OmniSphinx: Active Mix Networks

Daniel Schadt[1][0009−0009−6357−1314], Christoph Coijanovic[1][0000−0002−5873−2859], Shabi Shabani[2][0009−0001−7308−7587], and Thorsten Strufe[1][0000−0002−8723−9692]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
{daniel.schadt,christoph.coijanovic,thorsten.strufe}@kit.edu
[2] Karlsruhe Institute of Technology, Karlsruhe, Germany
mail@shabi-shabani.de

Abstract. Mix networks are an important tool to implement anonymous communication, which protects not just the content but also the metadata of messages. Over time, various packet formats for mix networks have been proposed, usually with single, specific goals in mind. These formats are incompatible with each other, requiring separate software and infrastructure to be set up. In this paper, we propose a new format, *OmniSphinx*, which solves this issue. In OmniSphinx, senders embed code in their packets that determines how they must be processed. The resulting active mix network can emulate any other mix format within a single deployment. Our empirical evaluation shows that emulation in OmniSphinx incurs reasonable overhead compared to native execution for typical mix network use cases: For Sphinx, the most compact format, computation time increases by around 90 µs, while headers increase by 33% in size.

Keywords: Mix nets · Anonymous communication · Onion encryption.

1 Introduction

The internet has become an invaluable tool for modern communication. Such communication, however, can be observed by the service providers and third parties. Many services thus employ encryption to protect the content of their communication, but metadata – information such as *who talks to whom* or *when do they talk* – is left unprotected. This metadata alone often reveals sensitive information, for instance a frequent contact with medical professionals or specific political groups.

Mix networks hide such metadata by relaying a message over a series of intermediaries (the *mix nodes*). This provides anonymity, but comes at the cost of fixed message layouts and limited functionalities: As each packet must be encrypted in a specific format, only communication patterns that are supported by the chosen format can be used. This puts pressure on mix network operators to support the most popular format, and limits users that need specific functionalities like multicast traffic. Ideally, a single mix network would not just support

any format that currently exists [6,11,19,12,16], but also be easily extended for future formats without requiring coordinated infrastructure updates.

To work towards such a mix network, we take ideas from "active networking" [21,4,1], a line of research that aimed to add customization to general purpose network nodes (e.g., to implement custom compression). A promising concept is to combine the code needed to process a packet with the payload in a self-contained structure called a *capsule*. Due to the performance penalty that this approach induces, it has not been adopted widely, especially as use-cases were lacking to justify the cost [10]. However, we see a good use-case in mix networks because it allows the nodes to be oblivious to the different mix formats.

Capsules inherently cause overhead, so the goal of this work is to investigate whether this approach is still viable within the scope of mix networks: As message decryption and shuffling already incur a higher latency than direct communication, we expect the performance penalty to be less of an issue. At the same time, there are concrete benefits to adding such dynamicity to mix networks: A single mix network instance can be used by clients with different requirements regarding the mix format, leading to better utilization of the instance, and potentially, a larger anonymity set. Further, if more mix nodes provide the functionality a client needs, they can choose from a more diverse set of nodes and operators.

We identify three challenges when applying active networking to mix formats: First, we have to ensure that the resulting format is flexible enough to emulate all relevant existing mix formats. Second, this flexibility should not come at the cost of privacy – an emulated mix format must achieve the same privacy guarantees as the native one. Third, the added flexibility should not incur costs that inhibit typical use cases of mix networks, such as email communication.

For our investigation, we develop OmniSphinx, a novel *active* mix format that achieves flexibility in mix networks. OmniSphinx is based on Sphinx, but extends the packet header with a *mix program* for each node on the packet's path. These mix programs are built from an instruction set that is tailored towards the required operations for existing mix formats, and they determine how the packet's payload should be processed at each node. This allows OmniSphinx to emulate all relevant existing mix formats. Our evaluation shows that OmniSphinx can emulate Sphinx with an overhead of 33% in header size, and additional 90 µs in computation time.

Finally, we recognize that OmniSphinx requires senders to be careful when assembling mix programs, as to not make packets traceable by accident. To help senders in constructing "secure" mix programs, we provide *information flow* rules that can be used to assess whether a mix program leaks information.

In summary, we make the following main contributions:

- We develop OmniSphinx, a novel mix format that implements ideas from active networking to make mix networks flexible.
- We present an instruction set that allows OmniSphinx to emulate all existing relevant mix formats.
- We introduce information flow analysis to the context of anonymous communication to argue about the privacy protection of arbitrary mix programs.

– We provide an empirical evaluation to determine bandwidth and computational overhead of OmniSphinx compared to non-flexible mix formats.

This paper is structured as follows: Section 2 introduces the necessary background. Section 3 compares our goals to related work. In Section 4, we introduce our assumed system and threat model. Section 5 gives a high-level overview of OmniSphinx's design, while Section 6 discusses the format in more technical detail. In Section 7, we describe how OmniSphinx can emulate Sphinx and PolySphinx. Section 8 discusses OmniSphinx's security guarantees, while Section 9 discusses its performance impact. Finally, Section 10 concludes this work.

2 Background

In this section, we introduce the general notation as well as specific background knowledge required for this paper.

2.1 Notation

We denote the security parameter as κ. Breaking the cryptographic primitives requires work that scales exponentially in κ. We denote the set of possible bytes as $\mathbb{N}_{255}$, the set of byte strings of length l as $\mathbb{N}_{255}^{l}$, and the set of byte strings of arbitrary length as $\mathbb{N}_{255}^{*}$. Given $a \in \mathbb{N}_{255}^{x}$ and $b \in \mathbb{N}_{255}^{y}$, $a||b \in \mathbb{N}_{255}^{x+y}$ denotes the concatenation of a followed by b. If $x = y$, $a \oplus b \in \mathbb{N}_{255}^{x}$ denotes the bitwise xor of a and b. Further, $a_{[i..j]}$ with $i \leq j < x$ denotes the substring of a from i to j, and 0^i denotes the string of i zeroes. We denote a Diffie-Hellman group as $\mathcal{G}$. Its order is q and its generator g.

