Privacy Analysis of Anonymous Communication Networks at the Example of Vuvuzela

Master's Thesis of

Christoph Coijanovic

at the Department of Informatics

Chair of IT-Security

Reviewer	Prof. Dr. Thorsten Strufe
Advisor	Christiane Kuhn, M.Sc.

January 01, 2020 – June 30, 2020

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, June 30, 2020

. . . .

(Christoph Coijanovic)

Abstract

In recent years, many anonymous communication networks have been proposed. Since it is common for papers to introduce their own privacy goals and adversary models, comparing protocols can be difficult. To fix this problem, Kuhn et. al. proposed a framework [1] that formalizes different privacy goals. Protocols which have been analyzed using this framework can be directly compared to each other. To provide more data for such a comparison, we present a formal analysis of the anonymous communication network *Vuvuzela* [2] using this framework. During the analysis, we identify significant weaknesses of Vuvuzela against *replay* and *sender timing* attacks. We propose a variant of Vuvuzela called *f*-Vuvuzela which fixes the discovered weaknesses. Finally, we reveal that these weaknesses are only partially fixed in *Stadium* [3] and *Karaoke* [4], which are follow-up protocols to Vuvuzela.

In den letzten Jahren wurden viele anonyme Kommunikationsnetzwerke vorgestellt. Da in den Veröffentlichungen oft eigene Privatsphäre-Ziele und Angreifermodelle definiert werden, kann es schwer sein, Protokolle miteinander zu vergleichen. Das Framwork von Kuhn et. al. [1] formalisiert verschiedene Privatsphäre-Ziele. Protokolle, die mit diesem Framework analysiert wurde, können direkt miteinander verglichen werden. Wir analysieren das anonyme Kommunikationsnetzwerk *Vuvuzela* [2] mithilfe des Frameworks. In unserer Analyse zeigen wir tiefgreifende Schwachstellen Vuvuzelas hinsichtlich Wiedereinspielung von Paketen und im zeitlichen Verhalten des Senders auf. Wir schlagen eine Variante von Vuvuzela, *f*-Vuvuzela, vor, in der die entdeckten Schwachstellen nicht mehr auftreten. Außerdem zeigen wir, dass die entdeckten Schwachstellen in den Follow-Up Protokollen *Stadium* [3] und *Karaoke* [4] nur teilweise behoben wurden.

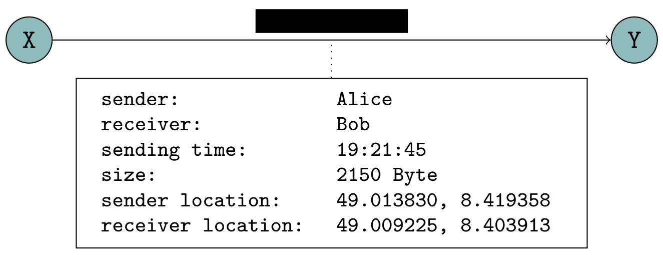
Contents

List of Figures

1. Introduction

Today, the internet plays an integral role in nearly every person's life. Massive amounts of personal data are shared and stored, which makes privacy a vital issue. While today a majority of web traffic is encrypted and transmitted over HTTPS[1], *metadata*, such as the sender, receiver and time of the communication is often still leaked. Refer to Figure 1.1 for an example. To understand why metadata is worth protecting, one can seek advice from the NSA whistleblower Edward Snowden:

> In sum, metadata can tell your surveillant virtually everything they'd ever want or need to know about you [5, p. 141]



```
sender:              Alice
receiver:            Bob
sending time:        19:21:45
size:                2150 Byte
sender location:     49.013830, 8.419358
receiver location:   49.009225, 8.403913
```

Figure 1.1: Even though the actual content of the message is hidden, the observer can still infer a lot of information from the metadata.

This is where *anonymous communication networks (ACNs)* come in: They aim to not only protect the communication's content, but also the metadata. Research into anonymous communication was started by David Chaum in the early 1980s [6]. Today, there is an abundance of proposed and implemented ACNs, including *Tor* [7], *Crowds* [8], *DC-nets* [6] and *Vuvuzela* [2].

If a privacy-focused individual wants to choose the right ACN for their needs, they can quickly be overwhelmed by the multitude of options, many of which appear very similar in their construction. The search is further complicated by the fact that many papers define their own privacy goals and definitions. Kuhn et al. introduce formal notions of privacy and a framework with which different ACNs can be analyzed and fairly compared [1]. This thesis uses their framework to analyze *Vuvuzela*. Vuvuzela in particular was chosen, because it claims to be able to handle 100 times more concurrent users (up to 2 million) compared to previous systems while protecting metadata [2, Sec. 1]. Vuvuzela also serves as the basis of multiple other ACNs, such as *Stadium* [3] and *Karaoke* [4], which promise further performance improvements. Thus, finding weaknesses in Vuvuzela has a comparably large impact on ACNs in general.

The result of a complete analysis is a set of *privacy notions* the ACN is proven to reach. An intuitive example for such a notion is *Message Unobservability*. If the ACN reaches Message Unobservability, the assumed adversary cannot learn the contents of messages send using the ACN. ACNs that have been analyzed with the framework can be objectively compared to each other. For a given use case, a list of necessary privacy notions can be compiled and used to narrow down the possible ACNs. If there are multiple ACNs that satisfy all requirements, additional parameters, such as performance and usability can be compared. These parameters are not part of the framework and therefore only play a subordinate role in this thesis.

[1]See Google Transparency Report: `https://transparencyreport.google.com/https/overview`

In this thesis, we analyze both parts of Vuvuzela, the *Vuvuzela Communication Protocol* and the *Vuvuzela Dialing Protocol*, under different assumptions. We find that the communication protocol only reaches *Communication Unobservability* under the strongest assumption. Under weaker and arguably more realistic assumptions, weaknesses regarding the replay of request and sender timing can be exploited to break nearly all privacy notions. We introduce a fixed variant of Vuvuzela, f-Vuvuzela, which is shown to reach *Communication Unobservability* under weaker assumptions. Lastly, we show that Stadium and Karaoke inherit some of Vuvuzela's weaknesses.

Outline. Chapter 2 introduces the framework, adversary model and Vuvuzela itself. We introduce assumptions needed for the analysis in Chapter 3. In Chapter 4, we analyze the Vuvuzela Communication Protocol, followed by the Vuvuzela Dialing Protocol in Chapter 5. Chapter 6 presents an additional analysis of Vuvuzela allowing greater variation in communication patterns between scenario than is assumed in [2]. Chapter 7 examines how privacy protection is impacted by combining Vuvuzela with *Alpenhorn*, an improved dialing protocol by the same authors. In Chapter 8, we introduce and analyze f-Vuvuzela which fixes the discovered weaknesses of Vuvuzela. Chapter 9 examines if the amount of server cover traffic in Vuvuzela can be reduced by changing the way clients exchange their requests. In Chapter 10, we take a look a protocols derived from Vuvuzela to determine if they are still susceptible to the same weaknesses. Chapter 11 contains a discussion of the fundamentals of Vuvuzela and our findings, Chapter 12 summarizes and concludes. We end by presenting possible future work in Chapter 13.

2. Background

In the following sections, we will provide the necessary background information for the analysis of Vuvuzela. We will start by explaining the framework of [1] in Section 2.1 and our adversary model in Section 2.2. Vuvuzela is split into two independent protocols: The *Vuvuzela Communication Protocol* (VCP), which we explain in Section 2.3 and the *Vuvuzela Dialing Protocol* (VDP), which we explain in Section 2.4.

2.1. The Framework

[1] introduces a game model that is very similar to the IND-CPA game used in formal security proofs of cryptographic systems. An adversary $\mathcal{A}$ has to submit a *challenge* consisting of two *scenarios* $\underline{r}_0$ and $\underline{r}_1$ to the challenger $\mathcal{C}$. A scenario consists of multiple *communications*. We represent each communications as a tuple $(sender, receiver, message, aux)$, where aux refers to auxiliary information, such as session identifiers. $\mathcal{C}$ then picks one of those scenarios by choosing a single bit b uniformly at random and submitting $\underline{r}_b$ to the ACN Π, which executes it. $\mathcal{A}$ receives all output from Π and has to decide which of his scenarios was submitted to Π by $\mathcal{C}$. He returns his guess b' to $\mathcal{C}$. $\mathcal{A}$ wins if $b' = b$ and loses otherwise. An overview of this game can be found in Figure 2.1.

This game then can be used to prove that a given ACN fulfills a given *privacy notion*. As an example, a notion an ACN could have is *Sender-Message Unlinkability* ($SM\overline{L}$). If an ACN achieves this notion, attackers cannot distinguish which message was send by which sender[1]. To model this notion with the game, we restrict the adversary to submit scenarios that only differ in the senders. Everything else (e.g., message content, the number of send messages and which sender sends how often, the receiver and the auxiliary information) has to be the same (see Fig. 2.2).

Since the adversary only has to output a single bit, he has a 50% probability of winning by guessing

[1]He might still be able to read the content of the messages or find out who the message was send to, but the identity of the sender remains hidden.

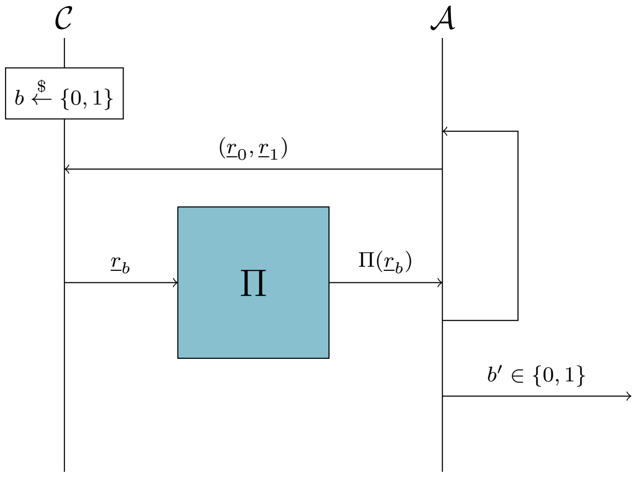


Figure 2.1: The formalized security game. $\mathcal{A}$ wins if $b' = b$.

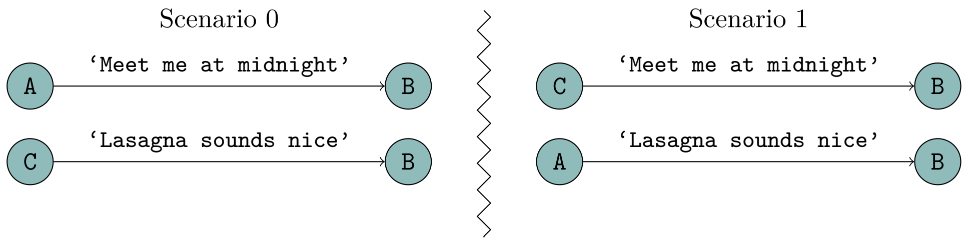


Figure 2.2: An example for an allowed challenge for Sender-Message Unlinkability

b' at random. The ACN fulfills the notion X for a given adversary $\mathcal{A}$, if $\mathcal{A}$'s probability of winning the corresponding game is $50\% + \varepsilon$, where ε is *non-negligible*.

2.1.1 Privacy Notions

In the following section, we will give an overview of the privacy notions that are relevant to our analysis of Vuvuzela. For a full list of notions and formal definitions, see [1].

The strongest possible notion for an ACN to reach is *Communication Unobservability* ($C\overline{O}$). If $C\overline{O}$ is reached, the ACN leaks *no* information about the communications. Thus, the adversary is not restricted in the scenarios he can choose, since all possible scenarios will look the same to him.

If a user wants his message content to be confidential, the ACN should at least reach the notion *Message Unobservability* ($M\overline{O}$). To break this notion, the adversary needs to be able to distinguish scenarios only by message content. Thus, all other variables (e.g., senders and receivers) need to be identical in both scenarios. Some ACNs leak information about the length of send messages. In that case, the adversary could win the $M\overline{O}$-game without actually distinguishing the content. To solve this issue, $M\overline{O}$ can be restricted further to $M\overline{O} - |m|$. Here, messages have to be of the same length in both scenarios.

Similarly to $M\overline{O}$, *Sender Unobservability* ($S\overline{O}$) can be defined. Here, the scenarios may only differ in the senders of the messages, the messages themselves and the receivers have to be equal in both scenarios. If $S\overline{O}$ is reached, the ACN does not allow the adversary to find out who the sender of a given message is.

$S\overline{O}$ can be further restricted by the *Message Partitioning per Sender* (P) and *Sender Frequency Histogram* (H) properties:

- P: Which messages are send from the same sender is equal in both scenarios
- H: How many senders send how often is equal in both scenarios

Receiver Unobservability ($R\overline{O}$) is defined analogously.

If the combination of sender and message content should remain confidential, *Sender-Message Pair Unobservability* ($(SM)\overline{O}$) has to be reached. To model this requirement, it is not enough to restrict the adversary to only changing sender and message between the scenarios, since this would also allow adversaries to win who could only identify either senders *or* messages. Instead, the concept of *scenario instances* is introduced: The adversary has to provide an alternative instance for each scenario, the challenger will choose one instance of one scenario at random and submit it to the ACN. While the bit choosing the scenario is picked once for each game, the instance bit is picked randomly for each challenge. For Sender-Message Pair Unobservability, the adversary can freely choose two sender message pairs as instance 0 and 1 in the first scenario. The instances of the second scenario must consist of a mixed variant of those pairs (see Figure 2.3).

Receiver-Message Pair Unobservability ($(RM)\overline{O}$) and *Sender-Receiver Pair Unobservability* ($(SR)\overline{O}$) are defined analogously.

Similarly, *Sender-Message Pair Unlinkability* ($(SM)\overline{L}$) can be defined: Here the ACN can leak

Message Unobservability

Scenario 0	Scenario 1
$(alice, bob, \mathbf{m_1}, aux)$	$(alice, bob, \mathbf{m_2}, aux)$

Sender Unobservability

Scenario 0	Scenario 1
$(\textbf{alice}, bob, m, aux)$	$(\textbf{carol}, bob, m, aux)$

Sender-Message Pair Unobservability

	Scenario 0	Scenario 1
I_0	$(alice, bob, \mathbf{m_1}, aux)$	$(alice, bob, \mathbf{m_2}, aux)$
I_1	$(carol, bob, \mathbf{m_2}, aux)$	$(carol, bob, \mathbf{m_1}, aux)$

Sender-Message Pair Unlinkability

	Scenario 0	Scenario 1
I_0	$(alice, bob, \mathbf{m_1}, aux)$	$(alice, bob, \mathbf{m_2}, aux)$
	$(carol, bob, \mathbf{m_2}, aux)$	$(carol, bob, \mathbf{m_1}, aux)$
I_1	$(carol, bob, \mathbf{m_2}, aux)$	$(carol, bob, \mathbf{m_1}, aux)$
	$(alice, bob, \mathbf{m_1}, aux)$	$(alice, bob, \mathbf{m_2}, aux)$

Twice-Sender Unlinkability

		Scenario 0	Scenario 1
I_0	r_0	$(alice, bob, m, aux)$	$(alice, bob, m, aux)$
	r_1	$(\textbf{alice}, bob, m, aux)$	$(\textbf{carol}, bob, m, aux)$
I_1	r_0	$(carol, bob, m, aux)$	$(carol, bob, m, aux)$
	r_1	$(\textbf{carol}, bob, m, aux)$	$(\textbf{alice}, bob, m, aux)$

Figure 2.3: Example challenges for selected notions. I_x denotes instance x, r_x denotes round x.

which senders are active and which messages are send, but not the link between them, i.e. who sends which message.

Lastly, if the fact that the same sender sends multiple messages needs to be protected, the ACN needs to reach *Twice-Sender Unlinkability* ($(2S)\overline{L}$). Here, the adversary has to submit one scenario where the same sender is active in two succeeding rounds and another scenario where a different sender is active in the second round[2].

Example challenges for the described notions can be found in Figure 2.3.

2.2. Adversary Model

When making statements about the level of protection that an ACN provides, one has to specify what kind of adversary is considered. Imagine a network where messages are send from sender to receiver through a chain of multiple servers. To ensure message confidentiality, the sender encrypts the message content with the first server's public key and sends the ciphertext to the server. The server decrypts it, re-encrypts it with the public key of the next server, forwards it and so on. In such a network, the content of messages would be hidden to an adversary that can only passively observe the traffic on the network links, since only encrypted messages are sent. But as soon as the adversary is able to *corrupt* one of the servers in the chain (e.g., by acquiring its secret key), he can find out the message's plaintext.

We formalize our adversary model and introduce different classes of adversaries based on [9], which proposes four main abilities an attacker can have:

- **Passive Observation** (G): The adversary can observe the activity of nodes and on links. He can do so either globally (on all nodes/links) or restricted to a subset of nodes/links.

[2]and another instance, where the roles of the senders are switched

$$\begin{tikzcd}
 & \mathcal{A}_{GCTM} \arrow[dl] \arrow[d] \arrow[dd, bend left] \arrow[dr] & \\
\mathcal{A}_{CM} \arrow[d] & \mathcal{A}_{CT} \arrow[dl] & \mathcal{A}_{GNM} \arrow[d] \\
\mathcal{A}_{C} & \mathcal{A}_{GT} & \mathcal{A}_{GN}
\end{tikzcd}$$

Figure 2.4: Hierarchy of relevant adversaries.

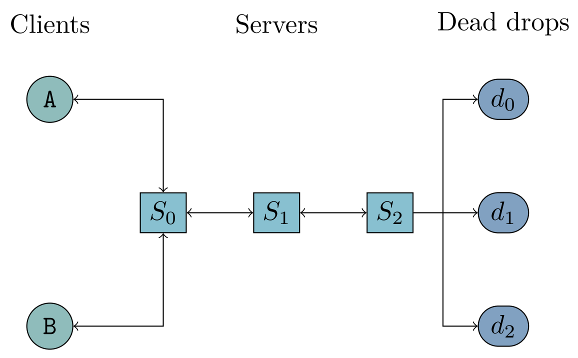


Figure 2.5: Vuvuzela network diagram

In our analysis, we always assume that passive observation can be done globally, since this matches Vuvuzela's adversary model.

- **Active Corruption** (C): The adversary can corrupt a defined number of links and nodes.
- **Timing** (T): The adversary has access to a timer which he can use for passive traffic analysis.
- **Traffic Modification** (M): The adversary can actively modify the network traffic, i.e., insert, delete or modify packets.

A concrete adversary can be created by combining the abilities above. For example, $\mathcal{A}_{GCM}$ denotes an attacker that can observe the network, actively control (a part of) the network and modify the traffic, but cannot delay packets since no timer is available.

Figure 2.4 (adapted from [9]) shows all adversaries that appear during our analysis and how they are connected to each other.

2.3. Vuvuzela Communication Protocol

Network Structure. The Vuvuzela network consists of a series of servers connected in one chain. All clients send requests to the first of these server. The last server in the chain has access to a number of *dead drops*, which can be thought of as mailboxes. Dead drops are used by the clients to exchange messages with each other. Figure 2.5 shows a diagram of the network structure.

Cryptographic Keys. In Vuvuzela, each client and each server have a long-lived pair of keys: One is their secret key *sk* and the other one is a public key *pk*.

Vuvuzela relies on symmetric encryption, which needs a shared key for encryption and decryption. To compute shared keys, *Elliptic-Curve Diffie-Hellmann* (ECDH) is used. We present an informal definition:

Definition 1 (Diffie-Hellmann key exchange)**.** Let Alice be a client with access to a key pair (pk_a, sk_a) and Bob be a client with access to the key pair (pk_b, sk_b). pk_a is known to Bob and pk_b is known to Alice. Alice and Bob can use the Diffie-Hellmann function DH to compute a shared key k_{ab}:

$$\underbrace{DH(pk_b, sk_a)}_{\text{computed by Alice}} = k_{ab} = \underbrace{DH(pk_a, sk_b)}_{\text{computed by Bob}}$$

Exchanging Messages. If Alice and Bob want to communicate with each other, they send their requests to the same dead drop. Clients cannot send requests any time they like: Communication occurs in rounds and clients send their requests after the first server announced the start of a new round. Alice and Bob choose a new dead drop d_{ab}^r each round by hashing their shared secret k_{ab} and the current round number r:

$$d_{ab}^r = H(k_{ab} \mid r)$$

If Alice wants to send the message m_1 to Bob in round r, she first encrypts it using their shared secret k_{ab} and combines it with the current dead drop id d_{ab}^r for them to form a request:

$$req := (d_{ab}^r, Enc_{k_{ab}}(m_1))$$

She then *onion-encrypts* (see Figure 2.6) the request with one encryption layer using a random temporary key for each server. The finished request is sent to the first server. Bob has also sent a request containing dead drop id d_{ab}^r.

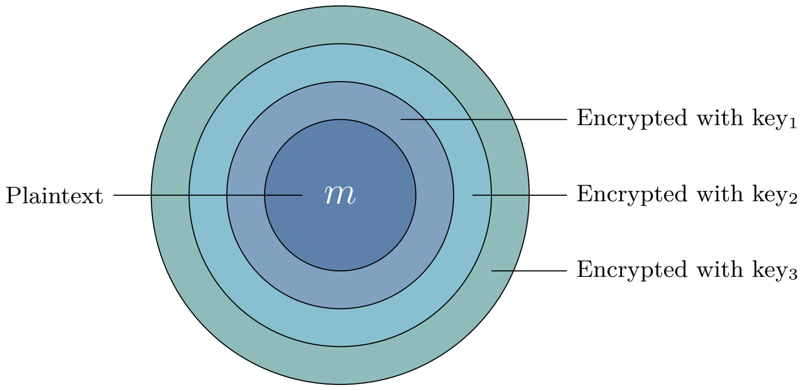


Figure 2.6: An illustration of onion encryption. The plaintext m is recursively encrypted using key$_1$ to key$_3$. With Vuvuzela, clients add one encryption layer per server. The encyption key k_{ai} for each layer is computed with the Diffie-Hellman function DH: The client chooses a random key pair (pk_i^t, sk_i^t) per server and computes $k_{ai} \leftarrow DH(sk_i^t, pk_i)$, where pk_i is the server's long-lived public key. pk_i^t is then appended to the request, so that the server can also compute $k_{ai} \leftarrow DH(sk_i, pk_i^t)$ to decrypt the onion layer. The resulting ciphertext has multiple layers akin to an onion.

Server Behavior. The Vuvuzela server chain implements a *mixnet*: The inner (i.e., all but the last) servers unpack the outer onion layer from every request, add random cover requests, shuffle the requests and send them to the next server. The goal of this procedure is to prevent the adversary from being able to link requests from before the server to requests after it. If this were not the case, the adversary could link senders to receivers by following the request's path through the network.

The last server performs the dead drop exchanges: If two requests contain the same dead drop id, their ciphertexts get exchanged. The modified requests are then send back through the chain to the respective clients, who can unpack them to find their communication partner's message.

Privacy Goal. Vuvuzela aims to provide *differential privacy.* That is, all possible communication patterns of a given client Alice (e.g., "Alice is talking to Bob", "Alice is talking to Carol", "Alice is not talking at all") look about the same for a observing adversary. For a more formal definition, see Section 15.1.

Cover Requests. What happens if Alice and Bob are communicating, but Alice does not have any message to send in the current round? It is not sensible for her to send no request, since this would reveal the fact that she is not sending a request in the current round (and it would also reveal when she is actually sending requests). Instead, she still sends a request to the shared dead drop with Bob containing a message of only 0-bits as cover. If she is not in active conversation with any other clients, she sends a cover request to a random receiver. More formally, we define the following types of requests:

Definition 2 (Request Types)**.** Vuvuzela differentiates between three types of requests:

1. **M**-Type (Message): A request containing a real message
2. **AC**-Type (Active Communication): A cover request to a real receiver (sent by a sender in active communication)
3. **IC**-Type (Inactive Communication): A cover request to a random receiver (sent by a sender not in active communication)

Multiple Conversations. If clients are allowed to participate in multiple conversations per round, adjustments have to be made:

Now, each client has an internal parameter ν, which indicates the maximum number of conversations he can have. To hide his real activity, he has to send ν requests each round. He sends as many **M**-Type request as he has messages to send. The rest of his active conversation partners receive an **AC**-Type request. If he is in less than ν active conversation, he sends **IC**-Type requests to fill the difference.

