b)} \\ & = \left(\pi_b \circ \pi_b^{-1} \circ \pi_c^{-1} \circ \pi_c \circ \pi_a\right)(F_a) && \text{(3c)} \\ & = \pi_a(F_a). && \text{(3d)} \end{aligned}$$

Thus, *if* there is a bot B_c in a suitable place, it can help B_b to inject packets into the legitimate path from B_a to T. As promised earlier, we now return to the question of attack impact. That is, we want to know how many bots B_b in a typical botnet can make use of this technique to send a flood of packets to a given target.

We have simulated the attack for the three different network topologies (see Appendix B). Some bots and a target node were placed randomly into the network. One of the bots was chosen randomly as B_a, which is given a valid filter to the target. After this, we calculated the number of bots that were able to inject traffic into the target. The experiment was repeated 1000 times for each set of parameters. Fig. 6 shows the CDF of the percentage of successful bots for the different topologies and two different FHID types.

Our first observation is that the attack efficiency depends on the FHID type. The more information about the path the FHID encodes, the less often the paths meet in a suitable way. Clearly, FHIDs that identify in–out interface pairs are more resistant to the attack than link identifiers. Another observation is that the network topology has a great effect. In a tree topology, almost all long paths meet each other and, thus, it is usually possible to form a path from almost every bot to the target. In the AS network, on the other hand, there are many alternative paths through the core of the network and fewer paths meet. However, even in the best case of Fig. 6(d), there is a probability of a few percent that more than half of the bots in the botnet can send packets to the target.

We also noticed that the attack failures arise from poor placement of the special node B_a and from the lack of a good candidate for B_c. In the simulation, we chose the B_a randomly. However, if the attacker could choose to place B_a far away from the target and if it could place even one bot very close to the target to act as B_c, then most bots could always send to the target. Whether the attacker can influence the placement of the bots depends on factors outside this analysis, such as the application-level protocols or service-discovery mechanisms. A related observation is that the success rate depends heavily on the size of the botnet. This is because the attacker can use the bot with the best location as B_c. As the number of bots grows, there will more likely be a good candidate for B_c near the target.

In order to verify the correctness of the simulation results, we have included an example analytical model of the injection-attack efficiency in Appendix C. The model only covers the tree topology, which is the easiest to describe mathematically and has the simplest routing policy. The analytical results in Fig. 11 match closely with the simulation results in Fig. 6(f), which gives us confidence in the simulation. Similar modeling of the AS and grid networks including their routing policies would

be a major undertaking because we have chosen the simulation topologies to represent three quite different types of networks and routing algorithms and not for the ease of analytical modeling.

Finally, we consider injection attacks in a network where the FHIDs are not updated regularly or where the permutations are rerandomized at the same time with the identifiers. In that case, the attacker cannot learn the mapping of the 1-bits in the composite permutations by waiting for identifier updates. Nevertheless, the injection attack is still possible by setting all the unknown bits to one. Let us denote by F_c the filter for sending packets from B_a to B_c [see Fig. 4(a)]. The attacker already has learned F_c when measuring the permutation $\pi_c \circ \pi_a$. To send packets from B_b to T, it uses the following filter:

$$F_x = (\pi_c \circ \pi_b)^{-1} \left(\underbrace{(\pi_c \circ \pi_a)(F_a \vee F_c)}_{F'} \right) \vee F_b. \quad (4)$$

This filter clearly satisfies the two subset relations required for an injection. However, it is not obvious from the above formula how the attacker is able to calculate its value. The key insight is that the attacker can set all of the unknown bits to 1. This is done using (5b) and (6b), where the $\neg$ symbol denotes bitwise NOT operation (i.e., bitwise complement)

$$F' = (\pi_c \circ \pi_a)(F_a \vee F_c) \quad (5a)$$

$$= \underbrace{(\pi_c \circ \pi_a)(F_a \wedge \neg F_c)}_{\text{left}} \vee \underbrace{(\pi_c \circ \pi_a)(F_c)}_{\text{right}} \quad (5b)$$

$$F_x = (\pi_c \circ \pi_b)^{-1}(F') \vee F_b \quad (6a)$$

$$= \underbrace{(\pi_c \circ \pi_b)^{-1}\left(F' \wedge \neg(\pi_c \circ \pi_b)(F_b)\right)}_{\text{left}} \vee \underbrace{F_b}_{\text{right}}. \quad (6b)$$

The right sides of (5b) and (6b) are known to the attacker, and it can calculate the left sides because the bits whose mapping in the two composite permutations is unknown are zeroed out. The only drawback compared to (2) is that, because more filters are combined, there is a greater risk of exceeding the maximum fill factor 50%. In a network with a large radius, this may prevent the attacker from using pairs of bots that are very far apart from each other. In our simulations, such cases do not occur, and the results of Fig. 6 apply also to this more generic version of the attack.

C. Injection With Unknown Topology

We now take the injection attack one step further by removing the assumption that the attacker knows the network topology and routing policy. This makes the attack more complicated because the attacker does not initially know which paths between bots meet as in Fig. 4. The attacker can, however, deduce this information by looking at the Bloom filters.