2.2 Mix networks

Mix networks [5] are a tool for anonymous communication. In a mix network, Alice sends a message to Bob over a series of intermediaries (the *mix nodes*), which hides their relation. To prevent an outside observer from tracking messages as they pass through mix nodes, two techniques are employed:

First, messages are encrypted multiple times, with each mix node removing one layer of encryption. This concept is colloquially known as onion encryption and prevents the observer from tracking messages based on their content. A *mix format* describes the exact way how messages are encrypted.

Second, messages are intentionally delayed and shuffled, either by waiting for a random time before they are forwarded, or by waiting for a certain amount of other messages. This prevents the observer from tracking messages based on their timing or output order. A *mixing strategy* describes the exact way how messages are delayed and shuffled.

A mix network provides anonymity to Alice even against global active adversaries, as long as any one mix node in Alice's chosen path is honest. This is called the *anytrust* assumption.

2.3 Mix formats

The mix format describes how a sender has to prepare a message, and how a mix node has to process it. Sphinx [6] is a prominent example of a mix format, which promises compactness and provable security.

A Sphinx packet consists of a header and the payload. The header contains a key encapsulation g^x based on a Diffie-Hellman key exchange, which the node uses to derive a shared secret g^{xy} using its private key y. This shared secret keys a stream-cipher, which is used to decrypt the remaining header, as well as a MAC, which is used to verify the integrity of the header, as well as a block-cipher, which is used to decrypt the payload. After decryption, the node can read the address of the next hop. Finally, the shared secret is blinded to produce a new key encapsulation, allowing the next node to derive a fresh shared secret.

Thanks to its compactness, Sphinx has been used as the basis for later adaptions and extensions: Beato et al. propose an improvement of Sphinx that uses authenticated encryption instead of the explicit hashes that Sphinx uses [3]. We call this variant *AE-Sphinx*. Hugenroth et al. propose *MultiSphinx* [11], a format that allows multiple (small) Sphinx packets to be combined into a single big packet. This allows for a more efficient sending of multiple small packets. Schadt et al. propose *PolySphinx* [19], a format that allows for efficient replication of messages with the same content to multiple recipients. This allows for efficient group communication.

Outside of Sphinx, Klooß et al. propose *EROR* [12], which requires weaker trust assumptions than Sphinx. EROR includes hop-by-hop integrity protection, which prevents tagging attacks in the presence of malicious receivers.

3 Related work

The goal of this work is to define a packet format for mix networks that offers flexibility to emulate existing and future mix formats.

OmniSphinx shares its setting and basic architecture with Sphinx [6] and Sphinx-like mix formats [11,19]. Unlike OmniSphinx, these formats rigidly define packet processing as part of the protocol and do not offer flexibility.

Rochet et al. propose **FAN** [17], a protocol that enables dynamic reprogramming of Tor [8] nodes. FAN's goal is to increase the speed with which updates to the Tor code base propagate through the network. FAN offers flexibility in the sense that the protocol can be easily updated. However, as packet processing is identical for all packets and determined by the canonical code base, it does not meet our notion of flexibility.

With **Bento** [14,15], Reininger et al. extend Tor with network function virtualization. Bento adds additional nodes to the Tor network to which users can upload code that is executed within trusted execution environments. This can be used to augment Tor's functionality, for example with cover traffic generation or load balancing. It does however have no impact on the actual packet processing, as it leaves the Tor nodes unchanged.

In the context of censorship circumvention, protocols like **Marionette** [9] and **Proteus** [22] allow proxies to be re-programmed quickly by clients. However, the focus of those systems is to be flexible in order to evade censorship systems. They do not process messages but instead aim to tunnel TCP streams covertly, and they lack the privacy guarantees that mix networks provide.

4 System and threat model

We assume a mix network consisting of *clients*, which want to send messages to each other, and *mix nodes*, which are the servers providing the service. We assume a service model [13] for the network, meaning that the final mix node in a path forwards a message to the recipient using a non-anonymous channel. We call the set of client addresses $\mathcal{C}$ (e.g., the set of fixed-length email addresses) and the set of mix node addresses $\mathcal{M}$ (e.g., IP addresses).

We assume that a public key infrastructure exists such that each mix node $m \in \mathcal{M}$ has a private key x_m, which is kept secret, and a public key $y_m = g^{x_m}$ that is known by the clients.

We assume that the adversary is global and active, meaning that it can observe any network link between two participants, and it can insert, modify, delay and drop messages on those links. We further assume that the adversary can control a subset $\check{\mathcal{M}} \subset \mathcal{M}$. We call nodes in $\check{\mathcal{M}}$ *corrupt*, and the remaining nodes *honest*. The adversary is bounded to use polynomial time in κ. The adversary's goal is to link sender and recipient of messages sent.

5 Design

We introduce *OmniSphinx*, a new mix format that lets senders specify how each individual packet should be processed. In OmniSphinx, senders embed a *mix program* for every node into their packet, and nodes execute the mix program to process the packet. By using different mix programs, senders can *emulate* existing and future mix formats. Note that the resulting OmniSphinx packets are not byte-wise compatible to the emulated mix format; rather, they implement the same conceptual design, meaning they provide the same functionality and achieve the same privacy guarantees.

OmniSphinx is based on Sphinx [6]. It structures a packet into two parts, the header and the payload. The header contains the mix programs, whereas the payload contains the sender's message. Like Sphinx, the base packet format employs onion encryption and MACs in the header to ensure integrity of the mix programs and that each mix node only learns "their" mix program. Unlike existing mix formats, OmniSphinx does not specify how payloads should be handled to give senders as much flexibility as possible. All payload processing is done according to the mix program.