Pseudocode for clients, inner servers and the last server can be found in Algorithms 5 to 7 (Section 15.2).

Adversary Model. Vuvuzela aims to protect against an adversary $\mathcal{A}$ who can [2, Sec. 2.3]:

- corrupt and control all but one Vuvuzela server
- control an arbitrary number of clients
- monitor, block, delay and inject traffic on any network link.

Importantly, Vuvuzela only protects communications where both sender and receiver are uncompromised. Thus, $\mathcal{A}$ is not allowed to corrupt challenge clients in our analysis. In the notation introduced in Section 2.2, this adversary is equivalent to $\mathcal{A}_{GCTM}$, restricted to not corrupting challenge clients or all servers.

2.4. Vuvuzela Dialing Protocol

The Vuvuzela dialing protocol is a separate protocol from the Vuvuzela communication protocol. Its purpose is to enable users to start new conversation and to restart previous communications.

The network structure is identical to the communication protocol: There is one chain of servers, clients send their requests to the first server and the last server has access to a number of *invitation dead drops.*

Invitation Dead Drops. Unlike the communication protocol, each client has a publicly known invitation dead drop. There are a total of μ invitation dead drops, where μ is set by the last server at regular intervals based one the current load. A client with public key pk_x uses invitation dead drop d, iff

$$d = H(pk_x) \mod \mu$$

Note that μ is supposed to be (much) smaller than the total number of clients n. Thus, multiple clients share the same invitation dead drop.

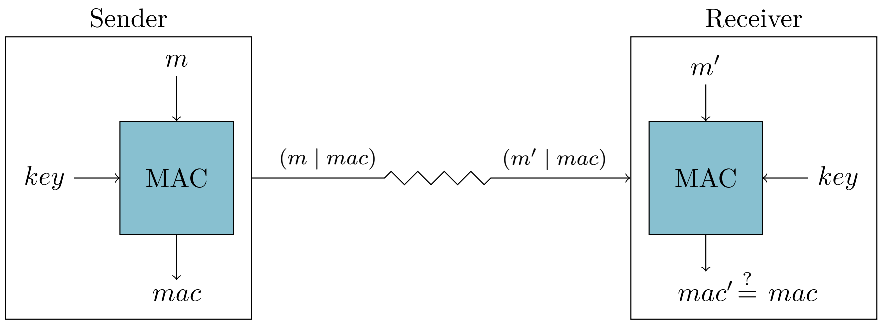


Figure 2.7: Message Authentication Codes. The message is authentic (i.e., $m = m'$), iff $mac = mac'$.

Inviting Clients. If Alice wants to invite Bob to communicate with her, she computes his dead drop id $d = H(pk_{Bob}) \mod \mu$. She then encrypts her public key, a nonce and a *message authentication code* (MAC) (see Figure 2.7) under Bob's public key and combines the resulting ciphertext with d to form a request req:

$$req := (d, Enc_{pk_{Bob}}(pk_{Alice}, \text{nonce}, \text{MAC}))$$

As with the communication protocol, the request is then onion-wrapped and send to the first server. From there, it is passed on down the chain. The last server adds the request (and all other matching requests) to dead drop d.

To receive requests, Bob does not use the server chain, but rather accesses the dead drop d directly. He downloads all requests from the dead drop and tries to decrypt each one with his secret key. If the resulting plain text has the right format (pk, nonce, MAC), he recognizes that this request was meant for him. He adds the request to a list of valid invitations and can start the conversation (based on the contained public key) any time he likes. If the plain text does not have the right format (i.e., the MAC is not valid), he discards the request. An overview of the invitation exchange can be found in Figure 2.8.

Cover Requests. As with the communication protocol, clients who do not have an invitation to send in the current round need to send a cover invitation. The cover request is *not* send to a random dead drop (which is the case in the communication protocol), but rather to a special *no-op* dead drop $\bar{d}$. Analogously to the communication protocol, we define different types of requests:

Definition 3 (Types of Requests)**.** The Vuvuzela Dialing protocol differentiates between two types of requests:

1. **I**-Type (Invitation): A request sent to a real client invitation dead drop
2. **C**-Type (Cover): A cover request sent to the no-op dead drop

Server Behavior. The servers behave very similar to the communication protocol with the only major difference being the generation of cover traffic. Here, each honest server (including the last) adds a random amount of cover requests to *each* dead drop. What is omitted is the handling of requests on the way back to the clients since the clients fetch their requests directly from the dead drops.

Pseudocode for clients, inner servers and the last server can be found in Algorithms 8 to 10 (Section 15.3).

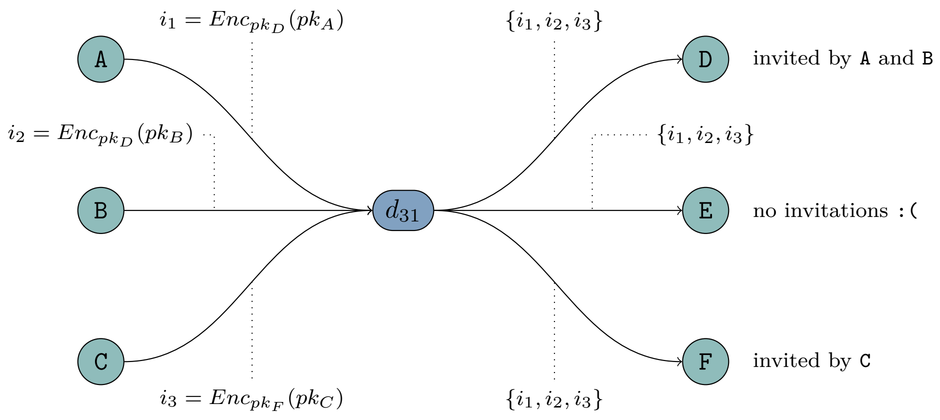


Figure 2.8: VDP invitation exchange. Clients `D`, `E` and `F` share dead drop d_{31}.

3. Assumptions

To analyze the Vuvuzela protocols, we have to make a number of assumptions. A formal analysis of a protocol requires the protocol to be defined in great detail. If the source paper does not provide sufficient details at any point, assumptions have to be made.

To ensure a meaningful result, the assumptions should be as minimal as possible. Our goal for each assumption is to either represent the intend of the original authors as close as possible or, if the intend cannot be inferred, to make assumptions that are common in related literature.

We start by listing a number of general assumptions that are applicable for both VCP and VDP in Section 3.1, then we present additional assumptions for VCP in Section 3.2 and for VDP in Section 3.3.

3.1. General Assumptions

Assumption 1 (Connected Clients). *We assume that the clients connected to the network are identical in both scenarios while the adversary is observing the protocol execution.*

This is reasonable, since [2, Sec. 2.2] already recommends leaving the Vuvuzela client running at all times and states that disconnecting from the network can leak information. Further, allowing differences in connected clients between scenarios would allow the adversary to break any notion trivially, since Vuvuzela does not hide protocol participation.

Assumption 2 (Rounds and Batches). *We assume that all communications an adversary submits in one batch occur in the same round.*

This is reasonable, since Vuvuzela already works in fixed communication rounds, which can be seen as equivalent to batches.

[2, Sec. 2.3] mentions that the authors "assume secure public and symmetric key encryption, key-exchange mechanisms, signature schemes and hash functions". This is not specific enough for our use case. Thus, we make two assumptions regarding the encryption scheme that is used:

Assumption 3 (Encryption Schemes I). *We assume that all used encryption schemes are* IND-CPA*-secure (see Figure 3.1).*

Assumption 4 (Encryption Schemes II). *We further assume that all used encryption schemes have the property of* diffusion*.*

Diffusion was introduced in [10], we present an informal definition:

Definition 4 (Diffusion). An encryption scheme has the property of diffusion, if changing one arbitrary bit of a ciphertext leads to changes in 50% of bits (at random positions) in the resulting plaintext.

We can reasonably make Assumptions 3 and 4 since there are multiple efficient encryption schemes available that fulfill both properties. The Vuvuzela prototype implementation uses the package `box` from the `go` variant of the `NaCl` library[1]. This package implements the stream cipher *XSalsa20*

[1]see `client.go`, line 230 and `onion.go`, lines 9 and 28

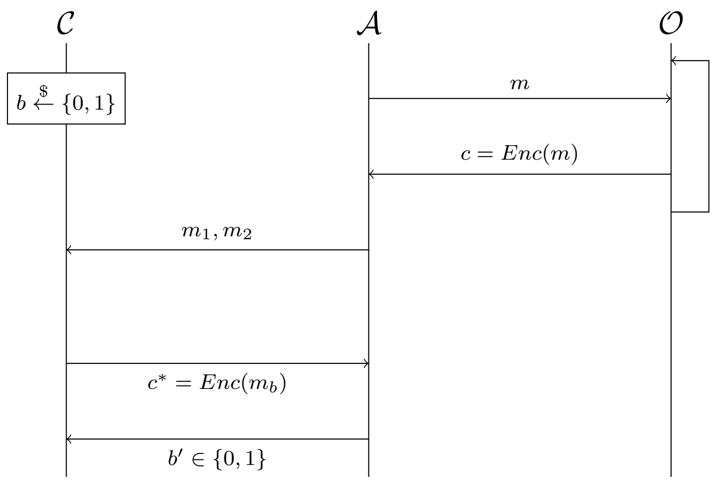


Figure 3.1: The IND-CPA security game. $\mathcal{A}$ wins if $b' = b$. The encryption scheme is IND-CPA secure if no efficient adversary has a non-negligible advantage of winning over random guessing.

for encryption. Related literature [11], [12] suggests that *XSalsa20* fulfills our assumptions.

Vuvuzela's notion of *differential privacy* assumes only one changing communication between scenarios: The adversary cannot tell if Alice is talking to Bob or Dave, as long as all other communications are the same in both cases.

This is quite restrictive for the adversary and may not be very realistic in the real world, since communication patterns are rarely static. Moreover, this restriction would prevent the adversary from submitting valid challenges in certain privacy notions, such as $(SR)\overline{L}$. With other notions (e.g., $S\overline{O}-P$), the adversary would be restricted to submitting challenges where both scenarios are identical. If both scenarios have to be identical, no adversary can break the notion, independent of the protocol in question.

To avoid loosing these notions, we can relax Vuvuzela's assumption and allow differences in the communication pattern of two users:

Assumption 5 (Scenario Variation)**.** *We assume that the adversary can only submit challenges where the scenarios differ in the communication pattern of* two *users.*

We can reasonably make this relaxed assumption, since it only requires a doubling of the mean number of cover requests each inner server adds. We go into more detail on how the server cover traffic scales with growing scenario variation in Section 8.3.

If Assumption 5 is made, one can also reasonably make the following assumption:

Assumption 6 (Server Cover)**.** *We assume that the addition of cover traffic by the servers is parameterized in a way that hides the dead drop access patterns sufficiently and is not computationally removable after as many rounds as the adversary gets to observe.*

This was the intend of the authors of Vuvuzela and depends on the right parameters of the Laplace distribution from which the number of added cover requests is chosen.

Having the scenarios only varying in the communication patterns of two users is still quite restrictive. To show the consequences of further relaxation of Assumption 5, we analyze Vuvuzela separately allowing for arbitrary variation between scenarios (see Chapter 6). In that case, we also discard Assumption 6, since scaling the server cover traffic to account for arbitrary changes would cause overwhelming overhead.

Assumption 7 (Package Layout)**.** *We assume that the packages send over network links only contain the request ciphertext.*

This is an reasonable assumption to make, since the path for each request is predetermined and unambiguous. There is no need for addresses or other information related to routing.

With Assumption 7, we can show the effectiveness of the mixnet:

Corollary 1 (Request Unlinkability)**.** *It follow with Assumption 7 that requests from after a honest inner server cannot be linked to requests from prior to it.*

The honest inner server removes the outermost encryption layer for each request and shuffles all requests with a random permutation. Since the encryption is IND-CPA secure (Assumption 3), $\mathcal{A}$ cannot link requests based on their content. Since the packages only consist of the ciphertext (Assumption 7), $\mathcal{A}$ cannot link them based on other package contents (e.g., IP headers) that are not effected by the decryption.

3.2. Assumptions for the Vuvuzela Communication Protocol

Assumption 8 (Communication Slots)**.** *We assume that the maximum number of active connections a client can have (ν) has to be equal in both scenarios.*

In the real world, ν would be set by the user in the client-software. It is reasonable to assume that this value is chosen during the setup process and rarely (if ever) changed after that. Thus, it should not differ between the scenarios.

In the following, we introduce three different ways of handling active conversations during the challenge-games.

Assumption 9 (Weak Client Cover)**.** *We assume that only clients that send a message in the current batch and their receiver are in active conversation.*

Assumption 10 (Medium Client Cover)**.** *We assume that in each scenario, the set of clients in active conversation is constant.*

Assumption 11 (Strong Client Cover)**.** *We assume that the set of clients in active conversations is equal in both scenarios.*

Example communications under the these assumptions can be found in Figure 3.2. These assumptions in particular where chosen since Assumption 9 is the weakest possible assumption regarding active conversations (clients drop out of active conversation as soon as they don't have a message to send) and Assumption 11 is the strongest possible assumption (all client that exchange messages at some point are always in active conversation with each other). Assumption 10 is a compromise between the other two.

Assumption 9 is arguably what the authors of Vuvuzela had in mind:

> [. . .] all user actions (both real and any possible "cover stories") will look about the same to an adversary. This covers all information that an adversary might learn about Alice's communications: not only whether she's talking to Bob, but whether she's communicating at all (or just running an idle client) [2, Sec. 2.2]

We analyze Vuvuzela under Assumptions 9 to 11 separately.

We make two assumptions regarding sender timing:

Assumption 12 (Sender Timing I)**.** *We assume that the time a client needs to add the padding bits is negligible and cannot be timed by an adversary.*

Assumption 13 (Sender Timing II)**.** *We also assume that the time needed to choose a random key pair when generating a cover request is* not *negligible.*

Setting the contents of a register to 0 can be done in one CPU cycle [13], choosing a valid random public key should take significantly longer.

submitted by $\mathcal{A}$

	Scenario 0	**Scenario 1**
r_0	$(\mathbf{alice}, bob, m_1, aux)$	$(\mathbf{carol}, bob, m_1, aux)$
r_1	$(\mathbf{dave}, bob, m_2, aux)$	$(\mathbf{carol}, bob, m_2, aux)$

Weak Client Cover Assumption

	Scenario 0	**Scenario 1**
r_0	$(\mathbf{alice}, bob, m_1, aux)$	$(\mathbf{carol}, bob, m_1, aux)$
	$(bob, alice, 0, aux)$	$(bob, carol, 0, aux)$
	$(carol, rand, 0, aux)$	$(alice, rand, 0, aux)$
	$(dave, rand, 0, aux)$	$(dave, rand, 0, aux)$
r_1	$(\mathbf{dave}, bob, m_2, aux)$	$(\mathbf{carol}, bob, m_2, aux)$
	$(bob, dave, 0, aux)$	$(bob, carol, 0, aux)$
	$(carol, rand, 0, aux)$	$(dave, rand, 0, aux)$
	$(alice, rand, 0, aux)$	$(alice, rand, 0, aux)$

Medium Client Cover Assumption

	Scenario 0	**Scenario 1**
r_0	$(\mathbf{alice}, bob, m_1, aux)$	$(\mathbf{carol}, bob, m_1, aux)$
	$(bob, alice, 0, aux)$	$(bob, carol, 0, aux)$
	$(dave, bob, 0, aux)$	$(bob, rand, 0, aux)$
	$(bob, dave, 0, aux)$	$(dave, rand, 0, aux)$
	$(carol, rand, 0, aux)$	$(alice, rand, 0, aux)$
r_1	$(\mathbf{dave}, bob, m_2, aux)$	$(\mathbf{carol}, bob, m_2, aux)$
	$(bob, dave, 0, aux)$	$(bob, carol, 0, aux)$
	$(alice, bob, 0, aux)$	$(bob, rand, 0, aux)$
	$(bob, alice, 0, aux)$	$(dave, rand, 0, aux)$
	$(carol, rand, 0, aux)$	$(alice, rand, 0, aux)$

Strong Client Cover Assumption

	Scenario 0	**Scenario 1**
r_0	$(\mathbf{alice}, bob, m_1, aux)$	$(\mathbf{carol}, bob, m_1, aux)$
	$(bob, alice, 0, aux)$	$(bob, carol, 0, aux)$
	$(carol, bob, 0, aux)$	$(alice, bob, 0, aux)$
	$(bob, carol, 0, aux)$	$(bob, alice, 0, aux)$
	$(dave, bob, 0, aux)$	$(dave, bob, 0, aux)$
	$(bob, dave, 0, aux)$	$(bob, dave, 0, aux)$
r_1	$(\mathbf{dave}, bob, m_2, aux)$	$(\mathbf{carol}, bob, m_2, aux)$
	$(bob, dave, 0, aux)$	$(bob, carol, 0, aux)$
	$(carol, bob, 0, aux)$	$(alice, bob, 0, aux)$
	$(bob, carol, 0, aux)$	$(bob, alice, 0, aux)$
	$(alice, bob, 0, aux)$	$(dave, bob, 0, aux)$
	$(bob, alice, 0, aux)$	$(bob, dave, 0, aux)$

Figure 3.2: Examples for the different client cover assumptions. r_x denotes round x. Black communications are **M**-Type requests, gray communications are **AC**-Type requests and **IC**-Type requests are light gray. Bob sends 2 requests in scenario 1 under the medium client cover assumption, because the number of available "communication slots" ν has to be equal in both scenarios by Assumption 8.

Assumption 14 (Receiver Reaction)**.** *We assume that the receiver will not react to a message in any observable way outside of the protocol.*

Otherwise, the receiver could be identified in a way that the protocol has no possibility to protect against. For example, if a receiver reacts to the message “turn your lights off” by turning his lights off, the adversary could observe that change in real life and identify the receiver. No protocol can protect against such attacks.

3.3. Assumptions for the Vuvuzela Dialing Protocol

Assumption 15 (Communication Slots)**.** *We assume that the maximum number of invitations a client can send per round (ν) has to be equal in both scenarios.*

In the real world, ν would be set by the user in the client-software. It is reasonable to assume that this value is chosen during the setup process and rarely (if ever) changed after that. Thus, it should not differ between the scenarios.

Assumption 16 (Client Cover)**.** *We assume that the clients that are participating in the protocol are equal in both scenarios.*

If Alice invites Bob in scenario 0 and Carol invites Bob in scenario 1, Carol would send a cover invitation in scenario 0 and Alice would send cover in scenario 1. Assumption 16 is equivalent to Assumption 11 for the communication protocol. There is no need to relax it, since serious attacks against the dialing protocol are possible as is.

4. Analysis of the Vuvuzela Communication Protocol

We start the analysis by presenting some auxiliary results in Section 4.1, which will simplify later proofs. After that, we analyze the Vuvuzela communication protocol under the weak client cover assumption in Section 4.2, under the medium client cover assumption in Section 4.3 and under the strong client cover assumption in Section 4.4. In addition to that, we analyze VCP under the assumption that challenge clients can be corrupted (Section 4.5.1) and under the assumption that the used encryption scheme is deterministic (Section 4.5.2). We summarize our results in Section 4.6. Finally, we discuss VCP and our results in Section 4.7.

4.1. Auxiliaries

Prior to the full analysis, we show some auxiliary results in this section, which we will rely on in the following proofs and attacks. These results can be split into two categories: First, we examine if request types are indistinguishable to the adversary in Section 4.1.1. After that, we present a way to directly link senders to receivers in Section 4.1.2.

4.1.1 Request Type Determination

To avoid leaking information about client activity, it is important that cover requests are indistinguishable from real requests. In this section, we show that this is only partly the case for Vuvuzela: While **M**-Type requests are indistinguishable from **AC**-Type requests to any adversary without the ability to corrupt challenge clients (Theorem 1), **IC**-Type requests can be distinguished either by timing (Theorem 2) or by replay (Theorem 3). Lastly, we show that replay and timing are the only ways to distinguish **IC**-Type requests in Theorem 4. Figure 4.1 shows an overview of our results.

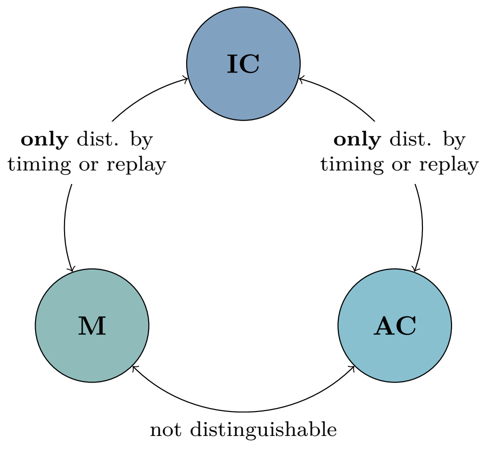

IC
M
AC
only dist. by timing or replay
only dist. by timing or replay
not distinguishable

Figure 4.1: Request type distinguishability in VCP. In all cases, it is assumed that the adversary is not able to corrupt challenge clients.

Our first big result is Theorem 1, which states that an adversary $\mathcal{A}$ without the ability to corrupt clients cannot distinguish if a given request is **M**-Type or **AC**-Type. Intuitively, this is the case because **M**-Type and **AC**-Type requests *only* differ in their content: Where the **M**-Type request contains a "real" message, the **AC**-Type request contains a string of 0s. Since the content of a request is at any point outside of sender and receiver always protected by at least one IND-CPA secure encryption layer, we can show that an adversary $\mathcal{A}$ that is able to distinguish between **M**-Type and **AC**-Type can be used to win the IND-CPA game.

Theorem 1 (**M/AC**-Type Indistinguishably)**.** *$\mathcal{A}$ cannot distinguish if a given request is **M**-Type or **AC**-Type without the ability to corrupt challenge clients.*

Proof. First, we note that **M**-Type and **AC**-Type requests *only* differ in their content: Where the **M**-Type request includes a "real" message, the **AC**-Type request includes $\mathbf{0}$. The handling of the requests is identical: The request is send through the mixnet to the last server, where it is deposited in a pseudo-random dead drop. This dead drop also contains the communication partner's request. The two request get their content exchanged and are send back to the clients through the server chain. Thus, we can argue that if $\mathcal{A}$ can distinguish if the given request is **M**-Type or **AC**-Type, it can be used to build an adversary $\mathcal{B}$ that breaks IND-CPA:

- $\mathcal{B}$ has to submit a pair of messages (m_0, m_1) to the challenger $\mathcal{C}$. She chooses $m_0 = \mathbf{0}$ and m_1 randomly.
- $\mathcal{C}$ chooses $b \in \{0, 1\}$ uniformly random and returns $c^* = Enc_{k_{ab}}(m_b)$ to $\mathcal{B}$. Note that $\mathcal{C}$ uses the secret key k_{ab} that is shared by Alice and Bob for encryption. k_{ab} is not known to $\mathcal{B}$, since the challenge would otherwise be trivial nor to $\mathcal{A}$, since $\mathcal{A}$ is not allowed to corrupt challenge clients.
- $\mathcal{B}$ simulates Vuvuzela for $\mathcal{A}$. In the simulation, Alice and Bob are in active communication with each other. $\mathcal{B}$ can generate all key material except k_{ab} himself. He embeds c^* inside of Alice's request to Bob. For Bob's request, he can use an arbitrary ciphertext generated by the encryption oracle (which, like $\mathcal{C}$, encrypts with k_{ab}). Since $\mathcal{B}$ does not know k_{ab}, he cannot compute the dead drop id that Alice and Bob use as defined by the protocol ($H(k_{ab} \mid r)$). Instead, he chooses a random one (which is indistinguishable to the "real" *pseudo*-random one to $\mathcal{A}$).
- $\mathcal{A}$ interacts with the simulated protocol and determines if Alice send a **AC**-Type ($\mathcal{A}$ returns $b'' = 0$ to $\mathcal{B}$) or **M**-Type request ($\mathcal{A}$ returns $b'' = 1$).
  - If $b'' = 0$, the request must have contained $\mathbf{0}$, thus $\mathcal{B}$ returns $b' = 0$
  - If $b'' = 1$, the request must have contained a message $\neq \mathbf{0}$, thus $\mathcal{B}$ returns $b' = 1$

$\mathcal{B}$ wins the IND-CPA game ($b' = b$), iff $\mathcal{A}$ determines the right type. If $\mathcal{A}$ has a non-negligible advantage in determining request type, then $\mathcal{B}$ has the same advantage in breaking IND-CPA. ■

Figure 4.2 illustrates $\mathcal{B}$'s use of $\mathcal{A}$ to break IND-CPA.