The key idea is that we can estimate the number of FHIDs in a path by counting the 1-bits in the filter. We can also determine the approximate number of common FHIDs in two paths by computing their dot product (i.e., the number of 1-bits in the bitwise AND). Multiple observations of the dot product over time, after FHID updates, are required for high-confidence estimates of these values.

Without filter permutations, observations of the dot product alone could be used to decide which paths meet. With permutations, a slightly more complex process is required because only a common path segment starting from or ending to the observer can be detected. Thus, in Fig. 4(b), bot B_a can estimate the number of common hops in the beginning of the paths represented by F_a and F_c, and B_c can estimate the number of common hops in the end of the paths represented by F_b and F_c. If the sum of these values is equal to or greater than the estimated length of the path F_c, then the paths F_a and F_b meet.

More specifically, two filters that do not contain any common connectivity graph vertices should have a dot product close to $\rho_1\rho_2 m$, where ρ_i are the fill factors and m is the filter length. On the other hand, if the two filters have c common elements, the dot product of the two Bloom filters should be larger than this. Generally, the value x of the dot product of two Bloom filters, which share c common elements and thus have approximately $c \cdot k$ bits at corresponding positions, follows the hypergeometric probability distribution

$$P(X = x) = \binom{\rho_1 m - ck}{x - ck} \binom{m - \rho_1 m}{\rho_2 m - x} \binom{m - ck}{\rho_2 m - ck}^{-1}. \quad (7)$$

The distribution has the mean value $ck + (\rho_1 m - ck)(\rho_2 m - ck)(m - ck)^{-1}$. By calculating several dot products after periodic identifier updates and comparing the dot products to the mean values for different c, the attacker can determine the length of the shared paths with ever increasing confidence.

Thus, with a relatively simple statistical computation from the filters between bots, the attacker can determine which paths meet in the way that enables the injection attack. Even after just a few identifier updates, the attacker can make relatively good guesses. An attacker with a large botnet does not need to get all the values right for a successful packet-flooding attack, only most of them.

D. Injection With Flow-Specific FHIDs

Our analysis thus far has covered FHIDs that are static or periodically updating and the same for all data flows. We now turn to flow-specific identifiers. They are implemented by including some flow-specific information from the packet headers in the input to the Bloom-filter hash functions (e.g., the z-Formation [2]). The flow-specific information can be a sender or multicast-group identifier, or there can be a special identifier space for naming the flows. For unicast flows, both endpoint names could be used in a way similar to the TCP connection identifiers. The reasoning behind this defense mechanism is that the FHIDs and Bloom filters for different flows will be independent of each other. Consequently, the attacker cannot reverse-engineer FHIDs by computing the bitwise AND of path filters or concatenate paths with the bitwise OR. It might thus appear that flow-specific identifiers prevent the attacks discussed so far in this paper. This is unfortunately not the case.

The weakness in flow-specific identifiers is the way in which they are assigned. The protocol proposals that employ flow-specific identifiers [2], [4] brush aside the question of how the independence of the identifiers between flows is enforced. Let

us consider what happens if it is the sender's and/or the receiver's responsibility to choose or set the flow-specific values in the packet headers. In that case, the attacker could violate the rules about setting these values, so that the same "flow-specific" values are actually used in all of the attacker's flows. In the injection attack, the attacker could copy the flow-specific data from the one legitimate flow (from B_a to T) and use these values in all the packets sent between the bots. In the reverse-engineering attack of Section III, the attacker could similarly use the same "flow-specific" values in all the packets. The honest forwarding nodes in the network do not have routing tables, and they do not store any information about the data flows. Thus, they have no way of verifying the sender, multicast-group or flow identifiers in the packets.

To better understand how this weakness is exploited, we consider the basic injection attack (see Section IV-A), but against a network that uses flow-specific FHIDs. Let $F(fid)$ denote a Bloom filter that is constructed with a flow identifier fid. Initially, the attacker has a legitimate Bloom filter from bot B_a to the target T. This filter $F_a(fid_\alpha)$ is a function of some unique flow identifier fid_α, which the attacker knows. The attacker cannot send data to the target with any other flow identifier than fid_α.

As explained in Section IV, the first step for the attacker is to construct a path between its bots B_b and B_c. Since both bots are controlled by the attacker, it may freely choose the flow identifier. The attacker creates the path filter from B_b to B_c using the known flow identifier fid_α, thus creating a filter $F_b(fid_\alpha)$. Now, in the simplest case where no permutations are used, the attacker can construct a Bloom filter from B_b to the target T with a simple bitwise OR similar to (1): $F_x(fid_\alpha) = F_a(fid_\alpha) \vee F_b(fid_\alpha)$. This works because the same flow identifier is used for every flow.

The permutations or other previously discussed security mechanisms do not affect this attack. In order to enforce the flow-specific identifiers, the identifiers would have to be assigned by a trusted topology manager that also creates all the filters used in the network.

V. Resubscription Attack

The final attack presented in this paper targets the distributed filter discovery defined by Särelä *et al.* [19]. The path Bloom filter is the bitwise OR of the FHIDs on the path. Recall that, in the distributed path discovery, this value is computed by sending a join packet from the subscriber to the publisher. The join packet starts with an all-zero filter, and the FHIDs are added to it hop by hop.