Each mix program consists of a sequence of *instructions*. The available instructions are defined by the *instruction set*. The instruction set is chosen to provide a good trade-off between flexibility and overhead: If the instructions

are too low-level (e.g., x86 machine code), representing a mix program takes up many instructions and becomes inefficient. On the other hand, if the instructions are too high-level (e.g., switching between different existing formats), then OmniSphinx deployments will not support new formats without updating the instruction set, which defeats its purpose. Additionally, the instructions are chosen to limit attacks from malicious users against mix nodes, like the attempted extraction of secrets or the abuse of computation time.

Packet processing at mix nodes is divided into three stages:

- During *preprocessing*, the node derives the shared secret and unwraps the onion encryption in the header. This reveals the mix program for the current node.
- During *program execution*, the node executes the instructions specified in the mix program. The program has access to the header, payload, and shared secret. A dedicated `Forward` instruction is used to enqueue outgoing packets for transmission.
- During *postprocessing*, the mix node ensures that each outgoing packet has the correct size, padding it if necessary.

As outgoing packets are created by the mix program, a single packet can result in no outgoing packets (e.g., to implement cover traffic), a single outgoing packet (e.g., a normal message), or multiple outgoing packets (e.g., for multicast traffic).

6 Implementation

In this section, we define the OmniSphinx format in detail.

6.1 Cryptographic primitives

We make use of a number of cryptographic primitives in our implementation of OmniSphinx: We use a secure pseudorandom generator, $\rho : \mathbb{N}_{255}^{\kappa} \to \mathbb{N}_{255}^{*}$, which takes as input a seed and outputs arbitrarily many random looking bytes. We use a MAC, which we model as a random oracle $\mu : \mathbb{N}_{255}^{\kappa} \times \mathbb{N}_{255}^{*} \to \mathbb{N}_{255}^{\kappa}$, taking the input key and bytes and producing an "authentication tag." We use hash functions $h_b : \mathcal{G} \times \mathcal{G} \to \mathbb{N}_q$ to blind group elements between hops, $h_\rho, h_\mu : \mathcal{G} \to \mathbb{N}_{255}^{\kappa}$ and $h'_\rho : \mathcal{G} \times \mathbb{N}_{255} \times \mathbb{N} \to \mathbb{N}_{255}^{\kappa}$ to derive keys from group elements, and $h_\tau : \mathcal{G} \to \mathbb{N}_{255}^{\kappa}$ to identify already-seen messages.

6.2 Instruction architecture

Instructions in OmniSphinx follow a register-based approach, where intermediate values are saved in one of $|\mathcal{R}| = 256$ registers. Each register $r \in \mathcal{R}$ can hold a byte string $b \in \mathbb{N}_{255}^{*}$. Each instruction takes a predefined number of arguments, where each argument can be either the index of a register to use as in-/output, or a constant value.

We write instructions as `Name(a, b, ...)`, where `Name` is the name of the instruction and $\mathtt{a}, \mathtt{b}, \ldots \in \mathcal{R} \cup \mathbb{N}_{255}$ are its arguments. Internally, each instruction is encoded as a byte representing the operation code, followed by bytes representing the arguments.

A special case is the `Load` instruction, which is used to embed byte string constants into the mix program. The `Load` instruction takes the form `Load(v, d)`, where `v` is the value to load and $\mathtt{d} \in \mathcal{R}$ is the destination register. Internally, `Load` is encoded with a byte representing the operation code for `Load`, followed by two bytes specifying the length of `v`, followed by the bytes of `v`, followed by a byte representing `d`.

A full list of the available instructions can be found in Table 1.

Instruction	Description
`Exponent`(b, e, d)	Computes b^e and stores the result in d
`ConcatByte`(a, b, d)	Concatenates the byte value of b to a and stores the result in d
`Concat`(a, b, d)	Concatenates the content of b to a and stores the result in d
`IsEqual`(a, b)	Checks if a and b are equal. Aborts if false.
`CutBytes`(a, l, d)	Extracts the first l bytes of a and stores the result in d
`XOR`(a, b, d)	Computes the bitwise XOR of a and b and stores the result in d
`Add`(a, b, d)	Computes the sum of a and b and stores the result in d
`Copy`(s, d)	Copies the content of s to d
`Pad`(a, l, d)	Appends l "0"s to a and stores the result in d
`Load`(s, d)	Loads the constant or intermediate value s into d
`CreateZeroes`(l, d)	Stores l "0"s in d
`PRG`(s, l, d)	Expands seed s into a pseudorandom string of length l and stores the result in d
`Hash`(s, d)	Applies the hash function to s and stores the result in d
`Encrypt`(a, b, d)	Encrypts plaintext b under key a and stores the result in d
`Decrypt`(a, b, d)	Decrypts ciphertext b under key a and stores the result in d
`MAC`(a, b, d)	Computes a MAC of b using key a and stores the result in d
`ForLoop`(a, b)	Repeat the next a instructions b times
`Forward`(a)	Forward the computed data to a
`Stop`$()$	Terminate the program execution

Table 1. The OmniSphinx instruction set.

6.3 Packet structure

We consider a packet to consist of two parts: A *header* η and a *payload* δ. OmniSphinx makes no assumptions about the payload and treats it as an opaque block of bytes $\delta \in \mathbb{N}_{255}^{l}$. The header has a pre-defined structure and contains all information required to process the packet at a mix node.

An OmniSphinx header consists of three parts: $\eta = (\alpha, \beta, \gamma)$. Here, $\alpha \in \mathcal{G}$ is a Diffie-Hellman group element, $\beta \in \mathbb{N}_{255}^{j}$ contains the mix programs and message authentication tags for the nodes along the path, and $\gamma \in \mathbb{N}_{255}^{\kappa}$ is the message authentication tag for the first node. We show this structure in Figure 1.