Before tackling **IC**-Type requests, we present three further corollaries concerning **M/AC**-Type requests.

The reduction in Theorem 1 works for all adversaries without the ability to corrupt challenge clients. For future reference, it's useful to explicitly list all abilities that the adversary cannot use to distinguish **M**-Type from **AC**-Type:

Corollary 2 (Adversary Abilities)**.** *Theorem 1 shows that $\mathcal{A}$ distinguish **M**-Type requests from **AC**-Type requests by using* any *combination of his abilities. $\mathcal{A}$ possesses the following abilities:*

- ***Alteration**: $\mathcal{A}$ can alter (i.e., modify, replay, drop, delay) arbitrary requests*
- ***Timing**: $\mathcal{A}$ can time the behavior of arbitrary clients and servers*
- ***Server Corruption**: $\mathcal{A}$ can corrupt (all but one) servers*
- ***Passive Observation**: $\mathcal{A}$ can passively observe on arbitrary network links.*

Since **M**-Type and **AC**-Type requests only differ in their contents, an analogous reduction to Theorem 1 can also be used to show that no adversary without the ability to corrupt clients can determine the content of a given **M**-Type request.

Corollary 3 (Content Indistinguishably)**.** *If $\mathcal{A}$ is not able to distinguish if a given request is **M**-Type or **AC**-Type, he can also not determine the content of a given **M**-Type request without*

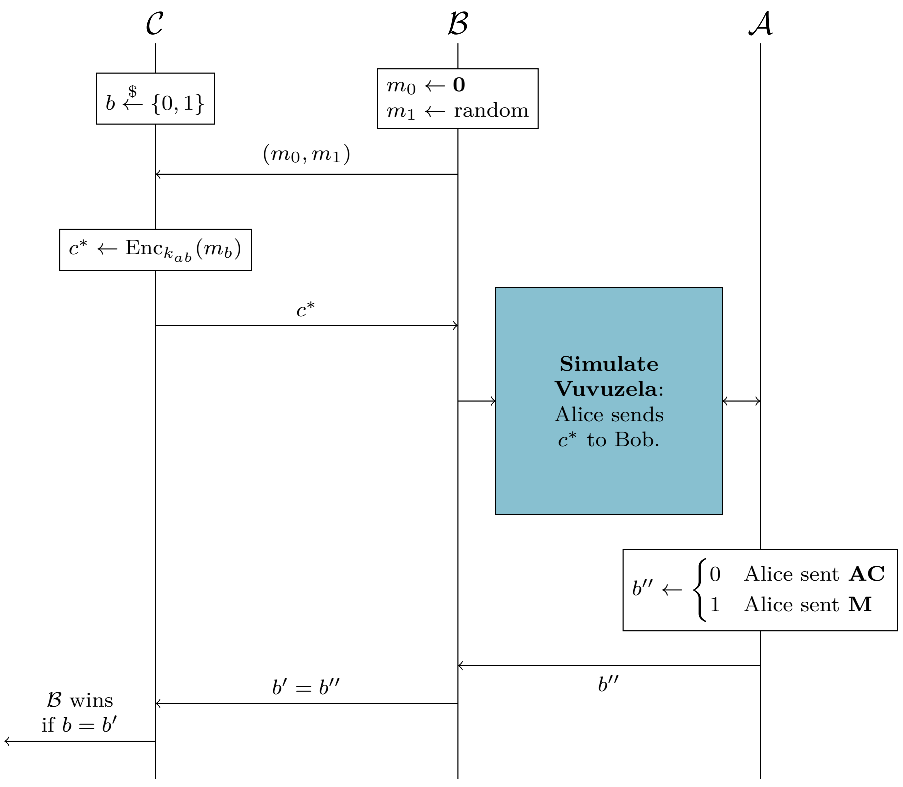

$\mathcal{C}$
$\mathcal{B}$
$\mathcal{A}$
$b \overset{\$}{\leftarrow} \{0,1\}$
$m_0 \leftarrow \mathbf{0}$
$m_1 \leftarrow$ random
(m_0, m_1)
$c^* \leftarrow \text{Enc}_{k_{ab}}(m_b)$
c^*
Simulate Vuvuzela: Alice sends c^* to Bob.
$b'' \leftarrow \begin{cases} 0 & \text{Alice sent } \mathbf{AC} \\ 1 & \text{Alice sent } \mathbf{M} \end{cases}$
b''
$b' = b''$
$\mathcal{B}$ wins if $b = b'$

Figure 4.2: $\mathcal{B}$ uses $\mathcal{A}$, who can distinguish if a given request is **M**-Type or **AC**-Type, to win the IND-CPA game.

corrupting sender or receiver of the request.

Finally, the adversary would also gain information about communication patterns if he could determine the distribution of **M**-Type requests versus **AC**-Type request in any given round. Luckily, we can show that the adversary cannot determine this distribution if he is not able to tell the type of any given request:

Corollary 4. *In a given round, adversaries without the ability to corrupt clients cannot determine how many requests are* ***M****-Type versus* ***AC****-Type.*

Proof. Let n be the number of requests send in the given round. We define a series of hybrid games:

$$H_i := i \text{ Requests are } \mathbf{M}\text{-Type, } n-i \text{ Requests are } \mathbf{AC}\text{-Type} \mid i \in \{0, \dots, n\}$$

In H_0, all requests are **M**-Type and in H_n, all requests are **AC**-Type. Theorem 1 implies that $\mathcal{A}$ cannot distinguish H_i from H_{i+1} (for $i \in \{0, \dots, n\}$), which in turn implies that $\mathcal{A}$ cannot distinguish H_i from H_j for $i, j \in \{1, \dots, n\}$. ■

While $\mathcal{A}$ cannot distinguish between **M**-Type and **AC**-Type, he can distinguish **IC**-Type requests from the other type under specific circumstances. We show that $\mathcal{A}$ can recognize **IC**-Type requests either by timing the sender of the request (Theorem 2) *or* by replaying certain requests (Theorem 3). After that, we prove that those two strategies are the *only* way for the adversary to distinguish **IC**-Type requests from the other types in Theorem 4.

Intuitively, **IC**-Type requests can be recognized by timing the sender, because the client's behavior differs substantially when sending **IC**-Type requests versus other types.

Theorem 2 (**IC**-Type Distinguishability I)**.** *An adversary with access to a timer can distinguish*

***IC**-Type requests from the other types.*

Proof. Let $\mathcal{A}$ be an adversary with access to a timer. $\mathcal{A}$'s goal is to determine if Alice sends an **IC**-Type request in the current round.

We examine Alice's behavior: If Alice sends a **M**-Type or **AC**-Type request, she behaves as follows:

1. Compute the shared key and dead drop id
2. Pad the message if necessary
3. Encrypt the message
4. Onion wrap the request
5. Send the request to the first server

If Alice sends an **IC**-Type message, steps 1-5 are identical, but there is an additional step 0: Generate a random public key.

The generation of this key takes additional time and therefore delays the sending of the request. $\mathcal{A}$ can measure the time it takes Alice to send the request (Assumption 13) and compare it to known values in order to determine the type of the request. ■

Replaying requests can also be used to explicitly distinguish if a given request is **IC**-Type versus **M/AC**-Type[1]:

Theorem 3 (**IC**-Type Distinguishability II)**.** *An adversary with the ability to replay requests and to corrupt the last server can distinguish **IC**-Type requests from the other types.*

Proof. $\mathcal{A}$ replays the request in question and observes the dead drop accesses at the last server. Both replay and original request reach the same dead drop. If the original request is **M/AC**-Type, there is a third request (from the communication partner) also reaching this dead drop. If $\mathcal{A}$ observes this triple dead drop access, the request must have been **M/AC**-Type, since triple dead drop accesses do not happen during normal protocol execution.

If there is no triple dead drop access, the original request must not have had a partner request, i.e., must have been **IC**-Type. ■

Lastly, we examine if other types of adversary can also distinguish **IC**-Type requests from the other types.

Theorem 4. *No adversary without the ability to corrupt clients, access to a timer or the ability to replay requests can distinguish if a given client sends **IC**-Type requests or other types.*

Proof. Let $\mathcal{A}$ be an adversary as described in the theorem. We assume that $\mathcal{A}$ is able to distinguish **IC**-Type requests. We construct a series of hybrid games:

- H_0: No restrictions on $\mathcal{A}$'s abilities
- H_1: H_0 + $\mathcal{A}$ is not allowed to alter requests
- H_2: H_1 + $\mathcal{A}$ is not allowed to corrupt servers
- H_3: H_2 + $\mathcal{A}$ is not allowed to passively observe on links

We show that if $\mathcal{A}$ can win H_0, he can also win H_3

- $H_0 \approx H_1$: $\mathcal{A}$ is not allowed to replay requests per definition. Other types of alterations (i.e., dropping, modifying, delaying) only change the distribution of single and double dead drop accesses (with overwhelming probability). Assumption 6 states that such changes are not analyzable by the adversary due to the server cover traffic. If $\mathcal{A}$ can win H_0, he can also win

[1]We were not able to find prior literature that explicitly mentions Vuvuzela's weakness against replay attacks. But we note that both *Stadium* and *Karaoke* which can be described as being derived from Vuvuzela, are aware of the issue and implement mechanisms to detect replays. Our timing-based attacks against Vuvuzela are novel to the best of our knowledge, both Stadium and Karaoke are still vulnerable. We will explore those protocols in greater detail in Chapter 10.

H_1.

- $H_1 \approx H_2$: $\mathcal{A}$ could gain information by corrupting inner servers or the last server.
  - The inner server does not change its behavior depending on the type of any request: It always decrypts all outer onion layers, adds cover request, shuffles and sends the requests to the next server. It's inner state only consists of the current permutation, information about the cover requests added and key material that enables it to remove/add one onion layer.
  - The last server does change its behavior if he processes an **IC**-Type request: The request causes a single dead drop access and the server generates a random ciphertext as answer. If $\mathcal{A}$ corrupts the last server, there has to be at least one inner server that behaves honestly. If follows with Corollary 1 that $\mathcal{A}$ cannot match requests at the last server to clients, and therefore not identify the given request. Thus, $\mathcal{A}$ does not learn information about the type of the request in question by corrupting servers. If $\mathcal{A}$ can win H_1, he can also win H_2.
- $H_2 \approx H_3$: Follows from Corollary 2: If $\mathcal{A}$ cannot distinguish **M**-Type from **AC**-Type by passively observing, he can also not distinguish **IC**-Type. An **IC**-Type request has the same content as an **AC**-Type request, but is send to a random receiver instead of a real one. No Adversary can distinguish receivers by passively observing on network links: The request cannot be linked from sender to receiver because of onion encryption and shuffling. The receiver cannot be determined by analyzing the request itself, since the only part of the request that depends on the receiver is the shared dead drop id. The id is not only protected by at least one encryption layer on every link, but also requires the shared key k_{ab} to be computed, which $\mathcal{A}$ cannot know without corrupting sender or receiver. If $\mathcal{A}$ can win H_2, he can also win H_3.

In H_3, $\mathcal{A}$ is deprived of *all* abilities. If $\mathcal{A}$ can distinguish the request type in game H_2, he can also do so without observing the challenge execution at all, which is impossible. Since $H_0 \approx H_1 \approx H_2 \approx H_3$ holds, we have shown that $\mathcal{A}$ also has no way of distinguishing in game H_0. ■

4.1.2 Sender-Receiver Linkability

Replay attacks cannot only be used to distinguish **IC**-Type requests but also determine whether two given clients are communicating *with each other*. We introduce the *Double Replay Attack*:

Theorem 5 (Double Replay). *An adversary with the ability to replay requests and to corrupt the last server can determine if two given clients are in active conversation with each other.*

Proof. Let $\mathcal{A}$ be an adversary with the ability to replay requests and to corrupt the last server. Let Alice and Bob be the clients in question.

$\mathcal{A}$ replays the requests of Alice and Bob. This can either happen on the link between Alice/Bob and the first server or at the first server itself.

Vuvuzela has no protection measures against replay attacks, so both replayed requests will reach the last server and cause dead drop accesses. If Alice and Bob are communicating with each other, their requests contain the same dead drop id. Since the duplicated requests also contain the same dead drop id, a *quadruple* dead drop access occurs on the last server which $\mathcal{A}$ can observe.

If Alice and Bob are not communicating, their requests will contain different dead drop ids and no quadruple dead drop access will happen.

This attack only works as described if both clients only send one request per round. If they send multiple requests per round, $\mathcal{A}$ can guess a request to duplicate. In this case, he does not always come to the right result, but he still succeeds with non-negligible probability. ■

An illustration of the double replay attack can be found in Figure 4.3.

4.2. Analysis under the Weak Client Cover Assumption

In the following, we analyze Vuvuzela under the weak client cover assumption (Assumption 9).

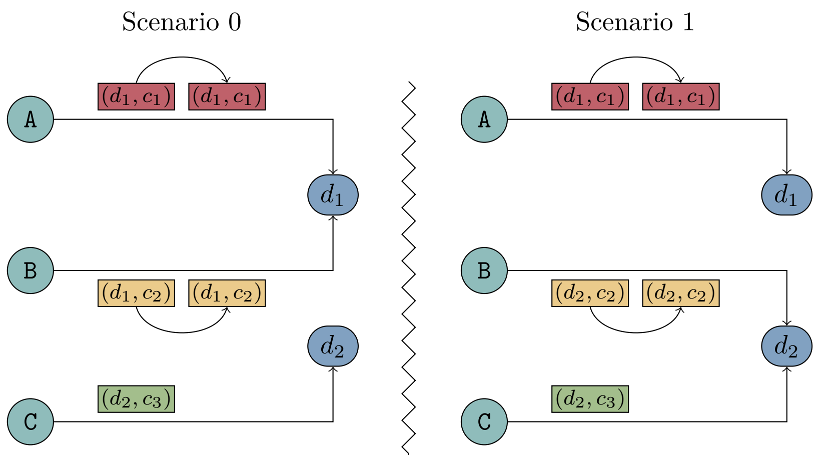


Figure 4.3: The double replay attack. The adversary $\mathcal{A}$ replays Alice's and Bob's requests. Alice and Bob are talking with each other in scenario 0; d_1 receives four and d_2 one accesses. Alice and Bob are *not* talking with each other in scenario 1; d_1 receives two and d_2 three dead drop accesses.

4.2.1 Replay Attacks

Theorem 5 states that an adversary that can replay requests and corrupt the last server can determine if two clients are in active conversation. This directly enables attacks against $(SR)\overline{L}$, $(SR)\overline{O}$, $(2S)\overline{L}$ and $(2R)\overline{L}$.

All attacks in this section can either be executed by $\mathcal{A}_{CM}$ or $\mathcal{A}_{GNM}$ (or stronger adversaries). $\mathcal{A}_{CM}$ has to corrupt the first server to replay requests from challenge clients in in a targeted way. $\mathcal{A}_{GNM}$ has access to the link between client and first server and can therefore replay requests as they leave the clients without needing to corrupt the first server.

As an example, we present the attack of $\mathcal{A}_{CM}$ against $(SR)\overline{L}$ The adversary chooses the following scenarios:

Table 4.1: I_x denotes Instance x. Black requests are **M**-Type requests submitted by the adversary, gray requests are **AC**-Type.

	Scenario 0	Scenario 1
I_0	$(alice, \mathbf{bob}, m, aux)$	$(alice, \mathbf{dave}, m, aux)$
	$(carol, \mathbf{dave}, m, aux)$	$(carol, \mathbf{bob}, m, aux)$
	$(bob, alice, 0, aux)$	$(dave, alice, 0, aux)$
	$(dave, carol, 0, aux)$	$(bob, carol, 0, aux)$
I_1	$(carol, \mathbf{dave}, m, aux)$	$(carol, \mathbf{bob}, m, aux)$
	$(alice, \mathbf{bob}, m, aux)$	$(alice, \mathbf{dave}, m, aux)$
	$(bob, alice, 0, aux)$	$(dave, alice, 0, aux)$
	$(dave, carol, 0, aux)$	$(bob, carol, 0, aux)$

The Attack. The adversary $\mathcal{A}_{CM}$ corrupts the first and last server and duplicates the requests of Alice and Bob. If he observes a quadruple dead drop access, Alice must have communicated with Bob and he returns $b' = 0$. Otherwise, he returns $b' = 1$.

Adversary Advantage. With this attack, the adversary $\mathcal{A}_{CM}$ *always* wins.

The attacks on $(2S)\overline{L}$ and $(2R)\overline{L}$ work as follows: $\mathcal{A}_{CM}$ can check if the main sender/receiver pair[2] of instance 0 is communicating in stage 1. If they are, he checks if they are still communicating in stage 2 (and returns $b' = 0$ if that is the case). If they are not communicating he knows that instance 1 was chosen. In that case, he checks in stage 2 if his main sender/receiver pair from instance 1 is communicating (and returns $b' = 0$ if they are).

[2]The main sender/receiver pair is the one that communicates in both stages of scenario 0.

When attacking $(SR)\overline{O}$ the adversary cannot win every time: With the double replay attack, the adversary can determine the communication partners in one instance. This still gives him an non-negligible advantage of winning.

Further Attacks

The notions $(SM)\overline{L}, (SM)\overline{O}, (RM)\overline{L}$ and $(RM)\overline{O}$ can also be attacked with the double replay technique and an additional trick: The adversary chooses one of the messages to be longer that the maximum message length for one request ℓ. The client who sends the message has to split it and send the rest in the next round. We call this the *message overflow* trick. As an example we present the attack of $\mathcal{A}_{CM}$ against $(SM)\overline{L}$. The adversary chooses the following scenarios:

Table 4.2: I_x denotes Instance x, r_x denotes round x. Black requests are **M**-Type requests submitted by the adversary, gray requests are **AC**-Type and light gray requests are **IC**-Type. m_1^2 denotes the first part of m_1 and m_1^2 the second part.

		Scenario 0	Scenario 1
I_0	r_0	$(alice, bob, \mathbf{m_1^1}, aux)$	$(alice, bob, \mathbf{m_2}, aux)$
		$(carol, \mathbf{bob}, \mathbf{m_2}, aux)$	$(carol, bob, \mathbf{m_1^1}, aux)$
		$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
		$(bob, carol, 0, aux)$	$(bob, carol, 0, aux)$
	r_1	$(alice, \mathbf{bob}, \mathbf{m_1^2}, aux)$	$(alice, \mathbf{rand}, \mathbf{0}, aux)$
		$(carol, \mathbf{rand}, \mathbf{0}, aux)$	$(carol, \mathbf{bob}, \mathbf{m_1^2}, aux)$
		$(bob, \mathbf{alice}, 0, aux)$	$(bob, \mathbf{rand}, 0, aux)$
		$(bob, \mathbf{carol}, 0, aux)$	$(bob, \mathbf{carol}, 0, aux)$
I_1	r_0	...	...

The Attack. $\mathcal{A}_{CM}$ duplicates Alice's and both of Bob's requests during the second round. If scenario 0 was chosen, one of Bob's request contains the same dead drop id as Alice's request (and $\mathcal{A}$ can observe a quadruple dead drop access). In scenario 1 both will contain different dead drop ids.

Adversary Advantage. With this attack, the adversary $\mathcal{A}_{CM}$ *always* wins.

4.2.2 Timing Attacks

Theorem 2 states that an adversary with access to a timer can distinguish **IC**-Type requests from the other types. Thus, all attacks in this section can be executed by $\mathcal{A}_{GT}$ and $\mathcal{A}_{CT}$ (and stronger adversaries). $\mathcal{A}_{GT}$ has access to the link between clients and first server and can therefore time the request leaving the sender. $\mathcal{A}_{CT}$ can corrupt the first server and time the request's arrival.

An Introductory Attack. With this Information, the adversaries $\mathcal{A}_{GT}$ and $\mathcal{A}_{CT}$ can attack $(2S)\overline{L}$. The adversary times the sending behavior of all his challenge clients. If the clients who are active/inactive are equal in both stages, he returns $b' = 0$. If the clients in active conversation change between stages, he returns $b' = 1$. $(2R)\overline{L}$ can be attacked with the same strategy.

Message Observability

We show an attack of $\mathcal{A}_{GT}$ against $(SR)\overline{O}$. The adversary chooses the following scenarios:

Table 4.3: I_x denotes Instance x. Black requests are **M**-Type requests submitted by the adversary, gray requests are **AC**-Type and light gray requests are **IC**-Type.

	Scenario 0	Scenario 1
I_0	$(alice, \mathbf{bob}, m, aux)$	$(alice, \mathbf{dave}, m, aux)$

	Scenario 0	Scenario 1
I_1	$(\mathbf{bob}, alice, 0, aux)$	$(\mathbf{dave}, alice, 0, aux)$
	$(\mathbf{dave}, rand, 0, aux)$	$(\mathbf{bob}, rand, 0, aux)$
	$(carol, rand, 0, aux)$	$(carol, rand, 0, aux)$
	$(carol, \mathbf{dave}, m, aux)$	$(carol, \mathbf{bob}, m, aux)$
	$(\mathbf{dave}, carol, 0, aux)$	$(\mathbf{bob}, carol, 0, aux)$
	$(\mathbf{bob}, rand, 0, aux)$	$(\mathbf{dave}, rand, 0, aux)$
	$(alice, rand, 0, aux)$	$(alice, rand, 0, aux)$

The Attack. $\mathcal{A}_{GT}$ times the sending behavior of all four challenge clients. He observes one of four possible cases:

- Carol and Dave send **IC**-Type requests ⇒ $\mathcal{A}_{GT}$ returns $b' = 0$
- Carol and Bob send **IC**-Type requests ⇒ $\mathcal{A}_{GT}$ returns $b' = 1$
- Alice and Dave send **IC**-Type requests ⇒ $\mathcal{A}_{GT}$ returns $b' = 1$
- Alice and Bob send **IC**-Type requests ⇒ $\mathcal{A}_{GT}$ returns $b' = 0$

Adversary Advantage. With this attack, the $\mathcal{A}_{GT}$ *always* wins.

(SR)$\overline{\mathbf{L}}$. The notion $(SR)\overline{L}$ cannot be broken with timing attacks, since all clients are in active conversation in both scenarios.

Sender and Receiver Observability

Analogously to replay attacks, timing attacks can be extended with the *message overflow* trick. For $(SM)\overline{L}$, $\mathcal{A}_{GT}$ can determine if Alice sends a **M/AC**-Type request (scenario 0) or an **IC**-Type request (scenario 1) during the second round. $(SM)\overline{O}$, $(RM)\overline{L}$ and $(RM)\overline{O}$ can be attacked in the same way.

4.2.3 Sender-Receiver Linkability

In the following, we show that no adversary can win the $M\overline{O} - |m|$-game under the weak client cover assumption.

Theorem 6. *No adversary without the ability to corrupt challenge clients can break $M\overline{O} - |m|$ under the weak client cover assumption.*

Proof. $M\overline{O} - |m|$ restricts the communications to only differ in the message content, sender and receiver have to be equal. As a consequence, the clients in active conversation with each other are equal in both scenarios. Further, since $M\overline{O} - |m|$ restricts messages to be the same length, the distribution of **M**-Type to **AC**-Type requests is also identical between the scenarios.