The problem with the distributed filter discovery is that nothing forces the subscriber to start the join packet off with an empty filter. A malicious subscriber may set the initial filter in the join packet to any filter value that it knows. For example, when some honest subscribers leave the multicast group, the malicious subscriber can observe this as a change in the Bloom filter of the received multicast packets. It can then send a join packet with the old multicast filter as the initial filter value. This will prevent any nodes from leaving the multicast group.

It should be noted that this resubscription attack works just as well with filter permutations. This is because the permutations done in the downstream direction for the filter in the packet headers and in the upstream directions for the filter in the join packet cancel each other out. Frequent updating of the FHIDs, on the other hand, can reduce the effect of the attack because the old filters will cease to function after the update. In order to prevent the resubscription attack completely, departures from the multicast group need to be timed together with global FHID updates.

VI. Discussion

One lesson from the attacks described in this paper is that the Bloom filters and their operations are not secure cryptographic constructs. It is possible to analyze and combine them in various unexpected ways in order to derive information and new filters that are useful in attacks. The correlation attack suggested by Rothenberg *et al.* [2] can be implemented, and we have seen other, even more effective attacks that combine forwarding paths in clever ways.

The natural question to ask at this point is whether any variant of Bloom-filter forwarding escapes the attacks presented in this paper. Some do. If the FHIDs and filters are computed by a trusted topology manager as a function of unique flow identifiers that are chosen by the topology manager, then the filters for all flows are independent of each other and cannot be combined with bitwise operations as required by the reverse-engineering or traffic-injection attacks. Moreover, if the FHIDs are updated frequently, that minimizes the impact of the resubscription attack. Unfortunately, the flow-specific identifiers require the forwarding nodes to compute the hash functions for each packet, which has serious implications on the forwarding performance. The cryptographic computation would completely negate the original argument that Bloom-filter forwarding is efficient because only simple bitwise operations on the filters are needed at each forwarding hop.

All the defense mechanisms that do not require per-packet hash computation at the forwarding nodes are less secure against the injection attack. Two proposed mechanisms stand out because they can significantly reduce the probability of two paths meeting in a way that allows a bot to participate in the injection attack. First, inbound–outbound interface pairs are a better FHID type than node or link identifiers because they result in a larger connectivity graph and more independent paths across the network. Second, the hop-count-dependent FHIDs proposed in LIPSIN restrict meeting points of two paths F_a and F_b to be at the same distance from the beginning of the two paths. This will, however, work only if the filters are computed by a trusted topology manager so that the attacker cannot tamper with the hop count used for filter creation. Moreover, the defense is fully effective only if ingress filtering is deployed to prevent the attacker from setting nonzero initial hop counts in the packets that it sends. Both of these defense mechanism increase the number of bots needed for an overwhelming flooding attack but their effectiveness depends on the circumstances such as the exact network topology and routing, which means that these are not strong security mechanisms.

Of the most important protocol proposals in the literature, FRM does not have any built-in security mechanisms, which makes the vulnerabilities described in this paper trivial

to exploit. The security of LIPSIN is based on secret and hop-count-dependent FHIDs. Additionally, loop avoidance is implemented statefully by caching recent packets. Bloomcast adds new security mechanisms: Flow-specific FHIDs that identify in–out interface pairs, filter permutations, and maximum fill factor. As explained above, some of these mechanisms can reduce the impact of the injection attack, but none can prevent it completely. The reverse-engineering attack will not be effective against Bloomcast, thanks to the filter permutations. In LANES, Bloom-filter forwarding is used for unicast forwarding. Nevertheless, LANES is vulnerable to the injection attack because even though the forwarding is intended only for unicast, its use for more complex routes cannot be enforced. This is because the forwarding nodes cannot differentiate intentional branches in the path from false positives. All the protocols are, at least to some extent, vulnerable to the resubscription attack because the timing of FHID updates has not been selected to prevent it.

A more traditional defense strategy would be to create a trusted routing and forwarding infrastructure and enforce correct routing with some combination of ingress filtering, cryptographic authentication, and accountability enforcement. These kinds of security features have been proposed for the IP routing, e.g., in the Accountable Internet Protocol (AIP) [24], self-certifying addresses [25], and packet-level authentication [26]. It may be possible to employ such measures in systems that are under a single administration, but their deployment to open networks like the global Internet is problematic. Hop-by-hop cryptographic authentication would also negate the efficiency gains of Bloom-filter forwarding.

VII. Conclusion

Bloom-filter-based multicast has been advocated as a scalable and DoS-resistant architecture. This paper presents, to our knowledge, the first systematic security analysis of the Bloom-filter-based forwarding proposals and their DoS vulnerabilities. The results indicate that the relatively abstract arguments [2], [3], [27] about the security of these protocols are inaccurate.

We explain how a distributed adversary can reverse-engineer secret link identifiers, which can then be used for intentional creation of routing loops and other anomalies. We also show that most versions of Bloom-filter forwarding are vulnerable to a form of packet injection that enables distributed flooding attacks by botnets. Moreover, we present a simple attack against distributed Bloom-filter discovery in which a subscriber prevents others from unsubscribing. These vulnerabilities were analyzed using a generalized network model, the connectivity graph, which allows us to analyze several variants of Bloom-filter forwarding in a unified way.