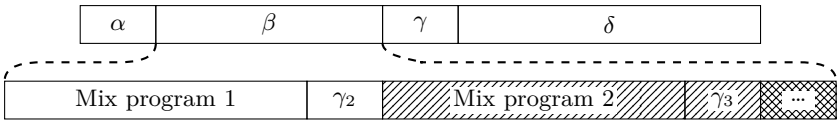


Fig. 1. Structure of an OmniSphinx packet. The lower row represents a zoomed-in view of β, in which the hatched elements are encrypted to the current node and therefore not readable.

6.4 Header creation

We describe header creation as a function that takes as input the mix programs p_0, ..., $p_{\nu-1}$ for each layer, as well as the public keys y_0, ..., $y_{\nu-1}$ that are used to encrypt the onion layers.

The client samples a random $x \leftarrow^{\$} \mathbb{N}_q^+$ and computes for each layer i the corresponding group element a_i, the shared secret s_i and the blinding factor b_i:

$$\begin{aligned}
&\alpha_0 = g^x, && s_0 = y_0^x, && b_0 = h_b(\alpha_0, s_0)\\
&\alpha_1 = g^{xb_0}, && s_1 = y_1^{xb_0}, && b_1 = h_b(\alpha_1, s_1)\\
&\dots && \dots && \dots\\
&\alpha_{\nu-1} = g^{xb_0b_1\dots b_{\nu-2}}, && s_{\nu-1} = y_1^{xb_0b_1\dots b_{\nu-2}}, && b_{\nu-1} = h_b(\alpha_{\nu-1}, s_{\nu-1})
\end{aligned}$$

Then, the client computes how the padding will look like at each node:

$$\phi_0 = 0^0 \quad \phi_i = (\phi_{i-1}||0^{|p_{i-1}|+\kappa}) \oplus \rho(h_\rho(s_{i-1}))_{[a_i..b_i]},$$

where $a_i = j - \sum_{k=0}^{i-2}(|p_k| + \kappa)$ and $b_i = j + |p_{i-1}| + \kappa$.

Finally, the client wraps the header from the inside out:

$$\begin{aligned}
\beta_{\nu-1} &= (p_{\nu-1} \oplus \rho(h_\rho(s_{\nu-v1})))_{[0..|p_{\nu-1}|]}||\phi_{\nu-1}\\
\gamma_i &= \mu(h_\mu(s_i), \beta_i)\\
\hat{\beta}_i &= p_{i+1}||\gamma_{i+1}||(\beta_{i+1})_{[0..j-|p_{i+1}|]}\\
\beta_i &= \hat{\beta} \oplus \rho(h_\rho(s_i))_{[0..j]}\\
\gamma_i &= \mu(h_\mu(s_i), \beta_i)
\end{aligned}$$

The resulting $\eta = (\alpha_0, \beta_0, \gamma_0)$ is the final header, and the resulting s_i can be used to pre-process the payload.

6.5 Header processing

Once a mix node receives a packet, it first verifies its structure by checking that $((\alpha, \beta, \gamma), \delta) \in (\mathcal{G} \times \mathbb{N}_{255}^{j} \times \mathbb{N}_{255}^{\kappa}) \times \mathbb{N}_{255}^{l}$. The node then computes the shared secret $s = \alpha^x$ using its private key x. To ensure that the received packet is fresh, the node checks whether the replay tag $h_\tau(s)$ appears in its table of tags already seen. If the packet is not fresh, it is discarded.

Next, the node verifies the integrity of the header by computing the tag $\gamma' = \mu(h_\mu(s), \beta)$. If $\gamma \neq \gamma'$, the packet is discarded.

Then, the node decrypts one layer of β as $B' = \beta \oplus \rho(h_\rho(s))$ to obtain the processing logic for this hop via the mix program p, and the message authentication tag $\hat{\gamma}$ for the next hop. The node does this by locating the `Stop` instruction in B'. The bytes until there are p, while the following κ bytes are $\hat{\gamma}$.

Finally, the node makes the values from the header accessible to the mix program by loading them into predetermined registers. Namely, it loads α, β, γ, δ, the processed versions of α, β, and the next-hop MAC $\hat{\gamma}$.

6.6 Postprocessing

For each outgoing packet, the node pads β and δ to the correct length. The padding is computed deterministically from the shared secret s and the index i of the outgoing packet as

$$\beta' = \beta || \rho(h_\rho(s, s_\beta, i)) \quad \text{and} \quad \delta' = \delta || \rho(h_\rho(s, s_\delta, i)),$$

where $s_\beta, s_\delta \in \mathbb{N}_{255}$ are arbitrarily chosen but fixed salt bytes.

7 Mix programs

In this section, we describe how OmniSphinx can be used to emulate existing mix formats at the example of Sphinx (Section 7.1) and PolySphinx (Section 7.2). We include the full mix programs in Section A.

After preprocessing, the mix node holds the the shared secret, current and next header elements, and the payload in their respective registers (see Section 6) and has the mix program loaded.

7.1 Sphinx

As OmniSphinx closely builds upon Sphinx's package format, most of the required processing for Sphinx is already done in OmniSphinx's pre- and postprocessing. This includes the derivation of the shared secret, derivation of next headers, padding, and assembly of the outgoing packet. As part of the mix program, an intermediate node first derives the decryption key by concatenating the shared secret with a salt, then `Hash`ing the result. After, `Decrypt` is called, and finally postprocessing is initialized with a `Forward` call.

If the node is an exit node, the node additionally has to extract the plaintext and receiver address from the decrypted payload. At this point, the payload should have form $\delta = 0^\kappa \parallel \Delta \parallel m$, where Δ is the receiver's address and m is the message plaintext. The node can extract the first κ bytes via `CutBytes`, then create known zeroes with `CreateZeroes`, and compare the two with `IsEqual`. Finally, the receiver address is extracted from the payload with a call of `CutBytes` and message delivery is initialized with a `Forward` call.

7.2 PolySphinx

There are three main differences between Sphinx and PolySphinx. First, clients can include multiple next headers for nodes on the path. If a node receives a packet with multiple headers, it uses them to send replicas of the packet onwards. Second, to enable replication without linkability, payloads are encrypted along the path rather than decrypted. Third, the exit node receives the key material needed to remove the encryption layers in form of a random seed that is used to generate the "key tree." The sender includes a distinct mix program for each type of node: simple forwarding, replication, and exit node.