It follows that any adversary $\mathcal{A}$ that can win the $M\overline{O} - |m|$-game can distinguish scenarios by the content of the **M**-Type requests, which is a contradiction to Corollary 3. ■

4.2.4 Other Adversaries

What remains to be shown are the limits of adversaries with out access to a timer or the ability to replay requests. Theorem 4 shows that no such adversary can distinguish between **IC**-Type requests and other requests. Due to the cover generation, the scenarios in the $C\overline{O}$-game are identical except for the type of the requests. Thus, no adversary without access to a timer or the ability to alter requests can win the $C\overline{O}$-game.

4.2.5 Summary

Under the weak client cover assumption $\mathcal{A}_{CM}$ and $\mathcal{A}_{GNM}$ can break any notion except $M\overline{O} - |m|$ using replay attacks. $\mathcal{A}_{GT}$ and $\mathcal{A}_{CT}$ can attack based on sender timing and break any notion except $M\overline{O} - |m|$ and $(SR)\overline{L}$. Adversaries without the abilities to time or alter requests cannot win the $C\overline{O}$-game.

4.3. Analysis under the Medium Client Cover Assumption

In the following, we analyze Vuvuzela under the medium client cover assumption (Assumption 10).

4.3.1 Message Unobservability

Theorem 7. *No adversary without the ability to corrupt clients can break $M\overline{O}$ under the medium client cover assumption.*

Proof. First we note that the set of clients in active conversation is always equal in both scenarios due to the definition of the $M\overline{O}$-game. What changes between the scenario are the messages that are send and therefore the content of the requests and the distribution of **M**-Type versus **AC**-Type requests per round.

Since $\mathcal{A}$ is not allowed to corrupt challenge clients, we can use Corollary 4: It states that no such adversary can distinguish how many requests are **M**-Type and **AC**-Type. It also follows with Theorem 1 that no adversary can can tell if any one specific request is **M**-Type or **AC**-Type.

Further, Corollary 3 shows that the adversary can not distinguish **M**-Type requests by their content.

Thus, $\mathcal{A}$ cannot distinguish between scenarios in any one round. What remains to be shown is that $\mathcal{A}$ does not learn any information that enables him to tell the **M**-Type versus **AC**-Type distribution from observing multiple rounds.

Much of the information an adversary can learn from observing Vuvuzela is chosen randomly each round. This includes the encryption keys for the onion layers, the used dead drop ids and the permutation for the request shuffle. What remains constant over multiple rounds are the main public and secret keys for clients and servers and therefore notably the innermost encryption key for sender-receiver pairs.

The server behavior in the current round does not depend on behavior in previous rounds. The ciphertexts appear uniformly random each round (even if the content is the same), because of the assumed IND-CPA security of the used encryption scheme. Internal client behavior cannot be observed, since $\mathcal{A}$ is not allowed to corrupt clients. ■

4.3.2 Replay Attacks

As described in Section 4.2, adversaries with the ability to alter requests can execute replay attacks. The attacks against $(SR)\overline{O}$ and $(SR)\overline{L}$ can also be done here.

Due to the stronger client cover assumption, the *message overflow* trick does not work anymore; The clients remain in active communication in the second round[3].

$(2S)\overline{L}$ and $(2R)\overline{L}$ are also not breakable by double replay attack under the medium client cover assumption, since both clients remain in active communication in the second stage.

4.3.3 Timing Attacks

Timing attacks as described in Theorem 2 require the set of clients in active communication to differ between the scenarios. Under the medium client cover assumption, the weakest notion to allow this is $S\overline{O} - P$. Here, the adversary can choose the following scenarios:

[3]And we have proven $M\overline{O}$ to be unbreakable, which implies that $(SM)\overline{L}$, etc. are also unbreakable.

Table 4.4: Black requests are **M**-Type requests submitted by the adversary, gray requests are **AC**-Type and light gray requests are **IC**-Type.

Scenario 0	Scenario 1
$(\mathbf{alice}, bob, m, aux)$	$(\mathbf{carol}, bob, m, aux)$
$(bob, \mathbf{alice}, 0, aux)$	$(bob, \mathbf{carol}, 0, aux)$
$(\mathbf{carol}, rand, 0, aux)$	$(\mathbf{alice}, rand, 0, aux)$

$\mathcal{A}_{CT}$ or $\mathcal{A}_{GT}$ can time the sending behavior of Alice and determine if she is sending an **IC**-Type request or not and therefore distinguish between the scenarios.

The notion $SF\overline{L}$ restricts the adversary to only submit challenges where the same clients are active in both scenarios. It follows that neither $\mathcal{A}_{CT}$ nor $\mathcal{A}_{GT}$ can win $SF\overline{L}$ by timing the senders.

The same is true for $(2S)\overline{L}$ and $(2R)\overline{L}$: Due to the restrictions of the medium client cover assumption, the clients in active communication remain the same in both stages and therefore in both scenarios.

4.3.4 Other Adversaries

In the previous section, we have shown that no adversary without the ability to time or alter requests can win the $C\overline{O}$-game under the weak client cover assumption. Since every attack possible under the medium client cover assumption is also possible under the weak client cover assumption, the same is true here: No such adversary can break $C\overline{O}$ under the medium client cover assumption.

4.3.5 Summary

Under the medium client cover assumption $\mathcal{A}_{CM}$ and $\mathcal{A}_{GNM}$ can break $(SR)\overline{L}$ and $(SR)\overline{O}$. $\mathcal{A}_{CT}$ and $\mathcal{A}_{GT}$ can break $S\overline{O} - P$.

No adversaries can break $M\overline{O}$, $(2S)\overline{L}$ and $(2R)\overline{L}$. Adversaries without the abilities to time or alter requests cannot win the $C\overline{O}$-game.

4.4. Analysis under the Strong Client Cover Assumption

In the following, we analyze Vuvuzela under the strong client cover assumption (Assumption 11).

Theorem 8 (Communication Unobservability)**.** *No adversary $\mathcal{A}$ without the ability to corrupt challenge clients can break $C\overline{O}$ under the strong client cover assumption.*

Proof. We define a series of hybrid games:

- H_0: The original $C\overline{O}$-game
- H_1: H_0, but all messages have to be 0
- H_2: H_1, but clients send **M**-Type requests (with message 0) instead of **AC**-Type requests
- H_3: Identical Scenarios

In the following, we show that any adversary that can win the game H_i can also win the game H_{i+1} (for $i \in \{0, \ldots, 2\}$). Since H_0 is the $C\overline{O}$-game, the adversary is not allowed to corrupt any senders or receivers in the hybrid games.

1. $H_0 \approx H_1$: Corollary 3 shows that any adversary $\mathcal{A}$ who breaks H_0 does so without relying on different messages between the scenarios. Thus, $\mathcal{A}$ can also break H_1.
2. $H_1 \approx H_2$: Corollary 4 shows that no adversary without access to challenge clients can determine how many requests are **M**-Type versus **AC**-Type. Note that the only difference

between H_1 and H_2 is the distribution of request types: There are only **M**-Type messages in H_2 and a mix of **M**- and **AC**-Type in H_1. It follows that if $\mathcal{A}$ wins the H_1-game, he can also H_2, since the difference between the games is indistinguishable to him.

3. $H_2 \approx H_3$: With Assumption 1, we assume that the set of clients in active conversations is the same in both scenarios. Since clients in active conversation send requests to all of their communication partners, and the message content is restricted to 0 by H_1, the scenarios in H_2 are already identical.

Let $\mathcal{A}$ have an non-negligible advantage of winning the H_0-game. We have shown that $\mathcal{A}$ also has a non-negligible advantage of winning the H_3-game. Since H_3 contains identical scenarios, no adversary can win with probability better than guessing. ■

4.5. Additional Assumptions

In the following sections, we analyze Vuvuzela under additional assumptions. We start with Section 4.5.1, where we examine the consequences of allowing the adversary to corrupt challenge clients. In Section 4.5.2, we assume that the innermost encryption is deterministic.

The analysis in both cases is under the strong client cover assumption, the results are applicable under all three client cover assumptions.

4.5.1 The Client Corruption Case

In this section, an adversary that has the ability to corrupt nodes (e.g., $\mathcal{A}_C$ and $\mathcal{A}_{GN}$) is allowed to corrupt the clients from his challenge. To avoid trivial attacks, we restrict him to only corrupting senders in receiver-focused notions (such as $R\overline{O}$) and only corrupting receivers in sender-focused notions (such as $S\overline{O}$).

To make it easier to see which kind of corruption is allowed during the attack, we borrow the following notation from [1, Sec. 7.2]:

- X_{C^s}: $\mathcal{A}$ is *not* allowed to corrupt challenge senders when attacking notion X (but may corrupt receivers)
- X_{C^r}: $\mathcal{A}$ is *not* allowed to corrupt challenge receivers when attacking notion X (but may corrupt senders)
- X_{C^e}: $\mathcal{A}$ can corrupt arbitrary challenge clients, but corrupted clients send identical messages in both scenarios.

First, we note that $M\overline{O} - |m|_{C^s}$ and $M\overline{O} - |m|_{C^r}$ are broken trivially[4]: In scenario 0, Alice send m_1 to Bob and in scenario 1, Alice sends m_2. The adversary can distinguish the scenarios by either corrupting the sender or the receiver and and checking which of the both messages is send/received.

Other notions can be broken with the *inner key attack*. Here, the adversary exploits the fact that in Vuvuzela, the innermost encryption key is long-lived, since it is directly derived from the communication partner's public and secret keys. Thus, the innermost encryption key only changes if one of the communication partners revokes their public key.

An intuitive example of this technique is the attack of $\mathcal{A}_C$ against $(2S)\overline{L}_{C^s}$. The adversary chooses the following scenarios:

Table 4.5: I_x denotes instance x, S_x denotes stage x. Cover requests are omitted.

		Scenario 0	Scenario 1
I_0	S_1	$(alice, bob, m, aux)$	$(alice, bob, m, aux)$
	S_2	$(\textbf{alice}, bob, m, aux)$	$(\textbf{carol}, bob, m, aux)$
I_1	S_1	$(carol, bob, m, aux)$	$(carol, bob, m, aux)$

[4]$M\overline{O} - |m|_{C^e}$ cannot be broken, it restricts the adversary to submitting identical scenarios.

	Scenario 0	Scenario 1
S_2	$(\textbf{carol}, bob, m, aux)$	$(\textbf{alice}, bob, m, aux)$

Here, Alice and Bob share one encryption key denoted as k_{ab} and Carol and Bob share an encryption key denoted as k_{cb}[5]. If Alice wants to send a message to Bob, she encrypts it with k_{ab} before adding the onion layer. If Carol wants to send a message to Bob, she encrypts it with k_{ac} before adding the onion layer. Consequently, Bob encrypts a message from Alice with k_{ab} and a message from Carol with k_{ac}.

The Attack. The adversary $\mathcal{A}_C$ corrupts Bob and observes which key Bob uses to decrypt the message m in stage 1 and in stage 2. If Bob uses the same key in both stages, the sender must have been the same, and $\mathcal{A}_C$ returns $b' = 0$. If Bob uses different keys for the stages, $\mathcal{A}_C$ returns $b' = 1$.

$(2R)\overline{L}_{C^r}$ can be attacked analogously.

Adversary Advantage. With this attack, the adversary $\mathcal{A}_C$ *always* wins.

Attacking other Notions

To attack the other notions, the *inner key attack* can be extended. To distinguish the scenarios, the adversary adds identification batches with a single (identical) communication in both scenario to the challenge. He uses each of these batches to determine the inner key for one sender-receiver pair since the communication partner in this batch is known and independent of the chosen scenario. This knowledge can be used to find out communication partners and therefore distinguish scenarios in batches with differing communications. As an example, we present an attack against $(RM)\overline{O}_{C^r}$.

$(RM)\overline{L}_{C^r}$, $(SM)\overline{O}_{C^s}$, $(SM)\overline{L}_{C^s}$, $(SR)\overline{O}_{C^e}$ and $(SR)\overline{L}_{C^e}$ can be attacked analogously by either $\mathcal{A}_C$ or $\mathcal{A}_{GN}$.

$\mathbf{(RM)\overline{O}_{C^r}}$. The adversary chooses the following scenarios:

Table 4.6: B_x denotes batch x, I_x denotes instance x. Cover requests are omitted.

		Scenario 0	Scenario 1
B_0		$(alice, bob, m_1, aux)$	$(alice, bob, m_1, aux)$
B_1		$(alice, carol, m_2, aux)$	$(alice, carol, m_2, aux)$
B_2	I_0	$(alice, bob, \mathbf{m_1}, aux)$	$(alice, bob, \mathbf{m_2}, aux)$
	I_1	$(alice, carol, \mathbf{m_2}, aux)$	$(alice, carol, \mathbf{m_1}, aux)$

The Attack. The adversary $\mathcal{A}_C$ corrupts Alice. During round 0 (where batch 0 is send), $\mathcal{A}_C$ observes which key Alice uses for the innermost encryption layer when encrypting m_1. Let this key be k_{ab}. During round 1 (where batch 1 is send), $\mathcal{A}_C$ observes which key Alice uses for the innermost encryption layer when encrypting m_2. Let this key be k_{ac}. During round 2 (where batch 2 is send), $\mathcal{A}_C$ can observe one of four possible behaviors:

1. Alice sends m_1 and uses k_{ab} for the encryption $\Rightarrow \mathcal{A}_C$ returns $b' = 0$
2. Alice sends m_2 and uses k_{ab} for the encryption $\Rightarrow \mathcal{A}_C$ returns $b' = 1$
3. Alice sends m_1 and uses k_{ac} for the encryption $\Rightarrow \mathcal{A}_C$ returns $b' = 1$
4. Alice sends m_2 and uses k_{ac} for the encryption $\Rightarrow \mathcal{A}_C$ returns $b' = 0$

Adversary Advantage. With this attack, the adversary $\mathcal{A}_C$ *always* wins.

[5]Alice and Carol might also share a encryption key, but it's not relevant to the attack.

Summary

We have seen that under the assumption that challenge clients are allowed to be corrupted, the adversaries $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ can break *all* notions. Adversaries that do not have the ability to corrupt clients do not have any advantage over Sec. 4.4.

4.5.2 The Deterministic Encryption Case

In the following section, we relax Assumption 3: Now the encryption scheme that is used for the innermost encryption is *deterministic*, which contradicts IND-CPA security.

Long-lived Symmetric Keys. The symmetric key k_{ab} that the client uses to encrypt the plaintext message is derived from the client's secret key and the communication partner's public key. Since these keys do not change over time, k_{ab} does not either. When combined with a *deterministic* encryption scheme, this has the effect that if Alice wants to send the same message multiple times to Bob (over the course of multiple rounds), $Enc_{k_{ab}}(m)$ is always the same. Since the encryption keys for the onion layers and the used dead drop id change every round, the request that is send up the server chain is not the same, but an adversary that has corrupted the last server can determine if a given message exchange has happened multiple times.

[2, Sec. 9] mentions the following:

> On the other hand, Vuvuzela's communication protocol provides forward secrecy by choosing new server keys each round, and existing techniques can achieve forward secrecy for message contents [31].

Where "[31]" references *Signal*'s Double Ratched Algorithm (see `signal.org`). While this would solve the issue, we still include this section since the main protocol description does not mention the need for *non-deterministic* encryption.

Attack: $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ vs. $(SM)\overline{L}$

The adversary chooses the following scenarios:

Table 4.7: B_x denotes batch x, I_x denotes instance x. Black requests are **M**-Type requests submitted by the adversary, gray requests are **AC**-Type.

		Scenario 0	Scenario 1
B_0		$(alice, bob, m_1, aux)$	$(alice, bob, m_1, aux)$
		$(carol, bob, 0, aux)$	$(carol, bob, 0, aux)$
		$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
		$(bob, carol, 0, aux)$	$(bob, carol, 0, aux)$
B_1	I_0	$(alice, bob, \mathbf{m_1}, aux)$	$(alice, bob, \mathbf{m_2}, aux)$
		$(carol, bob, \mathbf{m_2}, aux)$	$(carol, bob, \mathbf{m_1}, aux)$
		$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
		$(bob, carol, 0, aux)$	$(bob, carol, 0, aux)$
	I_1	$(carol, bob, \mathbf{m_2}, aux)$	$(carol, bob, \mathbf{m_1}, aux)$
		$(alice, bob, \mathbf{m_1}, aux)$	$(alice, bob, \mathbf{m_2}, aux)$
		$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
		$(bob, carol, 0, aux)$	$(bob, carol, 0, aux)$

The Attack. The adversary $\mathcal{A}_C$ or $\mathcal{A}_{GN}$ corrupts the last server. During batch B_0, the adversary saves all tuples (d, e) that the last server receives (after decrypting the last onion layer). One of those tuples is $(d, Enc_{k_{ab}}(m_1))$, i.e., the corresponding request to the communication $(alice, bob, m_1, aux)$.

During batch B_1, the adversary determines for each tuple (d', e') that the last server receives if there exists a tuple from the first batch with $e = e'$. There will always be at least two matches, since Bob sends one cover request each to Alice and to Carol in both batches. If there is a third match, the (real) communication from batch B_0 must have also happened during batch B_1. The adversary returns $b' = 0$.

Adversary Advantage. With this attack, the adversaries $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ *always* win.

$(SR)\overline{L}$ and $(RM)\overline{L}$ can be attacked analogously. With $(SR)\overline{L}$, the adversary has to decide if there are 6 or 7 matches, see Table 15.1 for more detail.

Attack: $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ vs. $(SM)\overline{O}$

The adversary chooses the following scenarios:

Table 4.8: B_x denotes batch x, I_x denotes instance x. Black requests are **M**-Type requests submitted by the adversary, gray requests are **AC**-Type.

		Scenario 0	Scenario 1
B_0		$(alice, bob, m_1, aux)$	$(alice, bob, m_1, aux)$
		$(carol, bob, 0, aux)$	$(carol, bob, 0, aux)$
		$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
		$(bob, carol, 0, aux)$	$(bob, carol, 0, aux)$
B_1		$(carol, bob, m_2, aux)$	$(carol, bob, m_2, aux)$
		$(alice, bob, 0, aux)$	$(alice, bob, 0, aux)$
		$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
		$(bob, carol, 0, aux)$	$(bob, carol, 0, aux)$
B_2	I_0	$(alice, bob, \mathbf{m_1}, aux)$	$(alice, bob, \mathbf{m_2}, aux)$
		$(carol, bob, 0, aux)$	$(carol, bob, 0, aux)$
		$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
		$(bob, carol, 0, aux)$	$(bob, carol, 0, aux)$
	I_1	$(carol, bob, \mathbf{m_2}, aux)$	$(carol, bob, \mathbf{m_1}, aux)$
		$(alice, bob, 0, aux)$	$(alice, bob, 0, aux)$
		$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
		$(bob, carol, 0, aux)$	$(bob, carol, 0, aux)$

The Attack. The adversary $\mathcal{A}_C$ or $\mathcal{A}_{GN}$ corrupts the last server. He behaves in the same way as described for $(SM)\overline{L}$. If he observes four matches between batches B_0 and B_2 *or* four matches between batch B_1 and B_2, he returns $b' = 0$. Otherwise, he returns $b' = 1$.

Adversary Advantage. With this attack, the adversaries $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ *always* win.

$(SR)\overline{O}$ and $(RM)\overline{O}$ can be attacked analogously.

Attack: $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ vs. $M\overline{O} - |M|$

The adversary chooses the following scenarios:

Table 4.9: B_x denotes batch x. Black requests are **M**-Type requests submitted by the adversary, gray requests are **AC**-Type.

	Scenario 0	Scenario 1
B_0	$(alice, bob, m_1, aux)$	$(alice, bob, m_1, aux)$
	$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
B_1	$(alice, bob, \mathbf{m_1}, aux)$	$(alice, bob, \mathbf{m_2}, aux)$
	$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$

The Attack. The adversary $\mathcal{A}_C$ or $\mathcal{A}_{GN}$ corrupts the last server. He behaves in the same way as described for $(SM)\overline{L}$. Here, he has to determine if there are one or two matches.

Adversary Advantage. With this attack, the adversaries $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ *always* win.

Attack: $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ vs. $(2S)\overline{L}$

Table 4.10: B_x denotes batch x, S_x denotes stage x. Black requests are **M**-Type requests submitted by the adversary, gray requests are **AC**-Type.

		Scenario 0	Scenario 1
I_0	S_1	$(alice, bob, m, aux)$	$(alice, bob, m, aux)$
		$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
		$(carol, bob, 0, aux)$	$(carol, bob, 0, aux)$
		$(bob, carol, 0, aux)$	$(bob, carol, 0, aux)$
	S_2	$(\mathbf{alice}, bob, m, aux)$	$(\mathbf{carol}, bob, m, aux)$
		$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
		$(carol, bob, 0, aux)$	$(alice, bob, 0, aux)$
		$(bob, carol, 0, aux)$	$(bob, carol, 0, aux)$
I_1	S_1	$(carol, bob, m, aux)$	$(carol, bob, m, aux)$
		$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
		$(alice, bob, 0, aux)$	$(alice, bob, 0, aux)$
		$(bob, carol, 0, aux)$	$(bob, carol, 0, aux)$
	S_2	$(\mathbf{carol}, bob, m, aux)$	$(\mathbf{alice}, bob, m, aux)$
		$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
		$(alice, bob, 0, aux)$	$(carol, bob, 0, aux)$
		$(bob, carol, 0, aux)$	$(bob, carol, 0, aux)$

The Attack. The adversary $\mathcal{A}_C$ or $\mathcal{A}_{GN}$ corrupts the last server. He behaves in the same way as described for $(SM)\overline{L}$. If he observes four matches between stage S_1 and stage S_2, he returns $b' = 0$. If he observes two matches between the stages, he returns $b' = 1$.

Adversary Advantage. With this attack, the adversaries $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ *always* win.

$(2R)\overline{L}$ can be attacked analogously.

On other Adversaries

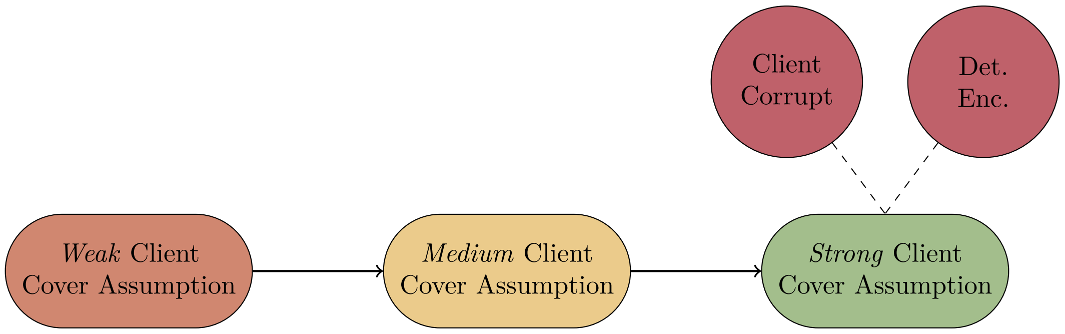
We have shown that the adversaries $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ can break any notion under the assumption that the symmetric encryption is deterministic. What remains to be shown is that the adversaries $\mathcal{A}_G$ and $\mathcal{A}_{GT}$ gain no further attack possibilities compared to Sec. 4.4.

We note that in order to use the additional assumption in this section, the adversary has to be able to observe the innermost ciphertext $Enc_{k_{ab}}(m)$. This is only possible by corrupting either a client or the last server. The adversaries $\mathcal{A}_G$ and $\mathcal{A}_{GT}$ are only able to passively observe on links, where the innermost ciphertext is always hidden by at least one further encryption layer which is generated with a new randomly chosen key each round.

4.6. Summary

We have analyzed Vuvuzela under three variants of the client cover assumption.

Under the *weak* client cover assumption, $\mathcal{A}_{CM}$ and $\mathcal{A}_{GNM}$ can break any notion except $M\overline{O} - |m|$. To do so, the adversaries use *double replay* attacks. $\mathcal{A}_{GT}$ and $\mathcal{A}_{CT}$ can break any notion except $M\overline{O} - |m|$ and $(SR)\overline{L}$. These attacks are enabled by timing differences in the sending behavior.