Our central conclusion is that Bloom-filter-based multicast is resistant to distributed packet flooding only under very stringent assumptions, i.e., when the link identifiers (or other forwarding-hop identifiers) are flow-specific and cannot be forged by the end-hosts. In practice, this requires per-packet cryptographic computation at the forwarding nodes and trusted infrastructure for discovering the Bloom filters, which would negate many of the advantages of the proposed Bloom-filter-based protocols. The protocol variants that do not implement these security mechanisms suffer from distributed DoS vulnerabilities comparable to the current Internet. This paper can be seen as a reminder that broad security claims should be presented with extreme care and that vague claims are often shown false after a more careful study.

Appendix A
Tutorial on Packet Forwarding and Filter Discovery

Fig. 7 shows an example network topology and link identifiers, one for each direction of each link. These link identifiers are m-bit bit arrays with randomly chosen k bits set (e.g., $m = 512$ and $k = 5$). Let us assume that node 1 wants to send packets to the multicast group consisting of nodes $\{3, 4\}$. The Bloom filter needed in the packet headers is $F = l_2 \vee l_6 \vee l_8$, where $\vee$ is the bitwise OR operation.

a) Packet Forwarding: When node 1 forwards a packet with the filter F, it matches the filter against the outgoing link identifiers l_2 and l_4 by checking whether $F \wedge l_2 = l_2$ and $F \wedge l_4 = l_4$, respectively. ($\wedge$ is the bitwise AND operation). The first of these checks is *true*, and the packet is forwarded to node 2. The second check is usually *false*. Node 2 computes similar checks for the outgoing links l_6 and l_8. Both of these are *true* and, thus, the packet is forwarded to nodes 3 an 4. These nodes still check whether the links l_9 and l_{10} might match F. The checks that usually produce the value *false* may sometimes return *true* because of a false positive in the Bloom filter. The false positives cause the packet to be forwarded to some nodes unnecessarily, but the protocol parameters are selected so that this does not cause significant overhead.

b) Filter Discovery With Topology Manager: In order to send packets to the multicast group, node 1 needs to first learn the value of the filter F. It may obtain this from a trusted topology manager. The topology manager handles subscriptions, i.e., tracks multicast group membership, computes the filters, and only gives the filter F to node 1, which is the authorized publisher for the group.

c) Distributed Filter Discovery: An alternative to the topology manager is distributed filter discovery. In that case, the subscribers (nodes 3 and 4) send *join packets* toward the publisher (node 1). The old Internet routing mechanisms may be used for passing the join packets upstream to the publisher. For example, node 3 creates a join packet with an empty (all-zero) Bloom filter and passes it to node 2. Node 2 sees that the join packet came from link l_5, and it adds the reverse link identifier l_6 to the filter with bitwise OR. Node 2 then passes the join packet, now containing the filter value l_6, to node 1. Node 1 also adds the reverse link l_2 to the filter. The filter value is now $l_6 \vee l_2$. This is the path filter from node 1 to node 3. Similarly, node 4 sends a join packet upstream to node 1, and its path filter $l_8 \vee l_2$ is collected on the way. To send to the multicast group $\{3, 4\}$, node 1 computes the bitwise OR of the path filters: $(l_6 \vee l_2) \vee (l_8 \vee l_2)$. This is equal to the filter $F = l_2 \vee l_6 \vee l_8$, which is exactly what node 1 needs.

d) Filter Permutations: The forwarding nodes at every hop may shuffle the Bloom filters in the multicast packets with a randomly chosen bit permutation. A bit permutation π is a