- The forwarding node's mix program is analogous to the Sphinx one, but replaces the `Decrypt` call with an `Encrypt` one.
- The replication node's mix program includes a `For` loop with the fixed replication factor. Inside the loop, each next header is extracted via a series of `CutBytes` calls, and then used to prepare and forward a replica of the packet.
- The exit node's mix program constructs the decryption keys based on the provided seed via a series of `Hash` calls.

8 Security argument

The following section contains our security argument for OmniSphinx. Classical mix formats have provable privacy guarantees. Through the formalized privacy properties *Layer Unlinkability* (LU) and *Tail Indistinguishability* (TI) [13], one can prove that malicious mix nodes cannot link senders to receivers. A proof of LU and TI requires that the behavior of honest mix nodes is fixed, as is trivially the case in classical mix formats. This is not the case in OmniSphinx, as the sender defines the behavior of the mix node through the instructions provided in each packet. OmniSphinx thus cannot guarantee LU and TI for arbitrary mix programs.

Instead, we consider security in OmniSphinx in three aspects. First, given a fixed mix program of just a `Forward` instruction, we can prove that OmniSphinx achieves adapted versions of LU and TI. In these adaptions, we only consider the packet header and omit the payload, as OmniSphinx treats it as a black box. The actual proof then follows Sphinx's security proof [20]: The confidentiality of the shared secret follows from the Diffie-Hellman assumption, the confidentiality of the remaining header from the security of the pseudorandom generator ρ, and

the integrity follows from the MAC μ. We refer to the extended version of this paper[3] for definitions and proofs of LU and TI.

Second, to verify that a given mix program is "secure," we introduce *information flow analysis* in the context of anonymous communication in Section 8.1. Information flow analysis has been widely used since the 1970s to argue about the confidentiality guarantees of programs [7,18]. As an example, we apply information flow analysis to OmniSphinx's emulation of Sphinx.

Finally, as mix nodes execute programs given by users, we need to ensure that malicious users cannot harm mix nodes. We discuss this aspect in Section 8.2.

8.1 Information Flow Analysis

Information flow analysis tracks the observability of data during protocol execution. Program values are classified either *benign* or *malignant* prior to execution. Execution may change the state of a value. The goal is to ensure that no *malignant* information is visible in the resulting packet.

For analysis, we model the mix program as a directed dependency graph, where the nodes represent values occurring during program execution. Values are input parameters, registers initialized during preprocessing, intermediate results, or final outputs. Each node has a corresponding security label $\ell \in \{\mathit{benign}, \mathit{malignant}\}$. Edges in the graph are determined by the mix program's instructions: An instruction producing $y \leftarrow f(x_1, \ldots, x_k)$ induces an edge from each $x_{i\in[1,k]}$ to y. The label $\ell(y)$ is determined by the labels $\ell(x_i)$ for $i \in [1, k]$ as well as the concrete instruction $f(\cdot)$:

For structural and arithmetic instructions (`XOR`, `Concat`, `Copy`, …) the output retains information about the input. Thus, if any input is *malignant*, the output is also *malignant*:

$$\ell(y) := \begin{cases} \mathit{malignant} & \text{if } \exists x_{i\in[1,k]} : \ell(x_i) = \mathit{malignant} \\ \mathit{benign} & \text{else} \end{cases}$$

The cryptographic operations `Encrypt`, `Decrypt`, and `MAC` take as input a key and a message/ciphertext. If the key is *malignant*, the operation ensures that the output is computationally indistinguishable from randomness and thus can be labeled *benign*. If the key is *benign*, the operation can be re-executed by the adversary, and the output inherits the message/ciphertext's label:

$$\text{For } f \in \{\texttt{Encrypt}, \texttt{Decrypt}, \texttt{MAC}\} \text{ with } \mathit{out} \leftarrow f(\mathit{key}, \mathit{in}) :$$
$$\ell(\mathit{out}) := \begin{cases} \mathit{benign}, & \text{if } \ell(\mathit{key}) = \mathit{malignant} \\ \ell(\mathit{in}) & \text{else} \end{cases}$$

Similarly, a `PRF`'s output is *benign* given that its seed is *malignant*, as it computationally indistinguishable from randomness.

[3] https://github.com/kit-ps/OmniSphinx/blob/esorics/Extended%20Version.pdf

Within a OmniSphinx program, the inputs of a `ForLoop` are always predetermined by the sender. Any program can thus be serialized for analysis. Given a `ForLoop`(d, i) instruction, serialization repeats the following d instructions i times and removes the original `ForLoop` instruction.

Finally, the mix program contains no unintended information flow, if each input of any `Forward` instruction the program contains is *benign*.

Analysis of Sphinx As an example of how to apply information flow analysis to mix programs, we analyze the Sphinx program for OmniSphinx.

The Spinx mix program is secure, if no *malignant* information is forwarded by the mix node. If *malignant* information is forwarded, incoming packets can be linked to outgoing ones and the mix network's privacy guarantees no longer hold. As the first step in the analysis, we need to label the inputs. The incoming packet $(\alpha, \beta, \gamma, \delta)$ is *malignant*, as it was computed by and is therefore known to the previous node. The shared secret s is also labeled *malignant*, as disclosing it would enable an adversary to directly derive the outgoing packet from the incoming one themselves. The next headers α', β', γ' are labeled *benign*, as they result from cryptographic operations and are not distinguishable from randomness, assuming s is know known. The *salt* is labeled *benign*, as it is a publicly known constant. The next node is labeled *benign*, as it is inherently revealed by the node when forwarding the packet.