Client Corrupt
Det. Enc.
Weak Client Cover Assumption
Medium Client Cover Assumption
Strong Client Cover Assumption

🟩 $C\overline{O}$ holds against $\mathcal{A}_{GCTM}$
🟨 $\mathcal{A}_{CM}/\mathcal{A}_{GNM}$ break $(SR)\overline{L}$ and $(SR)\overline{O}$. $\mathcal{A}_{CT}/\mathcal{A}_{GT}$ break $S\overline{O}-P$
🟧 $\mathcal{A}_{CM}/\mathcal{A}_{GNM}$ break any notion **except** $M\overline{O}-|m|$
$\mathcal{A}_{GT}/\mathcal{A}_{CT}$ brake any notion **except** $M\overline{O}-|m|$ and $(SR)\overline{L}$
🟥 $\mathcal{A}_{C}/\mathcal{A}_{GN}$ break any notion

Figure 4.4: Summary of the analysis

Under the *medium* client cover assumption, $\mathcal{A}_{CM}$ and $\mathcal{A}_{GNM}$ can break $(SR)\overline{L}$ and $(SR)\overline{O}$. $\mathcal{A}_{GT}$ and $\mathcal{A}_{CT}$ can break $S\overline{O}-P$. As with the weak client cover assumption, the attacks by altering adversaries are enabled by *double replay* attacks and the attacks by timing adversaries by sender timing.

Under the *strong* client cover assumption, no adversary can break $C\overline{O}$.

If the adversary is allowed to corrupt challenge clients, $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ are able to break *any* notion. This is due to the *inner key weakness.*

Lastly, we determined if the encryption scheme for the innermost onion layer is deterministic, the adversaries $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ can break *any* notion without corrupting clients and under the strong active sender assumption. They do so by observing the ciphertexts that are exchanged at the last server.

Figure 4.4 shows a visual summary of our results.

4.7. Discussion of the Vuvuzela Communication Protocol

The results of our analysis strongly depend on the variant of the client cover assumption: Even though no adversary can win any notion under the strong client cover assumption, all notions except $M\overline{O}-|m|$ break under the weak client cover assumption.

While the medium client cover assumption is more realistic than the weak client cover assumption (where the clients drop out of communication as soon as they have no message to send), Vuvuzela should also protect the privacy under the weak client cover assumption. The authors of the Vuvuzela paper explicitly state that the adversary should not be able to determine if the client is in active conversation or not.

The attacks under the weak and medium client cover assumptions are enabled by two main weaknesses: A weakness against replay attacks and a sender timing weakness.

Replay Attacks. The *double replay* attack enables the adversary to determine if two clients are in active conversation with each other. This is one of the main types of information ACNs try to protect and leaking it has big consequences in many real world scenarios. For example, a totalitarian government could easily prove that a suspected whistle blower is indeed talking to the international press.

To counter these attacks, the Vuvuzela servers can be extended to detect and discard replayed requests. There is existing literature (e.g, [14]) that explores methods to do so.

Timing Attacks. An adversary $\mathcal{A}$ can time the sending behavior of a client to determine the number of active conversations the client is in.

This weakness can be fixed by adjusting the sender protocol. The client could always generate the random key pair needed for cover traffic whether he needs it or not.

Client Corruption. Vuvuzela requires that both sender and receiver are trusted in order to make any privacy guarantees. This is quite a strong requirement, so we decided to analyze the consequences of allowing the adversary to corrupt (challenge) clients. If Vuvuzela is analyzed under the assumption that either senders or receivers can be corrupted by the adversary, successful attacks can be found against *any* privacy notion (see Sec. 4.5.1). These attacks are enabled by the fact that the innermost encryption key is long-lived. A real-world attack based on this weakness would work as follows:

Let Alice communicate with Bob, who is corrupted by the adversary $\mathcal{A}$. If Alice reveals her identity in a single message by accident to Bob (e.g., by signing the message), $\mathcal{A}$ learns a mapping from decryption key to communication partner. If Bob uses this key again in a later round to decrypt an incoming message, $\mathcal{A}$ knows that the message must have been from Alice, even if there is no identifying information in the message content.

To limit such attacks, the client's public and secret keys could have a limited lifetime. Every client could generate a new key pair and reset its inner state at a set interval (e.g., weekly). For this approach to work, the clients have to be synchronized, i.e. changes keys at exactly the same time. After the reset, the adversary could not match Alice's messages anymore since the shared key between Alice and Bob has changed.

Deterministic Encryption. If the long-lived encryption keys are combined with a deterministic encryption scheme, the adversary $\mathcal{A}$ can successfully attack *any* privacy notion without the need to corrupt clients. We analyzed this case in Sec. 4.5.2. There, the adversary can corrupt the last server and examine the ciphertexts that are exchanged via the dead drops. Identical messages between the same sender-receiver pair will always result in identical ciphertexts. An example for a real-world attack based on this weakness is the following:

The adversary $\mathcal{A}$ can determine how active the network is by corrupting the last server. We assume that clients do not send identical messages in normal communication most of the time and note that cover requests between a sender and receiver always result in the same ciphertext at the last server. $\mathcal{A}$ observes the dead drop accesses. The less known ciphertexts he observes, the higher the percentage of real messages has to be and therefore the more active the communications in the network are.

There are at least two ways to prevent these attacks: First, a non-deterministic encryption scheme could be used. If this is not possible, we suggest randomizing the innermost encryption key. Instead of encrypting every message with the same key k_{ab}, the client would use a collision-resistant hash function $H(\cdot)$ to compute k_{ab}^r in round r as follows:

$$k_{ab}^r = k_{ab} \oplus H(r)$$

k_{ab}^r is then used to encrypt the message (only) in round r, the resulting ciphertext will not be linkable to previous ciphertexts, even if the message is identical.

A remaining problem with this approach is that if Alice and Bob both send the same message in one round (i.e., an **AC**-Type request), identical ciphertexts are exchanged. Thus, the adversary could still determine the number of clients in active conversation. To prevent this, different inner keys for each direction can be used:

$$k_{ab}^{r} = k_{ab} \oplus H(pk_a, r)$$
$$k_{ba}^{r} = k_{ab} \oplus H(pk_b, r)$$

Alice would use k_{ab}^{r} to encrypt messages for Bob and Bob would use k_{ba}^{r} to encrypt messages for Alice.

5. Analysis of the Vuvuzela Dialing Protocol

We start the analysis by presenting some auxiliary results in Section 5.1, which will simplify later proofs. The main analysis can be found in Section 5.2. Section 5.3 contains a summary of our results. Finally, we discuss VDP and our results in Section 5.4.

5.1. Auxiliaries

We start by showing that invitations are handled independently from each other in Section 5.1.1. This will simplify later proofs, e.g., modifications of other invitations can have no effect on the invitation in question. After that, we examine if adversaries can distinguish request types in Section 5.1.2. Section 5.1.3 contains a proof that senders can *only* be linked to dead drops by replay, timing or client corruption.

5.1.1 Invitation Independence

Intuitively, invitations are handled independently from each other, since there is no interaction between invitations. Where the communication protocol exchanges the contents of requests to the same dead drop, the dialing protocol just adds invitations to their corresponding dead drop and let's the clients handle further processing.

Theorem 9 (Invitation Independence)**.** *The Vuvuzela Dialing Protocol handles invitations independently from each other.*

Proof. We start by examining inner server behavior: After receiving all requests of the current round, the inner server decrypts each request individually, generates cover requests and shuffles all requests before sending them to the next server. The last server executes the same steps, but instead of sending them to the next server, he adds the invitations one by one to their corresponding dead drop. Neither at any inner nor at the last server does the handling of a specific request depend on other requests. ■

5.1.2 Request Type Determination

Just as the Vuvuzela communication protocol, the dialing protocol is susceptible against sender timing and replay attacks. Both strategies can be used to determine if a client is inviting another client (i.e., sending an **I**-Type request) or sending a **C**-Type cover request. Attacking by timing the sender works in the same way as with the communication protocol: Because the sender has to make different computations depending on the type of request, the time it takes to send the request differs.

Theorem 10 (Sender Timing)**.** *An adversary $\mathcal{A}$ that is able to time the sending behavior of clients can determine if a given request is **I**-Type or **C**-Type.*

Proof. Similarly to the Vuvuzela Communication Protocol, the sender's behavior differs depending on the type of request that is send:

- When sending an **I**-Type request, the client has to compute the invitation dead drop id and generate the request ciphertext
- When sending an **C**-Type request, the client does not need to compute a dead drop id, but has to generate a random public key

We assume that this differing behavior leads to differing sending times of the request types which an adversary with access to a timer can detect. ■

Replay attacks to determine request types are enabled by the unfortunate choice of requiring all **C**-Type request to be send to a special *no-op* dead drop rather than a random one. Like with the communication protocol, replaying a request reveals the dead drop that it's send to.

Theorem 11 (Replay Attacks I)**.** *An adversary $\mathcal{A}$ with the ability to replay requests can determine if a given request is **I**-Type or **C**-Type.*

Proof. $\mathcal{A}$ replays the request in question and corrupts the last server. Since there is no detection of replayed requests, both the original and replayed request get passed to the last server.

$\mathcal{A}$ observes the invitations that are added to the dead drops. $\mathcal{A}$ determines which dead drop d receives two identical invitations (from the original and replayed request, which both contain the same invitation and go to the same dead drop). If d is the no-op dead drop, the request must have been a **C**-Type request, if not it must have been an **I**-Type request. ■

5.1.3 Sender-Dead Drop Unlinkability

With the Vuvuzela Dialing protocol, determining the type of a given request is actually a special case of linking a sender to a dead drop, since fining out that a request is **C**-Type is equivalent to linking the sender to the *no-op* dead drop.

Replay attacks can also be used to determine the destination dead drop of any request, just as in the communication protocol. Since each dead drop is only used by a fraction of all receivers, the adversary can substantially reduce the receiver anonymity set by linking sender to dead drop:

Theorem 12 (Replay Attacks II)**.** *An adversary $\mathcal{A}$ with the ability to replay requests can narrow the possible clients that a given sender invites to $^1/_m$ of the total number of clients.*

Proof. $\mathcal{A}$ replays the request of the sender (depending on his abilities either at the link between client and first server or at the first server itself) and corrupts the last server.

Since there is no detection of replayed requests, both requests get passed to the last server. There, the adversary can observe two identical invitations being added to the same dead drop[1]. Thus, $\mathcal{A}$ can determine which dead drop the client send his invitation to.

[2, Sec. 5.1] mentions that the adversary knows the mapping from clients to invitation dead drops. The invitation dead drop id for a client is the hash of his public key modulo m. Since the output of the hash function is pseudo-random, each dead drop is mapped to about the same number of clients. If there are n total clients and m dead drops, each dead drop contains invitations for $^n/_m$ clients. ■

If circumstances are favorable for the adversary, he can even identify the receiver of a given invitation unambiguously. Those circumstances can either be prior knowledge of possible receivers (see Corollary 5) or the ability to corrupt the last server (Lemma 1).

Corollary 5 (Receiver Identification)**.** *It follows from Theorem 12 that an adversary can use the replay attack to determine which one of multiple possible receivers a given sender is inviting, if the possible receivers do not have overlapping invitation dead drops. If the possible senders have partially overlapping dead drops, any receivers that do not use the determined dead drop can be excluded.*

Lemma 1 (Receiver Identification II)**.** *An adversary $\mathcal{A}$ with the ability to replay requests and to corrupt the last server can identify the receiver of a given invitation unambiguously.*

Proof. [2, Sec. 5.4] states that the number of invitation dead drops μ is chosen by the last server "for the upcoming rounds", implying some regularity. If the adversary $\mathcal{A}$ corrupts the last server,

[1]which does not occur during normal operation

he can choose μ.

If μ is chosen to be (sufficiently) greater than the total number of clients n, each dead drop is only used by one client with high probability.

$\mathcal{A}$ can identify the dead drop a sender sends his invitation to as described in Theorem 12. Since the mapping from dead drop to receiver is known to the adversary and (due to his choice of μ) is also bijective, he can identify the receiver of the replayed invitation unambiguously. ■

After showing that replay attacks can be used to link senders to dead drops, we show that the adversary cannot use any of his other abilities to do so. We start by showing that $\mathcal{A}$ cannot use any single ability to do so:

- Lemma 2 (Request Alteration) shows that $\mathcal{A}$ cannot link by alteration other than replay
- Lemma 3 (Server Corruption) shows that $\mathcal{A}$ cannot link by corrupting servers
- Corollary 6 (Passive Observation) shows that $\mathcal{A}$ cannot link by passively observing on network links

We then combine these results in Theorem 13 to show that also no combination of abilities can be used.

Remark (On Timing)**.** The adversary's ability to *time* requests is absent in the enumeration above. This is due to the fact that timing can be used to link a sender to a dead drop, *iff* the sender is sending a **C**-Type request[2]. However, timing can *not* not be used to link sender to dead drop if the sender is sending a **I**-Type request; The sender does the same computations regardless of the target dead drop.

Intuitively, other types of request alteration cannot be used to link senders to dead drops since the consequences of alteration are independent from the dead drop the request is send to.

Lemma 2 (Request Alteration)**.** *An adversary $\mathcal{A}$ cannot determine which dead drop a given client sends an invitation to by delaying, dropping or modifying requests.*

Proof. Due to invitation independence (Theorem 9), we only need to analyze the alteration of the request in question.

- If the request is *delayed*, it will cause an access at the dead drop it was intended to reach after it is released.
- If the request is *dropped*, it does not cause an access at the last server.
- If the request is *modified*, it will cause an access at an (unpredictable) other dead drop.

In all three cases, the dead drop access pattern changes *slightly.* We assume that those changes are hidden by the cover requests the (honest) servers add. ■

Intuitively, server corruption does not help the adversary to link sender to dead drop, since inner servers do not have any information about the target dead drop. The last server does necessarily know the target dead drop, but if the adversary corrupts the last server, one inner server has to remain honest. This honest inner server prevents the adversary from knowing which request at the last server belongs to which sender.

Lemma 3 (Server Corruption)**.** *An adversary $\mathcal{A}$ cannot determine which dead drop a given client sends an invitation to by corrupting servers, assuming that at least one server remains honest.*

Proof. When the adversary $\mathcal{A}$ corrupts the server s, he gets access to the inner state of s, all (secret) key material s might have and can observe (and manipulate) the behavior of the server.

If s is an inner server, its inner state consists of

- a key pair (pk_i, sk_i)

[2] **C**-Type requests can be detected by sender timing and are *always* send to the no-op dead drop.

- the number of cover requests added to each dead drop in the current round
- the current permutation π_i
- the public key pk_i^t for each request

None of this information depends on the target dead drop of the request in question. The behavior of an inner server does not depend on the dead drop id of the requests it handles: In each round, it

1. Collects all requests
2. Decrypts the outer onion layer
3. Adds cover requests
4. Shuffles the requests
5. Sends them to the next server

The key material the inner server has only allows him to decrypt the outer onion layer. The dead drop id is always protected by at least one more encryption layer. Thus, $\mathcal{A}$ cannot distinguish by observing inner server behavior or analyzing the state of an inner server.

The inner state of the last server consists of

- a key pair (pk_i, sk_i)
- the number of cover requests added to each dead drop in the current round
- the current permutation π
- the public key pk_i^t for each request
- a mapping from each request to dead drop

Since it adds each request to the corresponding dead drop, its behavior does depend on the target dead drop of the request in question. Evidently, the adversary can find out the target dead drop of any request at the last server. This is not a problem, since we assume that if $\mathcal{A}$ has corrupted the last server, there is at least one honest server proceeding it. This honest server will prohibit $\mathcal{A}$ from being able to link senders to requests at the last server. Thus, $\mathcal{A}$ cannot link senders to dead drops by observing the last server and its inner state.

Lastly, we examine active adversaries that change the behavior of corrupted servers. $\mathcal{A}$ can at most corrupt and control all but one server. Due to the onion decryption and mixing (and addition of cover requests) of all requests, this honest server makes it impossible for the adversary $\mathcal{A}$ to link request from before the honest server to requests after it, independent from what the corrupted servers do[3].

We have shown that neither passively observing nor actively interfering adversaries can map senders to dead drops by corrupting servers. ■

By corrupting servers, the adversary has access to all network links. Thus, Lemma 3 implies:

Corollary 6 (Passive Observation). *An adversary $\mathcal{A}$ cannot determine which dead drop a given client sends an invitation to by passively observing on network links.*

Finally, we can combine our results from Lemmas 2 and 3 and Corollary 6:

Theorem 13 (Sender-Dead Drop Unlinkability). *Let $\mathcal{A}$ be an adversary who can*

1. *corrupt all but one server*
2. *delay, modify and drop arbitrary requests*
3. *passively listen on all network links*

but cannot

1. *corrupt challenge clients*

[3]As an example, a corrupted (first) server could drop all but the request in question. But since the honest server adds cover requests that are indistinguishable from real requests, $\mathcal{A}$ still can tell which request he is interested it after the honest server.

2. time or replay requests.

Such an adversary $\mathcal{A}$ cannot determine which dead drop a given client sends an invitation to.

Proof. We construct a series of hybrid games:

- H_0: No restrictions on $\mathcal{A}$'s abilities
- H_1: $\mathcal{A}$ is not allowed to alter requests
- H_2: H_1 + $\mathcal{A}$ is not allowed to corrupt servers
- H_3: H_2 + $\mathcal{A}$ is not allowed to passively observe on network links

We show that if $\mathcal{A}$ can win H_0, he can also win H_4:

- $H_0 \approx H_1$: Follows with Lemma 2.
- $H_1 \approx H_2$: Follows with Lemma 3.
- $H_2 \approx H_3$: Follows with Corollary 6.

In H_4, the adversary $\mathcal{A}$ is deprived of *all* abilities. If $\mathcal{A}$ can distinguish the request type in game H_4, he can also do so without observing the challenge execution at all, which is impossible. Since $H_0 \approx H_1 \approx \cdots \approx H_4$ holds, we have shown that $\mathcal{A}$ also has no way of distinguishing in game H_0. ■

Corollary 7. *Since **I**-Type and **C**-Type requests only differ in the dead drop they are send to, if follows from Theorem 13 that no adversary who is not able to corrupt senders, corrupt all servers, time or replay requests can determine the type of a request.*

5.2. Analysis

We will analyze the Vuvuzela Dialing Protocol using the proposed framework of [1]. One peculiarity of the dialing protocol is that the "messages" cannot be chosen by the adversary: If Alice sends a message to Bob, the content is predetermined to be an invitation containing Alice's public key, a nonce and a MAC. Thus, challenge senders will always send the message $i = Enc_{pk_a}(pk_b, \text{nonce}, \text{MAC})$.

Client Corruption. We decided not to analyze the dialing protocol under the additional assumption that challenge clients can be corrupted, since there, all notions can be broken trivially: When corrupting the sender of an invitation, the adversary automatically learns the dead drop ids contained in each invitation. When corrupting receivers, the adversary automatically learns the public keys of the inviting senders.

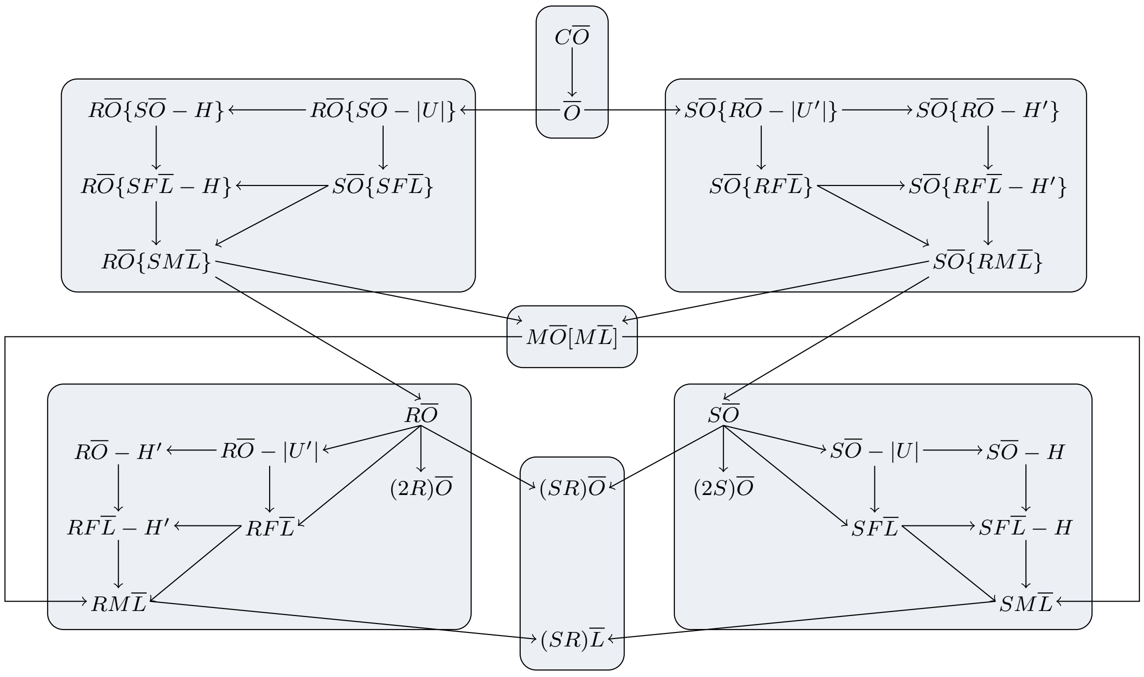
5.2.1 Applicable Notions

Due to the nature of the Vuvuzela Dialing protocol, not all notions of [1] can be applied here. [2, Sec. 5.1] states the following three categories of information that it wants to protect:

1. Which users participated in the protocol each round?
2. What dead drop did some sender add an invitation to?
3. Given a dead drop, how many invitations are in it (since the adversary can link recipients to dead drop IDs)?

Inapplicable Notions. Since the message in the Vuvuzela Dialing protocol is determined by the identities of sender and receiver, it cannot be chosen by the adversary. Thus, notions which allow $\mathcal{A}$ to choose differing messages between the scenarios are not applicable. This includes the following notions:

- Sender-receiver linkability (all notions are equivalent to identical scenarios)
  - $M\overline{O}$
  - $M\overline{O} - |m|$
- Receiver observability (all notions are equivalent to $S\overline{O}$)
  - $S\overline{O}[M\overline{O}]$



$C\overline{O}$
$\overline{O}$
$R\overline{O}\{S\overline{O}-H\}$
$R\overline{O}\{S\overline{O}-|U|\}$
$R\overline{O}\{SF\overline{L}-H\}$
$S\overline{O}\{SF\overline{L}\}$
$R\overline{O}\{SM\overline{L}\}$
$S\overline{O}\{R\overline{O}-|U'|\}$
$S\overline{O}\{R\overline{O}-H'\}$
$S\overline{O}\{RF\overline{L}\}$
$S\overline{O}\{RF\overline{L}-H'\}$
$S\overline{O}\{RM\overline{L}\}$
$M\overline{O}[M\overline{L}]$
$R\overline{O}$
$R\overline{O}-H'$
$R\overline{O}-|U'|$
$(2R)\overline{O}$
$RF\overline{L}-H'$
$RF\overline{L}$
$RM\overline{L}$
$(SR)\overline{O}$
$(SR)\overline{L}$
$S\overline{O}$
$(2S)\overline{O}$
$S\overline{O}-|U|$
$S\overline{O}-H$
$SF\overline{L}$
$SF\overline{L}-H$
$SM\overline{L}$

Figure 5.1: The hierarchy of remaining notions.

  - $S\overline{O}[M\overline{O}-|m|]$
  - $(SM)\overline{O}$
  - $(SM)\overline{L}$
- Sender observability (all notions are equivalent to $R\overline{O}$)
  - $R\overline{O}[M\overline{O}]$
  - $R\overline{O}[M\overline{O}-|m|]$
  - $(RM)\overline{O}$
  - $(RM)\overline{L}$

Further, the *Message Partitioning per Sender/Receiver* property is not of interest here. All notions containing P or P' are not applicable.