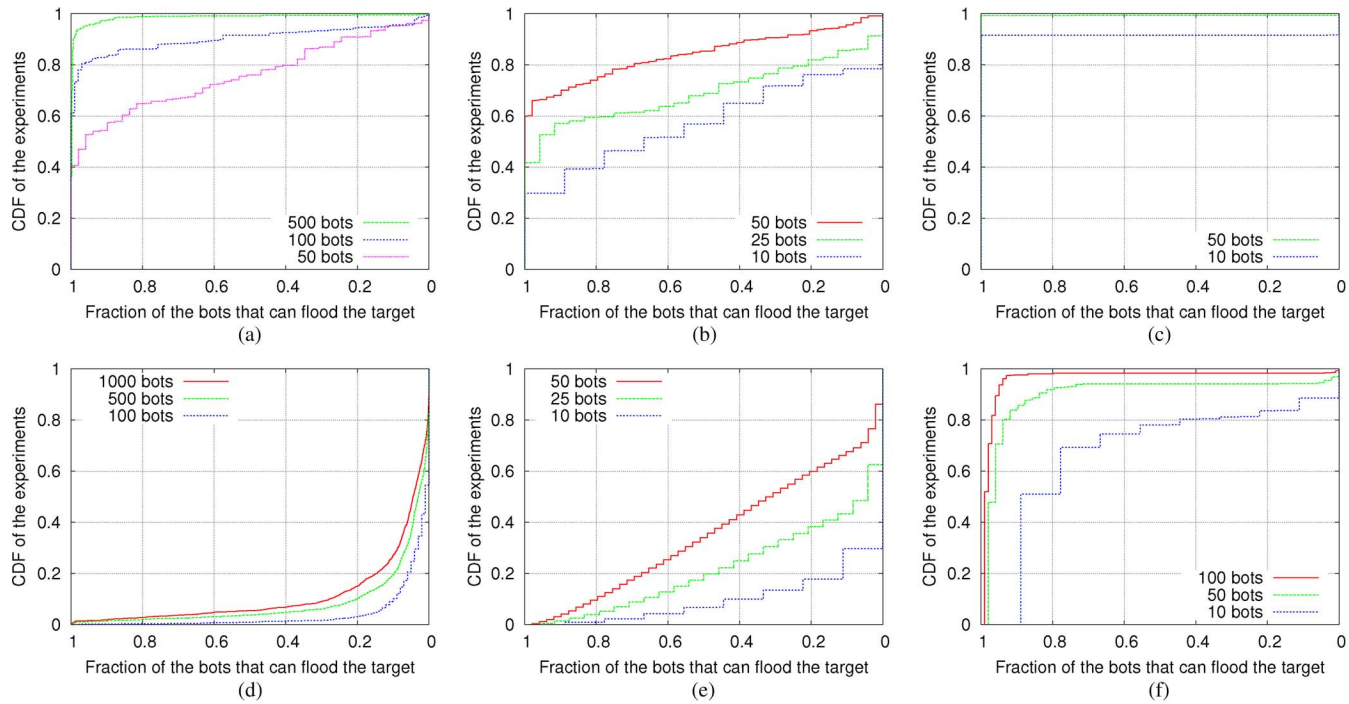


Fig. 6. Injection attack efficiency. (a)–(c) FHID = link. (d)–(f) FHID = in–out interface pair. (a) AS topology, 33 508 nodes. (b) 32 × 32 grid, 1024 nodes. (c) 5-ary tree of depth 7, 19 531 nodes. (d) AS topology, 33 508 nodes. (e) 32 × 32 grid, 1024 nodes. (f) 5-ary tree of depth 7, 19 531 nodes.

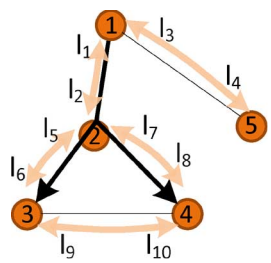


Fig. 7. Multicast tree example.

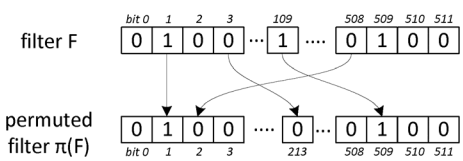


Fig. 8. Filter permutation shuffles the bits of the Bloom filter.

function that shuffles the order of the bits in a Bloom filter. Fig. 8 shows an example of this process for a 512-bit Bloom filter. Bit permutations are familiar from their use as P-boxes in symmetric cryptographic algorithms. In Bloom-filter-based forwarding, the aim of this security measure is to obscure the filter values from malicious nodes and to prevent forwarding loops and flow duplication.

In distributed filter discovery, each node that passes the join packet upstream needs to perform the inverse permutation. In centralized filter discovery, the topology manager needs to compute the path filters similarly in upstream direction and perform the inverse permutations on the partial filters at each node. The inverse permutation done during the upstream filter creation process ensures that when the filter is permuted during the downstream forwarding of multicast packets, the bits will be in the right locations of the filter at each forwarding node, so that they can be correctly compared to the next link identifier.

To see why the bit permutations do not break the forwarding mechanism, let us consider Fig. 7 again. Now, each forwarding node i is assigned a random but unchanging bit permutation π_i, and it permutes the Bloom filters in the packets before other processing. During the path discovery, the reverse permutations π_i^{-1} are applied. The path filter from node 1 to 3 becomes $\pi_1^{-1}(\pi_2^{-1}(l_6) \vee l_2)$. Similarly, the path filter from node 1 to 4 is $\pi_1^{-1}(\pi_2^{-1}(l_8) \vee l_2)$. The bitwise OR of these gives the filter that will be used in the data packets $\pi_1^{-1}(\pi_2^{-1}(l_6) \vee l_2) \vee \pi_1^{-1}(\pi_2^{-1}(l_8) \vee l_2)$, which is equal to $\pi_1^{-1}(\pi_2^{-1}(l_6) \vee \pi_2^{-1}(l_8) \vee l_2))$. When node 1 processes this data packet, it first applies the permutation π_1 and then checks whether the filter contains l_2 or l_4. The filter in the forwarded packet is updated to $\pi_2^{-1}(l_6) \vee \pi_2^{-1}(l_8) \vee l_2$. Node 2 applies π_2 to this. The result is $l_6 \vee l_8 \vee \pi(l_2)$, which node 2 matches with l_6 and l_8. This is also the filter value in the packets forwarded by node 2. As can be seen, the permutations are unwound exactly right before the matching.

Appendix B
Network Topologies Used in the Simulations

The attacks presented in Sections III and IV were evaluated with three different network topologies: a measured Internet AS graph [28], planar 32 × 32 grid graph, and 5-ary tree of depth 7. We have chosen these three drastically different topologies because Bloom-filter forwarding protocols have been proposed for many different types of networks and none has yet become their

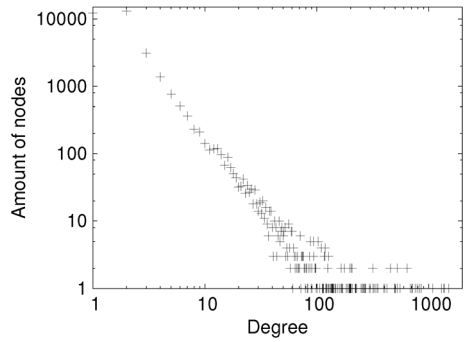


Fig. 9. Degree distribution of the internet AS topology [28].