The shared secret s and *salt* are concatenated with `ConcatBytes` and `Hash`ed. Neither operation changes the *malignant* label of s, so the output of `Hash` remains *malignant*. The *malignant* δ and the *malignant* hash output are input into `Decrypt`. Due to the cipher's confidentiality guarantees, the output of `Decrypt` changes to *benign*. The `next` node is input into `Load`, which does not change the label. Finally, `Forward` receives the output of `Decrypt`, `Load`, as well as α', β', and γ'. All inputs of `Forward` are labeled *benign*, therefore the mix program contains no unintended information flow. A graph representation of this information flow analysis can be found in Figure 2.

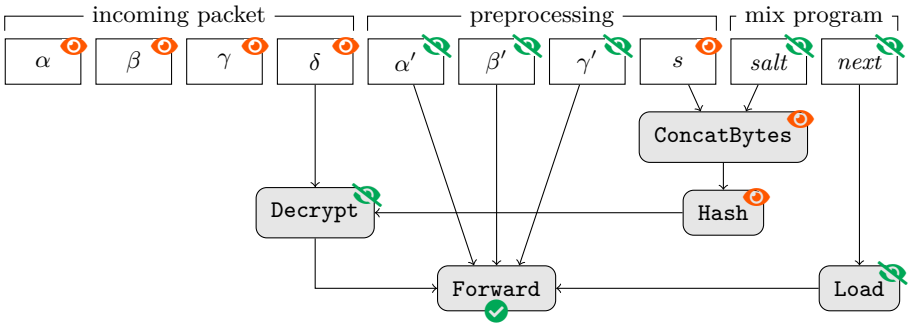


Fig. 2. Information flow analysis for the Sphinx mix program in OmniSphinx. denotes a *malignant* label, while denotes a *benign* label.

8.2 Mix node security

In terms of mix node security, we identify three potential harms: Malicious users can try to *exfiltrate* secrets from the mix node, such as its private key or information about other messages. Users can also try to *control* the node in order to use it for other malicious purposes, like becoming a part of a botnet. Finally, users can try to *jam* the node to prevent it from further work.

OmniSphinx protects against exfiltration, as the mix programs have no access to secret key material: The only value that depends on the secret key is the shared secret, which is different from packet to packet. However, the (malicious) sender already knows the shared secret, thus this provides no benefit to them. Thanks to the Diffie-Hellman key exchange, it is not possible to learn the secret key based on the shared secret. All other values that a mix program has access to come directly from the packet, and reveal no information about other packets that the mix node processes.

OmniSphinx also protects against malicious control and jamming: The instruction set is deliberately restricted to prevent arbitrary interactions with other systems. It does not allow custom network access, resource acquisition, or even infinite execution time. Additionally, the node can put further limits on the execution time or memory consumption per program that prevent a client from using up the node's resources.

9 Evaluation

In the following, we provide an empirical evaluation of OmniSphinx. We first describe our implementation of OmniSphinx, then evaluate performance in terms of bandwidth (Section 9.1) and computation (Section 9.2).

Implementation We implemented OmniSphinx in Java. For all cryptographic operations, we rely on BouncyCastle[4] v1.70. We implement `Hash` as SHA-256, `Mac` as HMAC-SHA256, `Encrypt` and `Decrypt` as LIONESS [2], and `Prg` as an AES-CTR keystream. As the group $\mathcal{G}$ we use the `secp224r1` NIST elliptic curve. We publish our implementation at https://github.com/kit-ps/OmniSphinx.

9.1 Bandwidth

Our goal is to determine how much bandwidth overhead emulation in OmniSphinx introduces versus the native mix formats. OmniSphinx packets differ from other Sphinx-like packets in the size of their β header, where OmniSphinx includes the mix programs. Within each instance, the size of β is determined by a public protocol parameter, otherwise packets can be linked via their header size. We want to determine the minimum size of β in OmniSphinx to emulate state of the art mix formats and compare the resulting header size to the header size of the native format.

[4] https://www.bouncycastle.org/

We compare OmniSphinx against Sphinx, AE-Sphinx, EROR, MultiSphinx, and PolySphinx. Native header sizes are derived from the respective format's description. For the minimum OmniSphinx β, we implement a reference mix program for each tested format. For MultiSphinx and PolySphinx, we assume a replication factor of $p = 3$. For all experiments, we assume path lengths of 5 and a security parameter $\kappa = 16\,\mathrm{B}$. We present the total header size.

For OmniSphinx, the header has to contain a mix program, whereas it only has to hold the routing information for the native protocols. We thus expect the OmniSphinx header to be larger than the native headers in all cases. We further expect the smallest difference in header size between OmniSphinx and Sphinx, as most of Sphinx's processing is already handled by OmniSphinx's preprocessing, resulting in a short mix program (see Section 7.1).

Mix format	Native	Emulated	Difference
Sphinx	205 B	273 B	+ 68 B + 33%
AE-Sphinx	205 B	465 B	+260 B +127%
EROR	485 B	725 B	+240 B + 49%
MultiSphinx ($p = 3$)	205 B	489 B	+284 B +139%
PolySphinx ($p = 3$)	1256 B	1545 B	+289 B + 23%

Table 2. Minimum header size of OmniSphinx emulations versus existing mix formats.

Table 2 presents our results and confirms our expectations: OmniSphinx headers are between 68 B and 289 B larger than native headers. The smallest difference is observed with Sphinx. Native MultiSphinx, AE-Sphinx and EROR re-use the MAC in the header to secure the integrity of the payload, however in OmniSphinx, this MAC is computed as part of the preprocessing and cannot be influenced. Therefore, those formats have to include an extra MAC in the header, leading to an increased size. Emulated PolySphinx headers have a large absolute increase over native ones, but since the headers are large to begin with, the relative increase is small.

While OmniSphinx produces significantly larger headers compared to native formats, the increase is small in absolute numbers. To emulate all evaluated formats, a β size of 1545 B has to be used for all packets. Then, emulating Sphinx introduces the largest overhead with 1340 B per packet over the native format assuming five hops. For a standard payload size of 2 KiB, this results at worst in an increase of packet size of 61%.