An overview of the remaining notions can be found in Figure 5.1.

5.2.2 Replay Attacks

Message Observability

We show an attack of $\mathcal{A}_{CM}$ against $(SR)\overline{L}$. The adversary chooses the following scenarios:

Table 5.1: I_x denotes Instance x. Black requests are **I**-Type requests submitted by the adversary, gray requests are **C**-Type.

	Scenario 0	Scenario 1
I_0	$(alice, \mathbf{bob}, i, aux)$	$(alice, \mathbf{dave}, i, aux)$
	$(carol, \mathbf{dave}, i, aux)$	$(carol, \mathbf{bob}, i, aux)$
	$(bob, rand, i, aux)$	$(bob, rand, i, aux)$
	$(dave, rand, i, aux)$	$(dave, rand, i, aux)$
I_1	$(carol, \mathbf{dave}, i, aux)$	$(carol, \mathbf{bob}, i, aux)$
	$(alice, \mathbf{bob}, i, aux)$	$(alice, \mathbf{dave}, i, aux)$
	$(bob, rand, i, aux)$	$(bob, rand, i, aux)$
	$(dave, rand, i, aux)$	$(dave, rand, i, aux)$

The Attack. We make use of Corollary 5: $\mathcal{A}_{CM}$ chooses Bob and Dave in such a way that they don't share the same invitation dead drop. This is trivial if $\mathcal{A}_{CM}$ can modify the number of dead drops μ as described in Lemma 1. Let Bob use dead drop d_B and Dave dead drop d_D. $\mathcal{A}_{CM}$ corrupts the first and last servers and replays Alice's request.

He observes the dead drop accesses at the last server. If the same invitation is added to dead drop d_B twice, he determines that Alice is inviting Bob and returns $b' = 0$. If the same invitation is added to dead drop d_D twice, Alice is inviting Dave and he returns $b' = 1$.

Adversary Advantage. With this attack, $\mathcal{A}_{CM}$ always wins.

Sender-Receiver Unobservability. The notion $(SR)\overline{O}$ can be attacked analogously. Here, the adversary can only win in one scenario, which still gives him a non-negligible advantage.

Twice Sender/Receiver Linkability

$(2S)\overline{L}$ and $(2R)\overline{L}$ can be attacked by $\mathcal{A}_{CM}$ and $\mathcal{A}_{GNM}$ using Theorem 11. We show the attack of $\mathcal{A}_{CM}$ against $(2S)\overline{L}$.

The adversary chooses the following scenarios:

Table 5.2: I_x denotes Instance x, S_x denotes stage x. Black requests are **I**-Type requests submitted by the adversary, gray requests are **C**-Type.

		Scenario 0	Scenario 1
I_0	S_0	$(alice, bob, i, aux)$	$(alice, bob, i, aux)$
		$(bob, rand, i, aux)$	$(bob, rand, i, aux)$
		$(carol, rand, i, aux)$	$(carol, rand, i, aux)$
	S_1	$(\mathbf{alice}, bob, i, aux)$	$(\mathbf{carol}, bob, i, aux)$
		$(bob, rand, i, aux)$	$(bob, rand, i, aux)$
		$(carol, rand, i, aux)$	$(alice, rand, i, aux)$
I_1	S_0	$(carol, bob, i, aux)$	$(carol, bob, i, aux)$
		$(bob, rand, i, aux)$	$(bob, rand, i, aux)$
		$(alice, rand, i, aux)$	$(alice, rand, i, aux)$
	S_1	$(\mathbf{carol}, bob, i, aux)$	$(\mathbf{alice}, bob, i, aux)$
		$(bob, rand, i, aux)$	$(bob, rand, i, aux)$
		$(alice, rand, i, aux)$	$(carol, rand, i, aux)$

The Attack. $\mathcal{A}_{CM}$ determines the type of request Alice sends in stage 0 as described in Theorem 11. If Alice sends an **I**-Type request, he also determines the type of request she sends in stage 1.

- If Alice sends an **I**-Type request in stage 1, he returns $b' = 0$
- If Alice sends an **C**-Type request in stage 1, he returns $b' = 1$

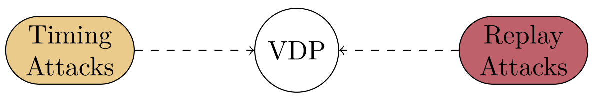
If Alice sends a **C**-Type request in stage 0, he knows that instance 1 was chosen and checks Carol's request in stage 1.

- If Carol sends an **I**-Type request in stage 1, he returns $b' = 0$
- If Carol sends an **C**-Type request in stage 1, he returns $b' = 1$

Adversary Advantage. With this attack, $\mathcal{A}_{CM}$ always wins.

Twice Receiver Linkability. $(2R)\overline{L}$ can be broken similarly to $(SR)\overline{L}$ using Theorem 11. Here, the challenge receivers Bob and Dave have to use different invitation dead drops.

During the first stage, the adversary checks if Alice is inviting Bob (instance 0). If so, he checks if she is also inviting Bob in stage 1 (and returns $b' = 0$ if that's the case). If not, he checks if Alice



🟨	$\mathcal{A}_{CT}$ and $\mathcal{A}_{GT}$ break $S\overline{O}-H$ and $(2S)\overline{L}$.
🟥	$\mathcal{A}_{CM}$ and $\mathcal{A}_{GNM}$ break *any* notion.

Figure 5.2: Summary of the analysis of the Vuvuzela Dialing protocol

is inviting Dave in stage 1 (and returns $b' = 0$ if that's the case).

With this strategy, he always wins the $(2R)\overline{L}$-game.

5.2.3 Timing Attacks

Timing attacks as described in Theorem 10 require the distributions of **I**-Type versus **C**-Type requests to be different between the scenarios. The weakest notions that allow this are $S\overline{O}-H$[4] and $(2S)\overline{L}$. $SF\overline{L}$ explicitly restricts senders/receivers to be the same in both scenarios and therefore cannot be attacked by timing senders.

We present the attack of $\mathcal{A}_{GT}$ against $S\overline{O}-H$, $(2S)\overline{L}$ can be attacked analogously.

The adversary chooses the following scenarios:

Table 5.3: Black requests are **I**-Type requests submitted by the adversary, gray requests are **C**-Type.

Scenario 0	Scenario 1
$(\textbf{alice}, bob, i, aux)$	$(\textbf{carol}, bob, i, aux)$
$(bob, rand, i, aux)$	$(bob, rand, i, aux)$
$(\textbf{carol}, rand, i, aux)$	$(\textbf{alice}, rand, i, aux)$

The Attack. The adversary $\mathcal{A}_{GT}$ times the sending behavior of Alice to determine if she sends an **I**-Type request (return $b' = 0$) or a **C**-Type request (return $b' = 1$).

5.3. Summary

We analyzed the Vuvuzela Dialing protocol with a reduced set of notions compared to the Communication protocol (see Figure 5.1) and found two main categories of attacks: Replay attacks and timing attacks.

With replay attacks, $\mathcal{A}_{CM}$ and $\mathcal{A}_{GNM}$ can break *any* notion.

With timing attacks, $\mathcal{A}_{CT}$ and $\mathcal{A}_{GT}$ can break $S\overline{O}-H$ and $(2S)\overline{L}$.

A visual summary of the results can be found in Figure 5.2

5.4. Discussion of the Vuvuzela Dialing Protocol

The Vuvuzela Dialing protocol has the same two main weaknesses as the communication protocol (replay and sender timing).

Replay Attacks. Replay attacks in the communication protocol require the adversary to replay both requests of two suspected communication partners to determine whether they are indeed communicating with each other or not. In the dialing protocol, the adversary only needs to replay

[4]$S\overline{O}-P$ is not part of the considered notions, see Figure 5.1.

one request to find out which dead drop the request is send to. This by itself would not be a huge problem, but an integral property of the dialing protocol is that the mapping from users to dead drop is publicly known. We can imagine the following real world attack:

An oppressive government suspects the activist Alice to either wanting to talk to Bob or Dave. Conveniently for them, Bob and Dave use different invitation dead drops. They can replay Alice's invitation and observe at which dead drop it arrives and therefore determine who the probable receiver is[5].

To counteract replay attacks, mechanisms have to be put in place to detect and discard replayed requests.

Number of Invitation Dead Drops. The dialing protocol has a set number μ of invitation dead drops, which is chosen by the last server in regular intervals. A concrete μ is a compromise between client privacy and performance: The smaller μ is, the more clients share each dead drop. On one hand, the more clients share a dead drop, the less information an adversary can learn about the receiver of an invitation by linking senders to dead drops. On the other hand, if more clients share one dead drop, this dead drop will contain more invitations that have to be decrypted by each client every time they fetch invitations from the dead drop.

A big problem with μ is that it is chosen by the last server without supervision. If the adversary corrupts the last server, he can set μ himself. If μ is set sufficiently larger than the total number of clients, each client would have his own dead drop with high probability. In this case, replay attacks as mentioned above could unambiguously identify receivers, which is a huge problem. Without protection of this crucial information, using the dialing protocol would be equivalent to inviting users directly without any privacy protection.

To solve to this problem, μ can be made immutable. A drawback with this approach is that the system is not able to adapt to changing client numbers. Two problems arise:

1. Performance suffers if many clients join and dead drops get more concurrent users.
2. Privacy suffers if many clients disconnect form the network, since the remaining clients continue using the emptying dead drops

Client Cover. Another problem of the dialing protocol is that real invitations are not indistinguishable from cover invitations. An adversary can either use a sender timing attack or a replay attack to detect whether a given client is active or not. This also has clear real world implications:

For the oppressive government, the fact that Alice is actively inviting people is probably reason enough to question her.

The replay variant of the cover request detection can be counteracted as mentioned above. The timing variant be dealt with by adjusting the client protocol to delay the sending time by a random interval big enough to hide any differences. Another sensible change would be to eliminate the no-op dead drop and to have clients send their cover requests to a random dead drop. The resulting increase in load on the dead drops can be offset by sending less server cover.

[5]The invitation could also be meant for any other user of this dead drop, but if they are certain that Alice only wants to talk to Bob or Dave, this is not a problem.

6. Analyzing Vuvuzela with greater Scenario Variation

Without Assumption 5, the adversary can have an arbitrary number of differing communications between scenarios (if allowed by the notion). Thus, the adversary can adjust his challenges to the point where the existing server cover (which is meant to hide one differing communication) is clearly inadequate. We simplify our following analysis by assuming that there is *no* server cover.

What does the Server Cover Traffic hide? The server cover traffic is meant to hide *dead drop access patterns*, i.e., how many dead drops are accessed and which proportion of them is accessed once versus twice. Dead drop access patterns can be observed by any adversary that is able to corrupt the last server (e.g, $\mathcal{A}_C$ or $\mathcal{A}_{GN}$). If dead drop access patterns differ between scenarios and server cover traffic is not present, such an adversary can learn additional information that enables him to distinguish scenarios.

Outside of the last server, the cover traffic obscures the number of requests send over network links and processed on the inner servers (except on the link between client and first server). Since the total number of requests is identical in both scenarios in all notions under all client cover assumptions, removing the server cover does not leak usable information at this point.

6.1. Weak Client Cover Assumption

Observing dead drop access patterns requires the adversary to be able to corrupt the last server.

Theorem 14 (Pattern Unobservability)**.** *As long as no empty communications are allowed (i.e., in all notions but $C\overline{O}$), dead drop access patterns are identical in both scenarios under the Weak Client Cover Assumption (Assumption 9) assuming no request alteration by the adversary.*

Proof. Assumption 8 states that each client sends the same number of requests in both scenarios. Combined with the fact that the same clients are connected to the service in both scenarios, it follows that the total number of requests is the same in both scenarios. What remains to be shown is that the distribution of single versus double dead drop accesses is also identical.

Every challenge communication the adversary submits causes a double dead drop access. Since there are no empty communications allowed, he has to submit the same number of communications in both scenarios. Thus, the number of double dead drop accesses is equal in both scenarios. We have already shown that the total number of dead drop accesses is equal. Therefore, the number of single dead drop accesses (total - double) is also equal and with that, the dead drop access pattern is identical. ■

Corollary 8. *It follows from Theorem 14, that the number of requests send over any network link are identical for* all *notions in both scenarios under the Weak Client Cover Assumption without server cover traffic.*

Lemma 4 (Communication Unobservability)**.** *$\mathcal{A}_C$ and $\mathcal{A}_{GN}$ can break $C\overline{O}$ under the Weak Client Cover Assumption without server cover traffic.*

Proof. We show that $\mathcal{A}_C$ can break $C\overline{O}$ by corrupting the last server and observing dead drop access patterns. The adversary chooses the following scenarios:

Scenario 0	Scenario 1
$(alice, bob, m, aux)$	$(alice, rand, 0, aux)$
$(bob, alice, 0, aux)$	$(bob, rand, 0, aux)$

$\mathcal{A}$ observes a double dead drop access in scenario 0 and two single dead drop accesses in scenario 1. He can recognize this difference and can therefore *always* distinguish scenarios[1]. $\mathcal{A}_{GN}$ can attack with the same strategy. ■

In Section 4.2, we have shown that $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ cannot break the $C\overline{O}$-notion assuming that there is enough cover traffic. This implies that they can only win $C\overline{O}$ here by observing dead drop access patterns. Thus:

Corollary 9. *It follows from Theorem 14, that $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ cannot break the $\overline{O}$ notion under the Weak Client Cover Assumption without server cover traffic.*

6.1.1 Other Adversaries

In Section 4.2, we have shown that $\mathcal{A}_{CM}$ and $\mathcal{A}_{GNM}$ can break any notion except $M\overline{O} - |m|$. The same is true here, $M\overline{O} - |m|$ cannot be broken, since the clients in active conversation with each other are identical in both scenarios; Without alteration, the dead drop access pattern is identical (as shown in Theorem 14). Alteration to requests have the same consequences to the dead drop access pattern in both scenarios: If Alice and Bob are communicating with each other in scenario 0, they are also communicating in scenario 1.

- *Dropping* Alice's request will cause Bob's request to be a single dead drop access in both scenarios
- *Modifying* Alice's request will cause Bob's and Alice's request to be a single dead drop access in both scenarios
- *Delaying* Alice's request (until a later round) will cause Bob's and Alice's request to both be a single dead drop accesses in both scenarios
- *Replaying* Alice's request (in the same round) will cause a triple dead drop access in both scenarios, if replayed in a later round, there will be a double dead drop access in the current round and a single access by the replayed request in the later round in both scenarios

$\mathcal{A}_{CT}$ and $\mathcal{A}_{GT}$ have been shown to be able to break any notion except $M\overline{O} - |m|$ and $(SR)\overline{L}$. Since they lack the ability to alter requests, Theorem 14 and Corollary 8 can be used to see that they do not gain additional capabilities here.

6.2. Medium Client Cover Assumption

First, we note that Theorem 14 also holds under the Medium Client Cover Assumption (Assumption 10). This is due to the fact that all attacks that are possible under the Medium Client Cover Assumption also work under the weak variant.

The changes are analogous to the previous section: $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ gain the ability to break $C\overline{O}$, other adversaries do not gain any abilities: $\mathcal{A}_{CM}$ and $\mathcal{A}_{GNM}$ can break $(SR)\overline{L}$ and $(SR)\overline{O}$, $\mathcal{A}_{GT}$ and $\mathcal{A}_{CT}$ can break $S\overline{O} - P$.

6.3. Strong Client Cover

Our results do not change under the *Strong Client Cover Assumption* (Assumption 11): No adversary can break the $C\overline{O}$ notion.

This follows from the fact that if the clients in active conversation with each other are identical in both scenarios, the dead drop access pattern (and number of requests on links) is also the same

[1]The difference can be made even more pronounced by adding more communications to scenario 0. In the extreme case, all clients would be in active conversation in scenario 0 and no one in scenario 1.

in both scenarios. Thus, the adversary cannot learn any additional information that helps him distinguish scenarios from observing dead drop access patterns, even if the server cover traffic is removed.

This is not a very surprising result, since the Strong Client Cover Assumption already requires the active communications and therefore the dead drop access patterns to be identical in both scenarios. Adding or removing the server cover traffic makes no difference.

6.4. Dialing Protocol

Due to Assumption 16, Theorem 14 also holds with the dialing protocol: As long as no empty communications are allowed, the dead drop access pattern is identical in both scenarios. The results are therefore identical: $\mathcal{A}_C$ and $\mathcal{A}_{GN}$ gain the ability to break $C\overline{O}$.

The rest of the results remains unchanged: $\mathcal{A}_{CM}$ and $\mathcal{A}_{GNM}$ can break any notion, $\mathcal{A}_{CT}$ and $\mathcal{A}_{GT}$ can break any notion except $S\overline{O} - H$ and $(2S)\overline{L}$.

6.5. Summary

Under the Strong Client Cover Assumption, dead drop access patterns are by definition always identical in both scenarios. This prohibits any additional attacks, even if server cover traffic is ignored.

Under the Medium and Weak Client Cover Assumptions and in the dialing protocol, dead drop access patterns are identical in both scenarios as long as no empty communications are allowed. Consequently, adversaries that can corrupt the last server (e.g., $\mathcal{A}_C$) can break $C\overline{O}$ (which is the only notion that allows empty communications).

7. Analysis of Alpenhorn

Alpenhorn [15] is a protocol by the the same authors as Vuvuzela. Its goal is to provide two services:

1. A system that allows users to initiate conversations (similar to the Vuvuzela Dialing Protocol)
2. A mechanism for clients to have evolving shared secrets

Alpenhorn does not enable communication between clients on its own, but rather is expected to be used in conjunction with a communication protocol. We analyze the effects of integrating Alpenhorn into Vuvuzela.

7.1. Background

We present a summarized version of the Alpenhorn protocol, leaving out details that are not relevant to the discovered attacks. This includes the acquisition of secret keys from the PKG servers and the use of digital signatures.

The Alpenhorn Dialing Protocol has the same network structure as the Vuvuzela equivalent, but is extended by a number of private key generators (PKG), which can be seen in Figure 7.1. The PKGs are used to enable an *identity-based encryption* service:

Definition 5 (Identity-Based Encryption)**.** An identity-based encryption (IBE) scheme provides three functions:

- $Enc(pk_M, id, m) \rightarrow c$: Generate a ciphertext on input of the master public key pk_M, an identity id and a message m
- $Dec(sk_{id}, c) \rightarrow m$: Decrypt the given ciphertext c with the user secret key sk_{id}
- $Extract(sk_M, id) \rightarrow sk_{id}$: Generate the corresponding user secret key sk_{id} for a given identity id on input of the master secret key sk_M

The client proves his identity to the PKG server (who is in possession of sk_M) and receives his user secret key sk_{id} in return. Encryption can happen without direct involvement of the PKG server, since the master public key pk_M is known to all clients.

Adversary Model. The use of IBE with the addition of n PKG servers alters the adversary model: In Alpenhorn, the adversary can not only corrupt all but one mixnet servers and monitor, block, delay and inject traffic at any network link, but also corrupt all but one PKG servers.

Normally, IBE only uses one PKG server and requires total trust in it, since the server can compute and therefore leak secret keys of all users. Alpenhorn mitigates this issue by distributing the secret information among the n PKG servers. Each server has its own master public key pk_M^i (and master secret key sk_M^i), clients have a secret key sk_{id}^i from each server. For encryption and decryption, the sum of the keys is used:

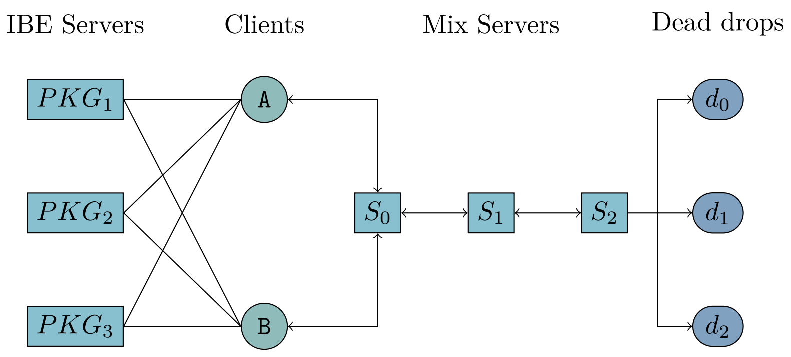


Figure 7.1: Alpenhorn network diagram

$$Enc\left(\sum_{i=1}^{n} pk_M^i, id, m\right) \rightarrow c$$
$$Dec\left(\sum_{i=1}^{n} sk_{id}^i, c\right) \rightarrow m$$

As long as one (arbitrary) PKG server is honest, the secret key $sk_{id} = \sum_{i=1}^{n} sk_{id}^i$ cannot be assembled by the adversary without corrupting the client.

Inviting Clients. The general invitation process is the same as in Vuvuzela: Clients send invitations identifying them to a predetermined invitation dead drop ($d = H(id_b) \mod \mu$) of the receiver through the server chain; The receiver downloads the invitations directly from his dead drop and tries to decrypt them to find out which are meant for him.

The first major difference is that sender and receiver cannot start communicating directly after decrypting the request; First the receiver has to send an own invitation back to the sender. This is due to the fact that random keys are used to compute their initial shared keys: Alice chooses a random (pk_A^{dh}, sk_A^{dh}) key pair and sends pk_A^{dh} to Bob. Since Alice does not know Bob's equivalent key, Bob has to also choose (pk_B^{dh}, sk_B^{dh}) at random and send an invitation with pk_B^{dh} back to Alice. They can then use Diffie-Hellmann to compute a shared key k_{ab}.

The second major difference is the use of identity-based encryption for the requests. Alice knows Bob's identity (`bob@kit.edu`) and can use it to encrypt her request to him directly (see Definition 5). The onion layers are encrypted with conventional cryptography, in the same way as Vuvuzela.

Shared Keys. In Vuvuzela, the shared key k_{ab} between clients is long-lived. Alpenhorn aims to provide *forward secrecy*[1], which means that k_{ab} has to be evolved over time. This is done with a cryptographic hash function H:

$$k_{ab}^{r+1} = H(k_{ab}^r)$$

k_{ab}^r is the key used by Alice and Bob in round r, k_{ab}^{r+1} is used in round $r + 1$.

7.2. Attacking the Alpenhorn Dialing Protocol

7.2.1 Replay Attacks

According to [15, Sec. 6], Alpenhorn uses the same mixnet design as Vuvuzela. Consequently, replayed requests are also forwarded through the server chain and added to their corresponding

[1]i.e., if a client is compromised, the adversary should not be able to learn past information

dead drop. The last mixnet server can be corrupted to observe where two identical requests are added. The number of invitation dead drops μ can be controlled by the adversary to lower the number of users that share the same dead drop.

All attacks on the Vuvuzela Dialing Protocol described in Section 5.2.2 still work with Alpenhorn: $\mathcal{A}_{CM}$ and $\mathcal{A}_{GNM}$ break *any* notion by corrupting the last server and replaying request to find out which dead drop the are send to.

7.2.2 Timing Attacks

With the Vuvuzela Dialing Protocol, sender timing attacks can be used to determine if the sender sends a real invitation or cover. This is also the case with Alpenhorn and the timing difference between the two cases is expected to be even greater:

- If the client wants to send a cover invitation he just sets the request to $(\bar{d}, 0^\ell)$, where $\bar{d}$ is the dedicated cover dead drop and ℓ is the length of a normal invitation.
- If the client wants to send a real invitation, he has to choose a random key pair, find out the receivers dead drop id, compute signatures and IBE-encrypt the invitation.

It follows that all attacks from Section 5.2.3 also work with the Alpenhorn Dialing Protocol: $\mathcal{A}_{CT}$ and $\mathcal{A}_{GT}$ break $S\overline{O} - H$ and $(2S)\overline{L}$ by timing the sender and distinguishing **I**-Type requests from **C**-Type Requests.