TABLE II
NETWORK TOPOLOGIES USED IN THE SIMULATIONS

topology	nodes	links	k	path selection
AS-graph [28]	33 508	150 002	12	Shortest valley-free path [29]. Random tie-breaking for equal-length paths.
Grid graph	1 024	1 984	4	Shortest path. Links were assigned a random weight $1.0 < w < 2.0$.
5-ary tree	19 531	39 060	5	Shortest path

standard operating environment. Table II summarizes the properties of the topologies and the forwarding-protocol parameters used in each of them.

The main differences between the topologies are the node degree distribution and the likelihood of two paths meeting. While the node degrees are nearly constant in the artificial topologies, the node degrees of the AS topology follow the power-law distribution (see Fig. 9). In the tree topology, there is only one possible path between two nodes and almost any two long paths will cross each other. In the planar grid topology and in the core of the AS graph, there are many more independent paths.

The number of Bloom-filter hash functions k (i.e., number of 1-bits in the FHIDs) was selected separately for each topology to match the node-degree distribution. In line with most of the literature, we have chosen to use a constant k for each network. In a topology with highly variable node degrees such as the AS topology, it would be preferable to use variable k that is set separately for each node (the number of 1-bits in the FHIDs growing logarithmically with the node degree) [4]. That would affect the details of our analysis, but not the fundamental results of this paper.

APPENDIX C
ANALYTICAL MODEL OF TREE TOPOLOGY

This appendix provides an analytical model of the injection attack in the tree-topology network when the FHID is the in–out interface pair. The purpose of the analysis is to confirm the simulation results in Section IV. Specifically, the following calculations correspond to Fig. 6(f).

The number of nodes in a b-ary tree of depth i is $\mathrm{TS}(i) = (b^{i+1} - 1)/(b - 1)$. We consider a network topology that is a full b-ary tree of depth d and start by placing randomly the target node T and the bot B_a in the network. In the tree topology,

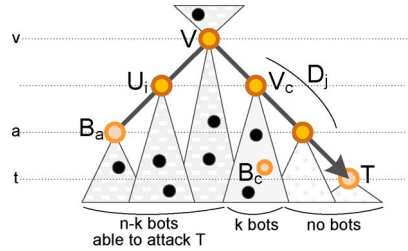


Fig. 10. Variables in the analytical model.

the path from B_a to T [see Fig. 4(a)] first traverses up the tree and then, at a node that we denote by V, turns down toward T. We denote the nodes on this path by $B_a, U_i (i = a+1 \dots v-1), V, D_j (j = t+1 \dots v-1), T$ and their levels in the tree by a, i, v, j, t, respectively. (Leaf nodes are on level zero, and the root on level d.) The variables are illustrated in Fig. 10. In random placement, t, v and a have the follow probability distributions:

$$P_T(t) = P(T \text{ is on level } t) = \frac{b^{d-t}}{\mathrm{TS}(d)}; \qquad t = 0 \dots d \tag{8a}$$

$$P_V(v|t) = P(V \text{ is on level } v \mid T \text{ is on level } t) = \frac{(\mathrm{TS}(v) - \mathrm{TS}(v-1))}{\mathrm{TS}(d)}; \qquad v = t+1 \dots d \tag{8b}$$

$$P_A(a|v) = P(B_a \text{ is on level } a \mid V \text{ is on level } v) = \frac{b^{v-a}}{\mathrm{TS}(v)}; \qquad a = 0 \dots v. \tag{8c}$$

Note that we intentionally choose T first, then V, and B_a last and compute the conditional probabilities in this order.

Next, we consider random placement of n bots in the network and which of them will be used as B_c in the injection attack. In Section IV-B, we observed that if there is even one bot close to T in the network, it may be used as B_c to facilitate the flooding attack from many other bots. However, we did not define what is considered "close" because it depends on the network topology and routing. In the tree topology with shortest-path routing, let us consider the path from B_a to T. From each node on the path, there are *branches* leading to other subtrees. V has additionally a branch up toward the root, unless V itself is the root. Each of the n bots is located on some such branch. Importantly, we denote by V_c the node on the path from B_a to T that is *closest* to T along the path, such that some $k > 0$ of the n bots are placed on the branches starting from V_c. The attacker may pick an arbitrary one of the k bots as B_c. In the tree topology, B_c and any other bots placed on the branches starting from V_c are not able to send to T using the injection attack. However, all the other bots, which are on more distant branches, can flood the target with the help of B_c. This is because the paths from them to B_c share at least one link with the path from B_a to T.