9.2 Computation

Like packet size, the computational overhead of OmniSphinx is dictated by the mix program. We divide the following evaluation into two parts: Our first goal is to evaluate the computational overhead of the base packet format. Then, we want

evaluate the computational overhead of individual OmniSphinx instructions, so that one can gauge the overhead that complex mix programs incur.

We measure package creation time, processing at a intermediate mix node, and at an exit node for OmniSphinx packets. Since the mix node requires *some* mix program to process the packet, we use the Sphinx mix program as described in Section 7.1. This gives us the opportunity to compare OmniSphinx's computational overhead to Sphinx, for which we use the `java-sphinx`[5] implementation. All measurements were obtained on a machine equipped with an AMD Ryzen 5 5625U and 16 GiB of RAM, using the `jmh`[6] benchmarking framework. For all measurements, we use a payload size of 1 KiB and a path length of five hops.

Generally, we expect processing times of OmniSphinx to be higher that those of native Sphinx, due to the additional interpretation of the mix program. We do not expect large differences, as the computationally heavy cryptographic operations are the same in both formats, and in OmniSphinx, they are represented by a single instruction.

Format	Packet Creation	Mix Processing	
		Intermediate Node	Exit Node
Native Sphinx	1.16 ms ± 0.00 ms	198.29 µs ± 0.32 µs	194.17 µs ± 0.28 µs
OmniSphinx	1.12 ms ± 0.00 ms	283.32 µs ± 1.29 µs	280.52 µs ± 1.43 µs

Table 3. Processing times of OmniSphinx versus native Sphinx.

Table 3 presents our results and confirms the expectations: Package creation has the same performance for Sphinx and OmniSphinx, as this part is implemented in native Java for both protocols. For package processing, we see that OmniSphinx is slower by approximately 90 µs. However, the total time needed to process packets is still negligible compared to overheads induced by network latencies.

For the per-instruction cost, we expect all instructions that simply move bytes between registers to be fast. We expect `MAC`, `Hash`, `Encrypt`, `Decrypt`, `Prg`, and `Exponent` to be slower, as those rely on computationally expensive cryptography.

Our results in Table 4 confirm our expectations. All byte-moving operations complete in approximately 1.5 µs. The symmetric primitives `MAC`, `Hash`, `Encrypt`, and `Decrypt` are slower, but only by a factor of 2–3. The slowest operation is `Exponent`, which represents a public-key cryptographic operation.

[5] https://github.com/rsoultanaev/java-sphinx, by Robert Soultanaev
[6] https://openjdk.org/projects/code-tools/jmh/

	Stop	CutBytes	Hash	MAC	Verify	Exponent	Pad	Prg	XOR	Decrypt	Encrypt	Concat	ConcatBytes	Copy	Add	Forward	Load	ForLoop
Average time	1.46 µs ± 0.00 µs	1.54 µs ± 0.01 µs	2.19 µs ± 0.01 µs	3.34 µs ± 0.13 µs	1.43 µs ± 0.01 µs	153.11 µs ± 0.25 µs	1.54 µs ± 0.01 µs	2.02 µs ± 0.01 µs	1.47 µs ± 0.00 µs	4.13 µs ± 0.02 µs	4.26 µs ± 0.03 µs	1.48 µs ± 0.00 µs	1.49 µs ± 0.00 µs	1.48 µs ± 0.01 µs	1.49 µs ± 0.00 µs	1.48 µs ± 0.00 µs	1.52 µs ± 0.01 µs	1.51 µs ± 0.01 µs

Table 4. Execution time per OmniSphinx instruction.

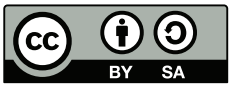
10 Conclusion

In this work, we investigated if techniques from active networking can provide meaningful benefits in the context of mix networks. To investigate this question, we implemented OmniSphinx, a novel mix format, where senders can embed custom mix programs in their packets. The mix programs are executed by each node on the packet's path and determine how the packet is processed. With OmniSphinx, an active mix network can be realized, where mix node operators no longer have to restrict themselves to supporting a single mix format only. As expected, emulation in OmniSphinx incurs higher overhead than native execution of the corresponding mix format, but in typical mix network use cases, such as email communication, the total overhead remains manageable: Sphinx packet processing time increases from approximately 200 µs to approximately 300 µs, and headers grow by 33%. At the same time, we see concrete benefits of active mix networks over classical ones: Clients with different needs regarding the mix network's functionality can share the same instance, resulting in better node utilization for operators and a larger choice of nodes for clients. Additionally, active mix networks may become a valuable tool for research in the future, as new mix formats are easy to implement and to deploy without having to rely on node operators or separate testing networks.

Acknowledgments. This work has in part been funded by the Helmholtz Association through the KASTEL Security Research Labs (HGF Topic 46.23) and by the German Research Foundation (DFG, Deutsche Forschungsgemeinschaft) as part of Germany's Excellence Strategy – EXC 2050/2 – Project ID 390696704 – Cluster of Excellence "Centre for Tactile Internet with Human-in-the-Loop" (CeTI) of Technische Universität Dresden.

This preprint has not undergone peer review (when applicable) or any post-submission improvements or corrections. The Version of Record of this contribution is published in Lecture Notes in Computer Science (Volume 16971), and is available online at https://doi.org/10.1007/978-3-032-38708-0_31.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.