7.3. Attacking the Vuvuzela Communication Protocol with Alpenhorn's Improvements

If Alpenhorn is combined with the Vuvuzela Communication Protocol, the long-term shared keys k_{ab} are replaced by a evolving key k^r_{ab}.

None of the attacks (and proofs) in Sections 4.2 to 4.4 rely on the shared key k_{ab}, ergo our results from these sections do not change.

The *inner key attacks* in Section 4.5.1 do rely on k_{ab}: The adversary corrupts the sender during an identification round where the receiver of a message is known and observes the inner key which is used to encrypt this message. During later rounds, the receiver can still be identified, because the same k_{ab} is used.

These attacks do not work anymore as described, since the k^r_{ab} that was observed during the identification round is not equal to the keys used in later rounds. But the adversary can modify his behavior: Since he has corrupted the client, he can observe the evolution of the key round by round and therefore regularly update the keys he needs to be on the lookout for in later rounds. After this modification, all attacks described in Section 4.5.1 can be executed successfully.

The attacks in Section 4.5.2 also rely on k_{ab}: When combined with an deterministic encryption scheme, the long-lived key allows the adversary to identify if identical messages are exchanged at the dead drops multiple times. This problem is solved by evolving k_{ab}; Now identical messages do not generate identical ciphertexts when send over multiple rounds. The attacks from Section 4.5.2 are not valid anymore. A remaining problem is the use of the same key for sender and receiver: If Alice and Bob send the same message m to each other, their requests both contain the identical ciphertext $Enc_{k^r_{ab}}(m)$. During normal conversation, it is not very likely for clients to exchange identical messages. But if the clients have no real message to send, they both send a **AC**-Type request containing **0** as message. Thus, the proportion between exchanges of identical versus differing ciphertexts reveals information about the current network activity[2] to the adversary. This problem can be solved by having different keys for each communication partner. In that case, the results would be equivalent to using a non-deterministic encryption scheme.

[2]i.e., how many clients are exchanging **AC**-Type requests versus **M**-Type requests.

7.4. Summary

The Alpenhorn Dialing Protocol is susceptible to the same attacks as the Vuvuzela Dialing Protocol. Sender timing differs even greater depending on the type of invitation send.

Integrating Alpenhorn into the Vuvuzela Communication Protocol does not have any positive effect on privacy protection under standard assumptions. It does however prevent attacks if a deterministic encryption scheme is used.

7.5. Discussion

As we have shown, adding Alpenhorn to Vuvuzela does not improve privacy protection (as long as a non-deterministic encryption scheme is used). Nevertheless, it comes with some cost:

First of all, the computational cost for the clients is significantly higher due to the IBE encryption and key evolution.

Secondly, there is an increase in required trust: Clients not only have to trust their communication partner and one mixnet server, but also one PKG server.

The perceived benefit of IBE is the ability to use (easy to know) identities instead of public keys for encryption. Having said that, an adversary could present a plausible but fake identity (e.g., `bob@kit.com` instead of `bob@kit.edu`) to Alice. If Alice and Bob need a way to verify their identifiers either way (e.g., in a real-life meeting), they could just verify public keys at this occasion and save themselves the overhead and additional trust of identity-based encryption.

8. Fixing Vuvuzela

In Section 4.4, we have shown that Vuvuzela reaches Communication Unobservability against any adversary that is not able to corrupt clients under the *Strong* Client Cover Assumption. In this section, we propose a fixed version of Vuvuzela (f-Vuvuzela) that reaches Communication Unobservability against any adversary (without client corruption) under the *Weak* Client Cover Assumption.

We found two main weaknesses of the Vuvuzela protocols: The possibility of *replay attacks*, which we fix in Section 8.1 and the *sender timing* weakness, which we mitigate in Section 8.2. In Section 8.3, we evaluate how much server cover traffic is needed to allow for greater changes in communication patterns between scenarios. We then analyze the updated protocol again in Section 8.4 using the framework from [1] to make sure that our fixes have the intended effect and that no further problems arise.

8.1. Replay

To fix Vuvuzela's weakness against replay attacks, replayed requests have to be detected and discarded. There are two important observations to make:

1. **Replays need to be detected over *all* rounds.** The double replay attacks presented previously can be modified to be executed over multiple rounds: The adversary can save the request of the two clients he suspects to be communicating with each other and corrupt the last server to observe which dead drops are (doubly) accessed in this round. He replays the requests in a later round. If a dead drop id from the first round is used again, it is highly likely that the two replayed request caused this double access and therefore the two clients are indeed communicating with each other.
2. **Replay detection needs to happen at *every* inner server.** The adversary can corrupt $n - 1$ servers in total and has to corrupt the last server in order to execute the replay attack. Corrupted servers will not discard the replayed requests, which leaves it up to the one remaining honest inner server to do so. Since the honest server can be at an arbitrary position, all (inner) servers must be equipped to detect replays.

A naive way of detecting replays would be to save all requests (or sufficiently long hashes of the requests) ever received at each server and compare new incoming request to this list. While this would work, it scales rather poorly, both in terms of computation time and required space, with growing numbers of users, servers and rounds.

To improve efficiency, we split the problem into two parts:

First, we add a mechanism to detect replays in the current round. This is an easy problem for each server, since the server has access to all requests of the current round at once. He can (previous to adding the cover requests) check the list of request for bit-identical copies and deduplicate it. This can be done with linear complexity in the number of requests (see Section 15.5).

Second, requests from previous rounds need to be detected. We suggest that the client adds the current round number r to the request, so that the server can check it. If the included r is smaller the real round number, the request must have been generated in a previous round and needs to be discarded.

It needs to be ensured that the adversary cannot modify the round number without changing the rest of the request and that each server can check the round number. Both can be achieved by adding the round number r inside of each onion-layer: On one hand, each server can unpack the outer layer to find a fresh r to check. On the other hand, the round number for the next server to compare is always (in previous servers and on network links) protected by an encryption layer. If the adversary tries to modify the ciphertext to change r, he will also change the rest of the content with high probability, because of the *diffusion* property of the encryption scheme (see Assumption 4).

Remark (Round Synchronization)**.** Our solution requires that all clients (and servers) agree on what the current round number r is. To achieve this synchronization, we propose that that starting time of the first round t_0 is hard-coded into the Vuvuzela protocol. We also propose for the rounds to have a fixed length t_r[1] Given a current time t_c, each client could compute the current round number r as

$$r = \left\lfloor \frac{t_c - t_0}{t_r} \right\rfloor$$

To make sure that the servers have enough time to process requests during one round, the client can't be allowed to send his request at any time. Instead, a sending interval at the beginning of the round has to be defined. After the interval has ended, the first server will start processing the received requests and will not accept any further ones. If a user wants to send a message after this interval, he has to wait for the beginning of the next round.

One slight disadvantage of this approach is that the clocks clients and servers need to be *somewhat* synchronized: If a client's clock is too slow, all his (valid) requests might be detected as being from previous rounds and discarded. If the client's clock is too fast, his requests might still be able to be replayed in later rounds. The second issue can be mitigated by also discarding requests with round numbers *greater* than the real r[2]. We classify this disadvantage only as slight, since rounds have a length on the order of minutes, normal operating system level time synchronization with an external server should be sufficient.

An adversary might be able to inject a wrong time into a non-corrupted client. It is reasonable to ignore this possibility in further analysis, since the clients only need to have *approximately* the same time. Users can be expected to notice if their computer's clock is off by a substantial amount.

The improved replay protection of our alternative approach comes at a (slight) cost of added computational load at both clients and servers to compute and verify the round numbers. The size of reach request is increased by $|r|$ times the number of servers, which adds network load and required memory capacity at the servers.

We present a revised inner server protocol in Algorithm 1. A revised client protocol is presented in Algorithm 2, after additional modifications to prevent timing-based attacks are explained.

8.2. Timing

To fix the sender timing issues, the time it takes to send an **IC**-Type request has to be made identical to the time it takes to send an **M/AC**-Type request. Since the **IC**-Type request takes longer to send, one could add a delay before sending **M/AC**-Type requests.

Determining the right amount of delay to a high degree of precision is not an easy task, since it depends on many factors (e.g., computations can take different times depending on architecture and version of the client's CPU). A better way of preventing timing attacks is to have the client make identical computations in both cases.

[1]The Vuvuzela paper states that the first server is responsible for coordinating the rounds by announcing the start of a new round to clients and servers, implying that rounds could have differing lengths.

[2]With this approach, the replayed message might still reach the dead drop, but the original request is discarded, the problem therefore solved. The affected client is not able to communicated in this case.

Algorithm 1 f-VCP: Inner server protocol

```
1: ▷ Changes preventing replay attacks are highlighted in blue
2: function DecryptRequest($req = (pk_i^t, r, c)$)                    ▷ No changes
3: function GenerateCover                                               ▷ No changes
4: function ProcessRound
5:     collect requests
6:     for all $req \in$ requests do
7:         $req = (pk_i^t, r, c) \leftarrow$ DecryptRequest($req$)
8:         if $r \neq$ current round number then
9:             discard $req$                    ▷ prevent replay from previous round
10:    Deduplicate(requests)                    ▷ prevent replay from current round
11:    GenerateCover
12:    choose random permutation $\pi_i$ and shuffle requests
13:    send requests to next server & wait for response
14:    reorder requests according to $\pi_i^{-1}$
15:    remove cover
16:    build up onion layer
17:    send to previous server           ▷ or in case of the first server to the clients
```

Client Communication Protocol. This is simple enough with the client communication protocol: Instead of choosing a random public key only if a **IC**-Type request is send, it is chosen in both cases. The public key used to compute k_{ab} is then either set to the communication partner's public key or the random key. The rest of the client protocol can be left unchanged. Pseudocode for the revised client protocol (including changes to prevent replay attacks) can be found in Algorithm 2.

Algorithm 2 f-VCP: Client Protocol.

```
1: ▷ Changes preventing replay attacks are highlighted in blue
2: ▷ Changes preventing timing attacks are highlighted in red
3: function ConstructRequest
4:     choose random $pk_z$
5:     if client has no partner then
6:         $pk_x = pk_z$
7:     else
8:         $pk_x = pk_b$
9:     compute shared key $k_{ab} \leftarrow DH(sk_a, pk_x)$
10:    compute dead drop id $d_{ab}^r \leftarrow H(k_{ab} \mid r)$
11:    if client has message $m$ to send then
12:        pad message $m$ if necessary
13:    else
14:        $m = \mathbf{0}$
15:    return $req = (d_{ab}^r, r, Enc_{k_{ab}}(m))$     ▷ round number $r$ is added to request
16: function OnionWrap(req)
17:    for $i \in$ Servers.reverse do
18:        choose random temporary keys $pk_i^t, sk_i^t$
19:        compute shared key $k_{ai} \leftarrow DH(sk_i^t, pk_i)$
20:        $req = (pk_i^t, r, Enc_{k_{ai}}(req))$        ▷ round number $r$ is added to request
21:    return $req$
22: function Main                                                       ▷ No Changes
```

Client Dialing Protocol. The client dialing protocol is changed analogously: A random public key is always chosen, the public key used for the request is then either set to the random key or to the communications partner's public key (if existing). If the random public key belongs to a real user, a unintentional invitation could be send, but the chance of this happening is negligible. Requests are then able to be constructed in the same way independent from the request type.

This has the additional advantage of removing the no-op dead drop $\bar{d}$ where **C**-Type requests where previously send to. While this adds overhead to the clients (depending on the number of dead drops and the amount of cover), it obfuscates the amount of cover send. With out this alteration, adversaries could determine network activity by observing the number of invitations send to dead drop $\bar{d}$.

We present a revised client dialing protocol in Algorithm 3.

Algorithm 3 f-VDP: Client Protocol

```
1: ▷ Changes preventing replay attacks are highlighted in blue
2: ▷ Changes preventing timing attacks are highlighted in red
3: function ConstructRequest
4:     choose random $pk_z$
5:     if client has no partner then
6:         $pk_x = pk_z$
7:     else
8:         $pk_x = pk_b$
9:     compute dead drop id $d \leftarrow H(pk_x) \mod \mu$
10:    compute nonce, MAC
11:    $c = Enc_{pk_x}(pk_a, \text{nonce}, \text{MAC})$
12:    return $req = (d, r, c)$                    ▷ round number $r$ is added to request
13:    function Main          ▷ round number $r$ is added to onion layers, see Algorithm 2
```

Last Server Communication Protocol. The last server also shows differing behavior in the communication protocol depending on request type, which could lead to timing differences: If the request in question is **M/AC**-Type it has a partner which is handled at the same time, if it is **IC**-Type, it is handled by itself. Handling two requests at once takes less time than handling two requests separately, which could lead to the request being sent back sooner.

To equalize the behavior, we introduce two arrays: rc (request content) contains the content of the requests, i.e., if the ith request is $req_i = (d_i, c_i)$, $rc[i]$ is set to c_i. $rc[-1]$ is set to a random value. The array rp (request partners) contains the identifiers for a request's partner: If req_i and req_j share the same dead drop id (i.e., belong to the same communication), $rp[i]$ is set to j and $rp[j]$ to i. If req_i does not have a partner request, $rp[i]$ is set to -1. After the request partner array is filled, the request content can be modified. Each requests content is changed to the content of its partner:

$$req_i \leftarrow (d_i, rc[rp[i]])$$

If the request has no real partner, its partner values is -1 and it receives the random value $rc[-1]$ as c_i. Unlike the original protocol, all cover requests from the last server have the same random value. This is not a problem, since the value is (outside of the last server and client) always hidden by encryption layers. We present a fixed variant of the last server protocol in Algorithm 4.

The last server dialing protocol does not need to be changed, since the last server cannot distinguish between request types due to the modification of the client protocol.

8.3. Dead Drop Access Patterns

As discussed previously, Vuvuzela's interpretation of differential privacy is quite restrictive by only allowing for the communication pattern of *one* client to differ between scenarios. If the difference is greater, the server cover traffic can not be assumed to be able to hide dead drop access patterns sufficiently.

To allow for greater variations, the obvious solution is to raise the amount of cover traffic the servers add. We examine how much additional cover traffic is needed per additional client whose communication patterns are allowed to differ:

Algorithm 4 f-VCP: Last Server Protocol

```
▷ Changes preventing replay attacks are highlighted in blue
▷ Changes preventing timing attacks are highlighted in red
function ProcessRound
    collect requests
    for all $req \in$ requests do DecryptRequest($req$)                ▷ See Algorithm 6
    initialize $rc$        ▷ request content: $rc[i] = c_i$ of $i$th request $req_i = (d_i, r, c_i)$
    $rc[-1] \leftarrow$ random
    initialize $rp = [0, \ldots, 0]$                                   ▷ request partner
    for all $req_i = (d_i, r, c_i) \in$ requests do                    ▷ Access dead drops
        if $\exists\, req_j = (d_i, r, c_j)$ then
            $rp[i] = j$
        else
            $rp[i] = -1$
    for all $req_i = (d_i, r, c_i) \in requests$ do
        $pid \leftarrow rp[i]$
        $req_i \leftarrow (d_i, r, rc[pid])$
    build up onion layers
    send requests to previous server
```

Each inner server adds n_1 single dead drop accesses and n_2 double dead drop accesses as noise. n_1 and n_2 are chosen using a *Laplace* distribution, which is defined by the location parameter μ (equivalent to the mean value) and the scale parameter s:

$$n_1 = \lceil \max(0, \text{Laplace}(\mu, s)) \rceil$$
$$n_2 = \lceil \max(0, \text{Laplace}(\mu/2, s/2)) \rceil$$

Lemma 3 in [2] states that the system is (ε, δ)-differentially private for up to t changes (between scenarios), if $\varepsilon = \frac{t}{s}$ and $\delta = \frac{1}{2} \exp \frac{t-\mu}{s}$.

For a given ε and δ, we get the following values for the Laplace distribution:

$$s = \frac{\varepsilon}{t}, \qquad \mu = 2 - \frac{t \ln \delta}{\varepsilon}$$

With a growing number of changes between scenarios t, the mean number of cover requests μ grows linearly, while the scale s falls linearly. Figure 8.1 and Table 8.1 shows the progression of μ for different values of t[3].

Table 8.1: $\mu(t)$ for different t

t	$\mu(t)$ for $\delta = 10^4$ and $\varepsilon = \ln 2$
1	15
10	134
100	1330
1000	13289

Multiple Rounds of Conversation. The parameters mentioned above are sufficient, if the adversary only observes a single round of conversations. If he gets to observe k rounds, the differential privacy parameters ε and δ worsen by a factor dependent on k (Theorem 2 in [2]):

[3]Source code to generate Figure 8.1 can be found at `https://gist.github.com/coijanovic/9491fafa36ee2dcdb7be8cc157b004c4`

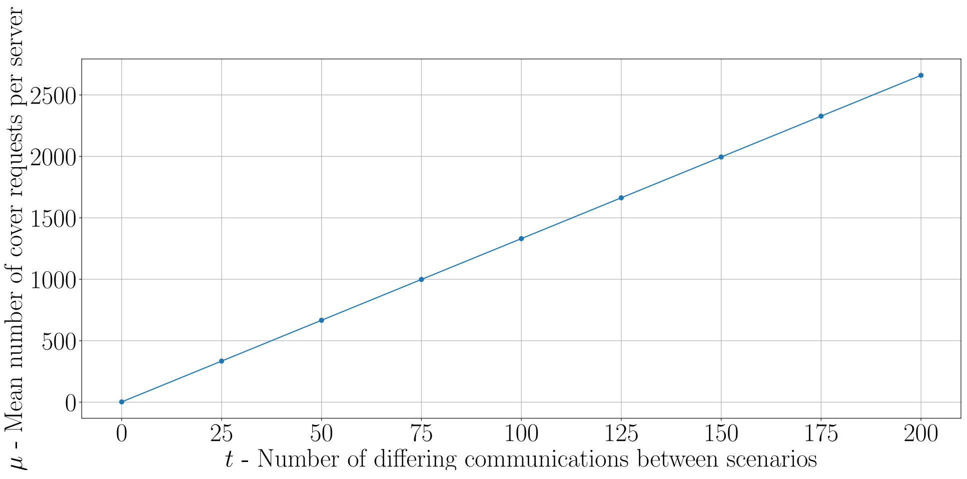


Figure 8.1: A plot of $\mu(t) = 2 - \frac{t \ln \delta}{\varepsilon}$ for $t \in \{0, \dots, 200\}$ with $\delta = 10^{-4}$ and $\varepsilon = \ln 2$. These value for δ and ε are recommended in [2, Sec. 6.4].

$$\varepsilon' = \sqrt{2k \ln(1/d)}\varepsilon + k\varepsilon(e^{\varepsilon} - 1)$$
$$\delta' = k\delta + d$$

d is a free parameter that allows a trade-off between ε' and δ'.

$\mathcal{A}_C$ vs $C\overline{O}$. To demonstrate that this factor is needed, we show that $C\overline{O}$ can be broken by observing dead drop access patterns:

The Attack. $\mathcal{A}_C$ is allowed to observe k rounds. In scenario 0, all n clients are in pairwise communication and in scenario 1, all clients are not in active communication. Thus, if scenario 0 is chosen, $n/2$ "real" double dead drop accesses occur (per round) and if scenario 1 is chosen, n single dead drop accesses occur. Let $\mathcal{A}_C$ corrupts the last server and all but one inner servers. We assume that the honest inner server adds *just enough* cover requests to hide the difference in dead drop access patterns in one round.

$\mathcal{A}_C$ knows that the honest inner server chooses the number of double cover requests with a mean value of $\mu/2$ and the number of single cover requests with a mean value of μ. Let da_1 be the total number of single dead drop accesses during the k rounds and let da_2 be the total number of double dead drop accesses during the k rounds. $\mathcal{A}_C$ computes $da_1' = da_1 - k \cdot \mu$ and $da_2' = da_2 - k \cdot \mu/2$.

If $da_1' > da_2'$, $\mathcal{A}_C$ concludes that scenario 1 must have been chosen and returns $b' = 1$. Otherwise, he returns $b' = 0$.

Takeaways. It was to be expected that allowing for greater differences between scenarios requires more server cover traffic. Fortunately, the amount of cover scales only linearly with growing numbers of differences t. Ideally, t (and therefore the amount of server cover) would depend on the total number of users (or better communications), one could set $t = \frac{\text{\# communications}}{2}$, allowing for 50% of all communication patterns to differ between scenarios. Sooner or later, the amount of cover traffic will become unbearable with a growing number of users; A compromise between scalability and dead drop access pattern protection has to be found.

8.4. Analysis

8.4.1 Vuvuzela Communication Protocol

Our goal is to show that the fixed Vuvuzela Communication Protocol (*f-VCP*) reaches *Communication Unobservability* ($C\overline{O}$) under the *weak* client cover assumption (Assumption 9) against any adversary who is not able to corrupt clients. This represents a significant improvement compared to the original Vuvuzela communication protocol, which only reaches $C\overline{O}$ under the *strong* client cover assumption. Under the weak client cover assumption, all notions with the exception of $M\overline{O} - |m|$ can be broken in VCP.

To do so, we first show that no adversary without the ability to corrupt clients can distinguish **IC**-Type requests from the other types. This requires Lemmas 5 to 7, where we show that $\mathcal{A}$ cannot distinguish by timing at any point in the server chain and cannot use replay to determine the type of request a given sender sends.

We start by showing that $\mathcal{A}$ cannot use timing to distinguish between types. Intuitively, this is clear since clients and the last server were especially modified to not show timing differences when processing different request types. Inner servers already show no timing differences when processing requests in Vuvuzela. Since the inner server modification do not cause differing behavior depending on request type, this also holds for f-Vuvuzela.

Lemma 5 (Timing)**.** *No adversary can distinguish* ***IC****-Type requests from other types by timing clients or servers.*

Proof. The adversary $\mathcal{A}$ can try to time inner servers, the last server or clients.

Inner Servers. In each round, the inner server collects all requests, removes the outer onion layer from each requests, adds server cover, shuffles all requests and sends them to the next server. For f-VCP, the server (additionally) always checks the included round number r for each request and deduplicates the list of requests. This behavior does not depend on the type of any request. Thus, $\mathcal{A}$ cannot distinguish request types by timing inner servers.

Clients. In the original Vuvuzela protocol, clients show differing behavior and timing differences when sending **M/AC**-Type requests compared to **IC**-Type requests. This is no longer the case with f-VCP, since the client executes the same operations in all cases: It first chooses a random public key, then sets the public key pk_x either to the random key or the partner's key, computes shared secret and dead drop id and constructs the request. There is a small difference with adding padding to the message (all padding in case of **IC**-Type), but it follows with Assumption 12 that this difference is not timeable by $\mathcal{A}$.

Last Server. In the original Vuvuzela protocol, the last server also shows timing differences depending on the type of request. This is no longer the case with f-VCP, since the last server executes the same operations in all cases: It always initializes the arrays rc and rp, checks each request for a partner and then updates the contents of each request before sending it back. ■

Next, we show the effectiveness of replay detection in f-Vuvuzela. While we were able to show that $\mathcal{A}$ cannot distinguish types by timing at any point in the server chain, we can only prove that $\mathcal{A}$ cannot determine the type of request a given sender sends by replaying. This is due to the fact that the detection of replays requires an honest server between the point where the request is replayed and the last server. If the adversary replays the request after the last honest server, there is no way of stopping it from reaching the last server. Fortunately, if the request is replayed after an honest server, the adversary is no longer able to link it to any client, making targeted attacks impossible. Intuitively, $\mathcal{A}$ cannot tell the type of request a given client sends by replay, because the honest inner server removes the replayed request.

Lemma 6 (Replay)**.** *No adversary can adversary can determine the type of request a given clients sends by replaying requests.*

Proof. In the original Vuvuzela protocol, replay attacks can be executed, because requests can be replayed where they can be linked to a specific client (e.g., on the link between client and first server) and both replay and original reach the last server. This is no longer the case in f-VCP, since the first honest inner server has added mechanisms that detect and discard replayed requests. It can be assumed that there is an honest inner server, since replay attacks in Vuvuzela require to corrupt the last server. If the last server is *not* corrupted, the adversary does not learn additional information other than that requests where replayed, which he already knows. Since the adversary is only allowed to corrupt all but one Vuvuzela servers, the remaining honest server has to be an inner one. The adversary has to replay the request prior to the first honest server to be able to link it to the sender.