Let us enumerate the nodes on the path from B_a to T and consider them as potential V_c. The total size of the branches starting from the node under investigation is denoted by C, and the total size of the later branches (ones closer to T) is denoted by E. In order for the current node to be V_c, the random placement of n bots must have landed no bots in the E nodes and $k > 0$ bots in

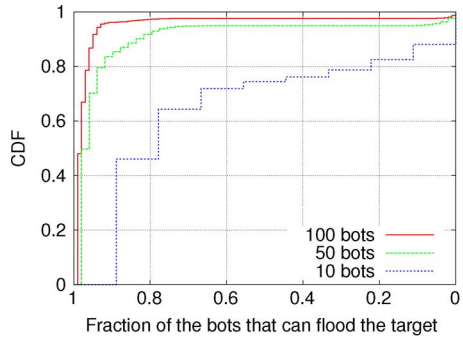


Fig. 11. Analytical results in the tree topology.

the C nodes. This allows us to compute the probability that the current node is V_c with a specific value of k

$$\text{For } 0 < k \leq n,$$
$$P^*(k|E,C) = \left(1 - \frac{E}{\mathrm{TS}(d)}\right)^n \binom{n}{k} \left(\frac{C}{\mathrm{TS}(d)-E}\right)^k \times \left(1 - \frac{C}{\mathrm{TS}(d)-E}\right)^{n-k}. \quad (9)$$

For the five types of nodes on the path from B_a to T, we get the following probabilities of each one being V_c with specific values $k > 0$:

$$\begin{aligned} P(V_c = B_a, k) &= P^*\left(k \mid \mathrm{TS}(d) - \mathrm{TS}(a), \mathrm{TS}(a)\right) \\ P(V_c = U_i, k) &= P^*\left(k \mid \mathrm{TS}(d) - \mathrm{TS}(i), \mathrm{TS}(i) - \mathrm{TS}(i-1)\right) \\ P(V_c = V, k) &= P^*\left(k \mid \mathrm{TS}(v-1), \mathrm{TS}(d) - 2 \cdot \mathrm{TS}(v-1)\right) \\ P(V_c = D_i, k) &= P^*\left(k \mid \mathrm{TS}(i-1), \mathrm{TS}(i) - \mathrm{TS}(i-1)\right) \\ P(V_c = T, k) &= P^*\left(k \mid 0, \mathrm{TS}(t)\right). \end{aligned} \quad (10)$$

The first formula covers the rare case where the closest bot to T and therefore all the bots are under B_a in the tree. It evaluates to nonzero only for $k = n$. The other formulas cover the path from B_a up to V and down to T. The last one represents the fairly common situation where there is at least one bot under T in the tree. From these values, we can accumulate the probability distribution for the number k of bots that are *unable* to flood the target

$$\begin{aligned} P(k) = \sum_{t=0}^{d} \Bigg(P_T(t) \sum_{v=t+1}^{d} \Bigg(P_V(v|t) \sum_{a=0}^{v} \Big(P_A(a|v) \\ \times \Bigg(P(V_c = B_a, k) + \sum_{i=a+1}^{v-1} P(V_c = U_i, k) \\ + P(V_c = V, k) + \sum_{i=t+1}^{v-1} P(V_c = D_i, k) \\ + P(V_c = T, k) \Bigg) \Bigg) \Bigg) \Bigg); \qquad k = 1 \ldots n. \end{aligned} \quad (11)$$

The results from this formula are shown in Fig. 11, and they correspond closely to the simulation results seen earlier in Fig. 6(f).

REFERENCES

[1] S. Ratnasamy, A. Ermolinskiy, and S. Shenker, "Revisiting IP multicast," *Comput. Commun. Rev.*, vol. 36, no. 4, pp. 15–26, 2006.

[2] C. Rothenberg, P. Jokela, P. Nikander, M. Särelä, and J. Ylitalo, "Self-routing denial-of-service resistant capabilities using in-packet Bloom filters," in *Proc. Eur. Conf. Comput. Netw. Defense*, 2009, pp. 46–51.

[3] P. Jokela, A. Zahemszky, C. Esteve Rothenberg, S. Arianfar, and P. Nikander, "LIPSIN: Line speed publish/subscribe inter-networking," in *Proc. ACM SIGCOMM*, 2009, pp. 195–206.

[4] M. Särelä, C. E. Rothenberg, T. Aura, A. Zahemszky, P. Nikander, and J. Ott, "Forwarding anomalies in Bloom filter based multicast," in *Proc. 30th IEEE INFOCOM*, 2011, pp. 2399–2407.

[5] C. Macapuna, C. Rothenberg, and M. Magalh aes, "In-packet Bloom filter based data center networking with distributed OpenFlow controllers," in *Proc. IEEE GLOBECOM Workshops*, Dec. 2010, pp. 584–588.

[6] C. Rothenberg, C. Macapuna, F. Verdi, M. Magalhães, and A. Zahemszky, "Data center networking with in-packet Bloom filters," in *Proc. 28th SBRC*, Gramado, Brazil, 2010, pp. 553–566.

[7] A. Zahemszky, P. Jokela, M. Särelä, S. Ruponen, J. Kempf, and P. Nikander, "MPSS: Multiprotocol stateless switching," in *Proc. IEEE INFOCOM Workshops*, 2010, pp. 1–6.

[8] J. Keinänen, P. Jokela, and K. Slavov, "Implementing zFilter based forwarding node on a NetFPGA," in *Proc. NetFPGA Dev. Workshop*, 2009, pp. 1–8.

[9] A. Ghani and P. Nikander, "Secure in-packet Bloom filter forwarding on the NetFPGA," in *Proc. 1st Eur. NetFPGA Dev. Workshop*, 2010, pp. 1–7.