This work is licensed under Creative Commons Attribution-ShareAlike 4.0 International. To view a copy of this license, visit https://creativecommons.org/licenses/by-sa/4.0/

References

1. Alexander, D.S., Arbaugh, W.A., Hicks, M.W., Kakkar, P., Keromytis, A.D., Moore, J.T., Gunter, C.A., Nettles, S.M., Smith, J.M.: The SwitchWare active network architecture. IEEE network (2002)
2. Anderson, R., Biham, E.: Two practical and provably secure block ciphers: BEAR and LION. In: Fast Software Encryption
3. Beato, F., Halunen, K., Mennink, B.: Improving the Sphinx Mix Network. In: Cryptology and Network Security (2016)
4. Bhattacharjee, S., Calvert, K.L., Zegura, E.W.: An architecture for active networking. In: International Conference on High Performance Networking (1997)
5. Chaum, D.L.: Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. Commun. ACM (1981)
6. Danezis, G., Goldberg, I.: Sphinx: A Compact and Provably Secure Mix Format. In: IEEE S&P (2009)
7. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM (1977)
8. Dingledine, R., Mathewson, N., Syverson, P.F.: Tor: The second-generation onion router. In: USENIX Security (2004)
9. Dyer, K.P., Coull, S.E., Shrimpton, T.: Marionette: A programmable network traffic obfuscation system. In: USENIX Security (2015)
10. Feamster, N., Rexford, J., Zegura, E.: The road to SDN: an intellectual history of programmable networks. ACM SIGCOMM Computer Communication Review (2014)
11. Hugenroth, D., Kleppmann, M., Beresford, A.R.: Rollercoaster: An Efficient Group-Multicast Scheme for Mix Networks. In: USENIX Security (2021)
12. Klooß, M., Rupp, A., Schadt, D., Strufe, T., Weis, C.: EROR: Efficient repliable onion routing with strong provable privacy. Cryptology ePrint Archive, Paper 2024/020 (2024), https://eprint.iacr.org/2024/020
13. Kuhn, C., Beck, M., Strufe, T.: Breaking and (Partially) Fixing Provably Secure Onion Routing. In: IEEE S&P (2020)
14. Reininger, M., Arora, A., Herwig, S., Francino, N., Garman, C., Levin, D.: Bento: Bringing network function virtualization to Tor. In: CCS (2020)
15. Reininger, M., Arora, A., Herwig, S., Francino, N., Hurst, J., Garman, C., Levin, D.: Bento: safely bringing network function virtualization to Tor. In: ACM SIGCOMM (2021)
16. Rial, A., Piotrowska, A., Halpin, H.: Outfox: a postquantum packet format for layered mixnets. In: WPES (2025)
17. Rochet, F., Dejaeghere, J., Elahi, T.: Towards flexible anonymous networks. In: WPES (2024)
18. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on selected areas in communications (2003)

19. Schadt, D., Coijanovic, C., Weis, C., Strufe, T.: PolySphinx: Extending the Sphinx Mix Format With Better Multicast Support. In: IEEE S&P (2024)
20. Scherer, P., Weis, C., Strufe, T.: Provable Security for the Onion Routing and Mix Network Packet Format Sphinx
21. Tennenhouse, D.L., Wetherall, D.J.: Towards an active network architecture. ACM SIGCOMM Computer Communication Review (1996)
22. Wails, R., Jansen, R., Johnson, A., Sherr, M.: Proteus: Programmable protocols for censorship circumvention. Free and Open Communications on the Internet (2023)

A Concrete mix programs

In this section, we provide concrete mix programs for the formats described in Section 7. Inputs to the program are embedded by the sender as byte string constants. For Sphinx, we show the relay program in Algorithm 1 and the exit program in Algorithm 2. For PolySphinx, we show the relay program in Algorithm 3, the replication program in Algorithm 4, and the exit program in Algorithm 5.

Algorithm 1 Sphinx relay program

Input: Next node n
// Decrypt payload
`ConcatByte`$(r_s, \texttt{0x01}, r_s)$
`Hash`(r_s, r_h)
`Decrypt`$(r_h, r_\delta, r_\delta)$
// Forward
`Load`(n, r_n)
`Forward`(r_n)
`Stop`

Algorithm 2 Sphinx exit program

// Decrypt payload
`ConcatByte`$(r_s, \texttt{0x01}, r_s)$
`Hash`(r_s, r_h)
`Decrypt`$(r_h, r_\delta, r_\delta)$
// Verify integrity
`CreateZeroes`(κ, r_z)
`CutBytes`(r_δ, κ, r_y)
`IsEqual`(r_z, r_y)
// Deliver message
`CutBytes`$(r_\delta, 2\kappa, r_n)$
`Forward`(r_n)
`Stop`

Algorithm 3 PolySphinx relay program

Input: Next node n, node key σ
`Load`(σ, r_k)
`Encrypt`$(r_k, r_\delta, r_\delta)$
`Load`(n, r_n)
`Forward`(r_n)
`Stop`

Algorithm 4 PolySphinx replication program

Input: Replication factor p, concatenated subheaders B
`Copy`(r_δ, r_Δ)
`Load`(B, r_B)
// Send p replicated packets
`ForLoop`$(7, p)$
  `CutBytes`$(r_B, 2\kappa, r_n)$
  `CutBytes`(r_B, κ, r_K)
  `CutBytes`$(r_B, 2\kappa, r_\alpha)$
  `CutBytes`(r_B, κ, r_γ)
  `CutBytes`$(r_B, \tau_{\text{Post}}, r_\beta)$
  `Encrypt`$(r_K, r_\Delta, r_\delta)$
  `Forward`(r_n)
`Stop`

Algorithm 5 PolySphinx exit program

Input: Replication factor p, key tree seed S, path P, recipient R
`Load`(S, r_S)
`Load`(P, r_P)
`Load`(R, r_R)
// Root of the key tree
`Hash`(r_S, r_σ)
`Hash`(r_σ, r_K)
// Build path through the key tree
`ForLoop`$(4, p)$
  `CutBytes`$(r_P, 1, r_i)$
  `Add`$(r_\sigma, r_i, r_\sigma)$
  `Hash`(r_σ, r_σ)
  `Concat`(r_σ, r_K, r_K)
// Decrypt the payload
`ForLoop`$(2, p+1)$
  `CutBytes`(r_K, κ, r_σ)
  `Decrypt`$(r_\sigma, r_\delta, r_\delta)$
`Forward`(r_R)
`Stop`