[10] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, pp. 422–426, Jul. 1970.

[11] B. Grönvall, "Scalable multicast forwarding," *Comput. Commun. Rev.*, vol. 32, pp. 68–68, January 2002.

[12] X. Tian, Y. Cheng, and B. Liu, "Design of a scalable multicast scheme with an application-network cross-layer approach," *IEEE Trans. Multimedia*, vol. 11, no. 6, pp. 1160–1169, Oct. 2009.

[13] X. Tian, Y. Cheng, and X. Shen, "DOM: A scalable multicast protocol for next-generation Internet," *IEEE Netw.*, vol. 24, no. 4, pp. 45–51, Jul.–Aug. 2010.

[14] S. Silva, R. Silva, R. Pinto, and R. Salles, "Botnets: A survey," *Comput. Netw.*, vol. 57, no. 2, pp. 378–403, 2012.

[15] J. K. Mullin, "A second look at Bloom filters," *Commun. ACM*, vol. 26, pp. 570–571, Aug. 1983.

[16] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *Comput. Surv.*, vol. 35, pp. 114–131, June 2003.

[17] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol label switching architecture," Internet Engineering Task Force, RFC 3031 (Proposed standard), Jan. 2001 [Online]. Available: http://www.ietf.org/rfc/rfc3031.txt

[18] K. Visala, D. Lagutin, and S. Tarkoma, "LANES: An inter-domain data-oriented routing architecture," in *Proc. ReArch*, 2009, pp. 55–60.

[19] M. Särelä, C. E. Rothenberg, A. Zahemszky, P. Nikander, and J. Ott, "BloomCasting: Security in Bloom filter based multicast," in *Nordsec 2010 Conference*, 2011.

[20] M. Särelä, "BloomCasting for publish/subscribe networks," Ph.D. dissertation, Aalto Univ., Espoo, Finland, 2011.

[21] R. Lee, Z. Shi, and X. Yang, "Efficient permutation instructions for fast software cryptography," *IEEE Micro,*, vol. 21, no. 6, pp. 56–69, Nov.–Dec. 2002.

[22] Y. Hilewitz, Z. Shi, and R. Lee, "Comparing fast implementations of bit permutation instructions," in *Conf. Rec. 38th Asilomar Conf. Signals, Syst. Comput.*, 2005, vol. 2, pp. 1856–1863.

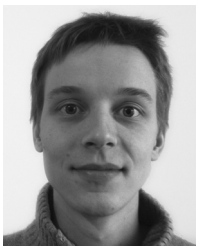
[23] Z. Shi, X. Yang, and R. Lee, "Arbitrary bit permutations in one or two cycles," in *Proc. IEEE Int. Conf. Appl.-Specific Syst., Archit., Processors*, 2003, pp. 237–247.

[24] T. Anderson, T. Roscoe, and D. Wetherall, "Preventing Internet denial-of-service with capabilities," *Comput. Commun. Rev.*, vol. 34, no. 1, pp. 39–44, 2004.

[25] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel, "Separating key management from file system security," *Oper. Syst. Rev.* vol. 33, pp. 124–139, Dec. 1999.

[26] D. Lagutin, "Securing the Internet with digital signatures," Ph.D. dissertation, School of Science and Technology, Aalto University, Espoo, Finland, 2010.

[27] M. Ain, S. Tarkoma, D. Trossen, and P. Nikander, "Conceptual architecture of PSIRP including subcomponent descriptions," PSIRP project, Deliverable D2.2, 2008.
[28] CAIDA, La Jolla, CA, USA, "The CAIDA AS relationships dataset," Jan. 20, 2010 [Online]. Available: http://www.caida.org/data/active/as-relationships/
[29] L. Gao, "On inferring autonomous system relationships in the Internet," *IEEE/ACM Trans. Netw.*, vol. 9, no. 6, pp. 733–745, Dec. 2001.



Markku Antikainen received the M.Sc. degrees in security and mobile computing from Aalto University, Espoo, Finland, and the Royal Institute of Technology, Stockholm, Sweden, in 2011, and is currently pursuing the Ph.D. degree in computer science at Aalto University.

Before starting the graduate studies, he worked as a Security Consultant with Nixu, Ltd., Espoo, Finland.

Tuomas Aura received the M.Sc. and Ph.D. degrees from Helsinki University of Technology, Espoo, Finland, in 1996 and 2000, respectively. His doctoral thesis was on authorization and availability in distributed systems.

He is a Professor of computer science and engineering with Aalto University, Espoo, Finland. Before joining Aalto University, he worked with Microsoft Research, Cambridge, U.K. He is interested in network and computer security and the security analysis of new technologies.

Mikko Särelä received the M.Sc. degree in computer science from Helsinki University of Technology, Espoo, Finland, in 2004, and the Dr. Tech. degree in electrical engineering from Aalto University, Espoo, Finland, on his work on information-centric network architectures in 2011.

He works as a Post-Doctoral Researcher with Aalto University. During his Ph.D. studies, he worked with Nomadiclab, Ericsson, Kirkkonummi, Finland, on information-centric network architecture and future Internet. His current research interests include Ethernet scalability and security, energy efficiency, and mobility.