Same Round Replay. If the replayed request is send during the same round, the first honest inner server receives two bit-identical requests (among the other requests from this round). In *f*-Vuvuzela, the inner servers have an added DEDUPLICATE function that is executed after decrypting the outer onion layers of the requests. DEDUPLICATE compares all requests to each other and checks if there are any bit-identical duplicates. If there are, one copy is removed from the list of requests, preventing that both the original request and the replay reach the last server.

The server only has to check for bit-identical duplicates due to the *diffusion* property of the encryption scheme: If the adversary changes the replayed request slightly to circumvent the DEDUPLICATE function, the contained information is changed with decryption and the replayed request will not reach the same dead drop anymore with overwhelming probability.

Later Round Replay. To prevent the adversary being able to replay the request in a later round, the client includes the current round number r inside of each onion layer. The first honest inner server reveals r after decrypting the outer onion layer. It compares the included r to its reference round number and discards the request if they don't match. If the adversary replays the request in a later round, the request's r will be out of date and the request is prevented from reaching the last server.

The adversary cannot update r in the replayed request, since the r that the server checks is always protected by an encryption layer prior of reaching the server. The diffusion property prevents the adversary from being able to manipulate r deliberately through this encryption layer.

Reverse Replay. It is conceivable that the adversary could also execute a reverse replay attack, i.e., replay requests at the last server and check at which client they arrive. This class of attacks is already prevented by the original Vuvuzela protocol: When receiving the requests from the clients, the first server saves a mapping from request index to user (e.g., "ith request is from Alice"). On the way back, each client only gets send the requests he is associated with. A replayed request necessarily has a different index than the original request and is therefore not send to the same client[4]. ■

Lemma 7 (Replay & Timing)**.** *No adversary can find out the type of request a given client sends a request to by replaying* and *timing requests.*

Proof. As discussed previously, $\mathcal{A}$ has to replay the request prior to the first honest server to be able to link it to the client. The first honest server will filter out the replayed request, which requires additional time. The time the server needs to filter the requests does only depend on the number of replayed requests, *not* the type of the request. Thus, $\mathcal{A}$ can gain no further information by replaying *and* timing. ■

Next, we show that no adversary without the ability to corrupt clients can determine the type of request a given client sends in *f*-VCP. To make the proof more compact and less repetitive, we rely on Theorem 1, where we showed that no such adversary can distinguish if the client sends **M**-Type or **AC**-Type requests in the original Vuvuzela communication protocol. In order to use Theorem 1, we have to show that it still holds in *f*-VCP.

Recall that we have proven Theorem 1 by showing that any adversary $\mathcal{A}$, that can distinguish **M**-Type requests from **AC**-Type requests, can be used by another adversary $\mathcal{B}$ to break the IND-CPA game. We were able to do so, since **M**-Type requests only differ from **AC**-Type requests in their contents in VCP. This is still the case for *f*-VCP, since the way **M/AC**-Type requests are generated and handled remains identical: The requests are still send through the mixnet to the last server, where they have their contents exchanged with a partner request in a pseudo-random dead drop. The introduced replay mechanism works independently of request type: If a request is duplicated or from a previous round, it is discarded. Thus, the same reduction to IND-CPA

[4]Corrupting the first server to force it to send the replayed request to Alice does not work, since the honest inner server makes it impossible for the adversary to identify the replayed request at the first server.

security can be made in f-VCP and Theorem 1 still holds.

We also make use of Theorem 4, which states that in VCP, $\mathcal{A}$ can *only* distinguish if a client sends **IC**-Type versus the other types by timing or replay. We recall why $\mathcal{A}$ cannot do so by other means and examine if those reasons are still valid in f-VCP:

Request Alteration. In VCP, $\mathcal{A}$ is not able to distinguish **M/AC**-Type from **IC**-Type by alterations other than replay, because the server cover traffic hides any resulting changes. This remains true in f-VCP, as requests are handled in the same way and the server cover also remains unchanged.

Server Corruption. In VCP, $\mathcal{A}$ is not able to distinguish **M/AC**-Type from **IC**-Type by corrupting servers, since inner servers do not gain information that can be used to distinguish types and at the last server, requests cannot be linked to clients anymore (if the last server is corrupted, a inner server has to be honest). The same is true in f-VCP, as the only additional information that the servers learn is the included round number in each request, which is identical in all requests from honest clients.

Passive Observation. In VCP, $\mathcal{A}$ is not able to distinguish **M/AC**-Type from **IC**-Type by passively observing, since **AC**-Type and **IC**-Type requests only differ in the receiver: **AC**-Type requests are send to a real receiver and **IC**-Type requests to a random one. The same is true for f-VCP, as the construction of requests remains unchanged.

Thus, Theorem 4 also holds in f-VCP. With that, proving request indistinguishably becomes just a matter of combining previous results:

Theorem 15 (Request Indistinguishably)**.** *No adversary $\mathcal{A}$ without the ability to corrupt clients can determine the type request a given client sends in f-VCP.*

Proof. Combining Theorem 1 and Theorem 4 with Lemmas 5 and 6 yields our desired result. ∎

We use Theorem 15 to show that no adversary can determine the *request type distribution* (i.e., how many requests are **M**-Type versus **AC**-Type versus **IC**-Type).

Lemma 8. *In a given round, adversaries without the ability to corrupt clients cannot determine the request type distribution, given sufficient server cover traffic.*

Proof. There are two possible cases:

1. *Prior to the first honest server*, $\mathcal{A}$ cannot determine the request type distribution due to Theorem 15. If $\mathcal{A}$ cannot determine the type of any single request at this point, he also cannot determine how many requests are of the same type.
2. *After the last honest server* (which can be the same as the first honest server), $\mathcal{A}$ can track each request to the last server and observe if it causes a single or double dead drop access and therefore determine the type of all requests. This is *not* a contradiction to the claim, since at this point, the honest server(s) have added random amounts of single and double dead drop accesses to the requests. The server cover requests are indistinguishable from real requests for the adversary, since the servers generate them in the same way as clients do the real ones. The amount of server cover traffic is chosen in a way that makes it impossible for the adversary to tell the 'real' distribution of types. ∎

Finally, $\mathcal{A}_{GCTM}$ versus $C\overline{O}$ can be proven similarly to the way $\mathcal{A}_{GCTM}$ versus $C\overline{O}$ under the strong client cover assumption was proven in VCP (Theorem 8). Under the strong client cover assumption, clients only send **M**-Type and **AC**-Type requests. This is no longer the case under the weak client cover assumption, where also **IC**-Type requests are send. The proof has to be modified with the help of Lemma 8.

Theorem 16 (Communication Unobservability)**.** *No adversary $\mathcal{A}$ without the ability to corrupt clients can win the $C\overline{O}$-game with a non-negligible advantage under the weak client cover assump-*

tion.

Proof. We define a series of hybrid games:

- H_0: The original $C\overline{O}$-game
- H_1: H_0, but all messages have to be 0
- H_2: H_1, but clients send **IC**-Type requests instead of **AC/M**-Type requests
- H_3: Identical Scenarios

In the following, we show that any adversary that can win the game H_i can also win the game H_{i+1} (for $i \in \{0, \dots, 2\}$). Since H_0 is the $C\overline{O}$-game, the adversary is not allowed to corrupt any senders or receivers in the hybrid games.

1. $H_0 \approx H_1$: Corollary 3 shows that any adversary $\mathcal{A}$ who breaks H_0 does so without relying on different messages between the scenarios[5]. Thus, $\mathcal{A}$ can also break H_1.
2. $H_1 \approx H_2$: Lemma 8 shows that no adversary without access to challenge clients can determine how many requests are **IC**-Type versus **AC/M**-Type. Note that the only difference between H_1 and H_2 is the distribution of request types: There are only **IC**-Type messages in H_2 and a mix of **IC**-, **M**- and **AC**-Type in H_1. For this change to be indistinguishable, the dead drop access patterns have to also be indistinguishable from each other. This requires a sufficient amount of server cover traffic (t = #communications). It follows that if $\mathcal{A}$ wins the H_1-game, he can also H_2, since the difference between the games is indistinguishable to him.
3. $H_2 \approx H_3$: In H_2, clients are restricted to only sending **IC**-Type requests. We note that the number of clients connected to the service is equal in both scenarios and that clients send the same number of requests in both scenarios by Assumption 8. It follows that the number of requests send is equal in both scenarios and therefore the scenarios of H_2 are already identical.

Conclusion. Let $\mathcal{A}$ have an non-negligible advantage of winning the H_0-game. We have shown that $\mathcal{A}$ also has a non-negligible advantage of winning the H_3-game. Since H_3 contains identical scenarios, no adversary can win with probability better than guessing. ■

8.4.2 Vuvuzela Dialing Protocol

The structure of the dialing protocol inherently allows it to link receivers to dead drops. Our goal in this section is to show that in f-Vuvuzela it is impossible for the adversary to link (honest) senders to dead drops. If senders cannot be linked to dead drops, they also cannot be linked to receivers. The dialing protocols central purpose is to hide which sender is inviting which receiver. If sender-dead drop unlinkability is proven, it can also be proven that no adversary without the ability to corrupt clients can break $C\overline{O}$. This is due to the fact that the only thing the adversary can choose in his challenges is which client invites whom, request content is determined without the adversary's influence.

In Chapter 5, we have determined that no adversary without the ability to corrupt clients or time/replay requests can link a given sender to a dead drop. We start our analysis by showing that linking senders to dead drops is also not possible in f-VDP by timing and/or replaying.

Intuitively, timing cannot be used to link senders to dead drops, since f-VDP was explicitly modified to remove all timing differences compared to VDP.

Lemma 9 (Timing). *No adversary can find out which dead drop a given clients sends a request to by timing clients or servers.*

Proof. The adversary can try to time inner servers, the last server or clients.

Inner Servers. The last server does not change its behavior depending on the dead drop a given request is send to, it always

[5]We can assume that Corollary 3 also holds in f-VCP, since it is implied by Theorem 1. Theorem 1 was shown to hold in f-VCP.

1. Decrypts all requests
2. Checks for actuality of the requests
3. Deduplicates the requests
4. Generates cover traffic
5. Shuffles requests
6. Sends the requests to the next server

Clients. The client always executes the same steps when sending an invitation:

1. Chooses a random public key
2. Sets the used public key
3. Computes a dead drop id
4. Constructs the request
5. Onion encrypts and sends the request

Last Server. The last server also does not change its behavior depending on the dead drop a given request is send to, it always

1. Decrypts all requests
2. Checks of actuality & deduplicates
3. Generates cover
4. Shuffles requests
5. Adds each request to the corresponding dead drop

The requests themselves also do not change depending on which dead drop they are send to except for the included dead drop id d, which always has the same length.

If client, inner servers and last server always execute the same steps on requests that have the same length independently from which dead drop the given request is send to, they do not show differing timing. Thus, the adversary cannot determine which dead drop the request is send to by timing. ■

Intuitively, $\mathcal{A}$ cannot link senders to dead drops by replay, since f-VDP has the same added replay protection mechanisms as f-VCP.

Lemma 10 (Replay)**.** *No adversary can find out which dead drop a given clients sends a request to by replaying requests on any network link.*

Proof. In the original Vuvuzela Dialing Protocol, an adversary can replay an invitation at any point where he can link it to the sender (e.g., on the link between sender and first server) and observe at the last server which dead drop receives two identical requests to determine which dead drop the client send his invitation to.

Like f-VCP, f-VDP has added mechanisms that enable the first honest inner server to detect and discard replayed requests. Since the inner server modifications are the same in f-VDP as in f-VCP, the proof of Lemma 6 can also be used here. Reverse replay attacks do not need to be considered, because in the dialing protocol, the invitations are not returned to the clients via the server chain but rather accessed directly at the dead drops. The mapping from client to used dead drop is already publicly known. ■

Corollary 10 (Replay & Timing)**.** *Lemma 7 (Replay & Timing) from f-VCP also holds in f-VDP, since the inner server behavior is identical in both protocols: No adversary can determine the type of request a given client sends or the dead drop the request is send to by combining replay and timing attacks.*

Next, we can prove sender-dead drop unlinkability. As with the analysis of f-VCP, we rely on results from the analysis of the original Vuvuzela Dialing Protocol. There, Theorem 13 states that adversaries without the ability to corrupt clients cannot link senders to dead drops by corrupting servers, altering requests (other that replaying) or passive observation. Theorem 13 holds in f-VDP, if Lemmas 2 and 3 hold in f-VDP.

Request Alteration (Lemma 2)**.** In VDP, $\mathcal{A}$ cannot link senders to dead drops by altering requests (other than replaying), because the server cover traffic hides any observable changes in dead drop access patterns. This remains true in f-VDP, as requests are handled in the same way and the server cover generation remains unchanged.

Server Corruption (Lemma 3)**.** In VDP, $\mathcal{A}$ cannot link senders to dead drops by corrupting (all but one) server, since inner servers do not gain information that can be used to distinguish types and at the last server, requests cannot be linked to clients anymore (if the last server is corrupted, a inner server has to be honest). The same is true in f-VCP, as the only additional information that the servers learn is the included round number in each request, which is identical in all requests from honest clients.

Thus, Theorem 13 still holds in f-VDP.

With that, proving sender-dead drop unlinkability becomes just a matter of combining previous results:

Theorem 17 (Sender-Dead Drop Unlinkability)**.** *No adversary without the ability to corrupt clients can link a sender to the dead drop he sends an invitation to.*

Proof. Combining Theorem 13 with Lemmas 9 and 10 yields our desired result. ■

To prove communication unobservability, we also need to show that the adversary cannot distinguish **C**-Type from **I**-Type requests. Intuitively, this makes sense, because they only differ in the dead drop they are send to.

Theorem 18 (Type Indistinguishably)**.** *No adversary without the ability to corrupt clients can determine the type of a given request if innermost encryption scheme is key-private.*

Proof. In f-VDP, request types are only distinguished by the public key that is used to encrypt them: While a "real" public key (i.e., one that is used by another user) is used for **I**-Type requests, a random public key is chosen for **C**-Type requests. If the innermost encryption scheme is *key-private* (as introduced in [16]), public keys cannot be determined from ciphertexts. Thus, the adversary is not able to distinguish request types. ■

With that, we can prove that $C\overline{O}$ holds in f-VDP:

Theorem 19 (Communication Unobservability)**.** *No adversary without the ability to corrupt clients can win the $C\overline{O}$ game with a non-negligible advantage assuming sufficient server cover traffic to cover arbitrary changes in dead drop access patterns.*

Proof. First, we note it follows with Assumptions 15 and 16 that in both scenarios the same clients are connected and that the same number of requests are send per client. What can differ is the distribution of **I**-Type versus **C**-Type and the dead drops the **I**-Type requests are send to[6].

Due to Theorem 18, $\mathcal{A}$ cannot determine the type of any given request. Assuming sufficient server cover traffic to hide differing dead drop access numbers, it can be proven analogously to Corollary 4 that $\mathcal{A}$ can also not distinguish different distributions of **I**-Type and **C**-Type. Due to Theorem 17, $\mathcal{A}$ cannot determine the dead drop any given client sends his request to.

What remains are differences between the scenarios in the dead drop access patterns (e.g., in scenario 0, all clients could invite Bob and in scenario 1, all clients could invite differing receivers). Since we assume that any differences in dead drop access patterns are hidden by the cover traffic of the one remaining honest server, $\mathcal{A}$ cannot use this information to tell the scenarios apart. ■

[6]The dead drops that **C**-Type are send to can also differ, but since they are randomly chosen by honest clients, $\mathcal{A}$ has no way of influencing them.

8.5. Discussion

First, we want to give a rough estimate of how the changes of f-Vuvuzela effect performance. We start by examining the added computational overhead for clients, inner servers and the last server. After that, we consider the increased request size, which causes additional network load and storage requirements in clients and servers.

Client Communication Protocol. When sending an **M/AC**-Type request, the client additionally has to choose a random public key. When sending **IC**-Type requests, no additional steps have to be taken. In both cases, the current round number r has to be computed and added to the request in each onion layer. We conclude that the added computational load for clients is minimal.

Client Dialing Protocol. As with the communication protocol, the client has to additionally choose a random public key when sending a **I**-Type request. When sending a **C**-Type request, he has to compute a nonce, MAC and execute one encryption as opposed to choosing a random ciphertext. In both cases, the current round number r has to be computed and added to the request in each onion layer. The added computational load when sending an **I**-Type request is minimal, but the added load when sending a **C**-Type request is more significant. Nevertheless, we conclude that this is not unreasonable for the client, since the overhead for sending **C**-Type requests is now equal to **I**-Type requests, which the client can handle.

Inner Servers. The inner server is changed in the same way in both the communication and dialing protocol: It has to check each request for a valid round number. The added overhead for this step is minimal, since the contained r only has to be compared to a saved reference integer. It also has to deduplicate the list of requests, which adds a more significant amount of overhead. As described in Section 15.5, *for each request*, a hash value has to be computed and an array has to be accessed once. We conclude that this is an reasonable amount of overhead since it scales linearly with the number of requests.

Last Server. In the communication protocol, the last server has to additionally fill the rc array, which requires one array access per request. Instead of one loop where requests are checked for a partner *and* have their contents exchanged, f-VCP requires the list of request to be iterated over *twice*. While the computational overhead at the last server is greater than at the other entities, we still conclude that it is reasonable, since it scales linearly in the number of requests. In the dialing protocol, the last server remains unchanged.

Network Load. The amount of added network load (and required storage at clients and servers) depends on the number of servers in the chain and the total number of expected rounds between network resets[7]. As an example, we assume that the server chain consists of three servers (which is the number of servers used in the existing experimental Vuvuzela implementation, see [2, Sec. 8.1]). If 32 bit round numbers are used, $2^{32} = 4.294.967.296$ unique round numbers are possible. If each round lasts 2 minutes, these round numbers would last for 8.589.934.592 minutes or more than 16.000 years, which we think is a reasonable amount of time between resets.

Since the round number has to be added in each onion layer, the total overhead per request is $3 \cdot 32 = 96$ bit. This is an insignificant amount of overhead, since the requests also contain 3 128 bit `Curve25519` public keys, a 128 bit dead drop id and the message itself. While [2] does not suggest a concrete message length, it is reasonable to assume that it would be at least as long as a standard SMS message, which has a limit of 160 characters (or 1120 bits) [17].

8.5.1 Remaining Limitations

There a some aspects where Vuvuzela is lacking which are not fixed by f-Vuvuzela.

Chain Length Scalability. In Vuvuzela one of the servers in the chain has to be trusted to enable any privacy protection. Naturally, an advanced user might want to add his own Vuvuzela server to the chain to improve the chance of it containing an honest one. Since each server has

[7]Which could be necessary when a new version of the server/client software is available.

to process the incoming requests, a significant amount of latency (depending on the number of requests) is added to the network with each additional server.

Excessive Server Cover. Lengthening the server chain comes with a second problem: Since each server could be the one remaining honest one, it has to add enough server cover traffic to cover changes in dead drop access patterns by itself. If multiple servers are honest, more cover that necessary is added each round, requiring more memory and computational power at later servers.

Client Corruption. f-Vuvuzela still assumes that clients can be trusted. If a client is corrupted, the adversary can break *any* notion in both f-VCP and f-VDP.

9. Improving Vuvuzela

Our goal in this section is to improve upon the structure of Vuvuzela, to minimize the amount of server cover traffic needed. To do so, we propose *Two-Phase VCP* (*2P*-VCP).

We start by outlining a steppingstone protocol, which introduces the structure of *2P*-VCP but is only secure against passive adversaries. In section Section 9.1, we discuss what changes can be made in order to protect against active adversaries.

The network structure remains identical to Vuvuzela's: There is a chain of mixnet servers. Clients are connected to the first server and the last server has access to a number of dead drops.

Two Phases. Our main contribution consists of splitting each round into two distinct phases. In the first phase (*deposit*), the clients add requests to dead drops. In the second phase (*withdrawal*), the dead drops are accessed by the receivers (see Figure 9.1).

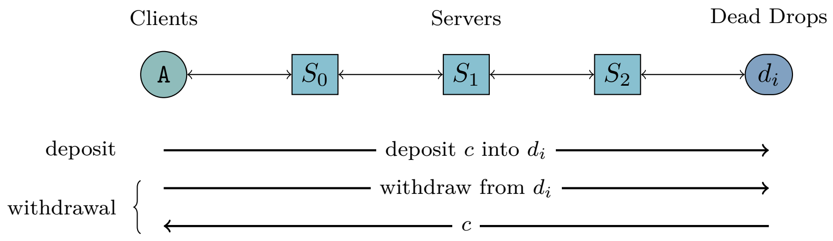


Figure 9.1: *2P*-VCP request flow for **IC**-Type requests. Client A deposits a (random) ciphertext c into dead drop d_i and withdraws her own ciphertext from the same dead drop. If the client is communicating with another user, she withdraws from another dead drop.

Computing Dead Drop IDs. Contrary to Vuvuzela, where sender and receiver exchange messages using the *same* dead drop ($H(k_{ab} \mid r)$), *2P*-VCP uses differing ones. Each conversation has an *inviter* (the one who send the initial invitation) and an *invitee* (the one who was invited). If the *inviter* sends a request to the *invitee*, she computes $d^r_{ab} = H(k_{ab} \mid r)$ with r being the current round number. If the *invitee* sends a request to the *inviter*, he computes $d^r_{ba} = H(k_{ab} + 1 \mid r)$.

Having differing dead drops for sender and receiver is not a novel idea, e.g., *Pung* [18] uses the same concept.

Message Deposit. Each client has a fixed number ν of available communication slots. Each round, the client sends ν requests[1]. These can be a mix on **M/AC**-Type and **IC**-Type requests depending on the number of active conversations the client is in. Let Alice be in active conversation with Bob and let Alice be the inviter.

- If Alice has a message m to send to Bob in the current round, she computes the **M**-Type request $req = (d^r_{ab}, enc_{k_{ab}}(m))$, where k_{ab} is the shared key between Alice and Bob and $d^r_{ab} = H(k_{ab} \mid r)$
- If Alice has no message to send, she computes the **AC**-Type request $rep = (d^r_{ab}, enc_{k_{ab}}(\mathbf{0}))$

If Alice has no active conversation partner for the current slot, she chooses a random public key pk_z

[1]The client is restricted to sending one message per receiver per round.

with which she can compute the shared key k_{az} and the **IC**-Type request $req = (d^r_{az}, enc_{k_{az}}(\mathbf{0}))$. In this case, Alice computes d^r_{az} as the inviter.

In all cases, the request is onion wrapped and send through the server chain in the same way as VCP. The last server adds the requests content to the specified dead drop.

Contrary to VCP, the request is *not* send back through the server chain to the client at this point.

Message Withdrawal. After all messages have been deposited, the clients can start the withdrawal phase. To do so, Alice computes the dead drop ids where she suspects messages for her for each communication slot. For her communication with Bob, this would be $d^r_{ba} = H(k_{ab}+1 \mid r)$. To withdraw Bob's message, she sends an **W**-Type (Withdrawal) request to this dead drop through the server chain. A **W**-Type request only contains a dead drop id and no ciphertext. For slots where Alice has no communication partner, she withdraws her own cover request from the first phase.

After receiving the **W**-Type requests, the last server appends the contents of the corresponding dead drop to each request. If the last server comes upon a request that contains a dead drop id, which has not been accessed in the first round (which could happen due to transmission errors), it generates a random cover ciphertext as response. The modified request is then send back to the client through the server chain as in VCP.

Timing. One has to make sure that there are no timeable differences in sender behavior depending on the type of request send. Analogous techniques to *f*-Vuvuzela (see Section 8.2) can be used.

Security Intuition. Before tackling protection against active adversaries, we want to give some intuition on why *2P*-VCP is secure against passive adversaries.

The adversary cannot learn anything other than the number of requests send[2] by observing the dead drop access patterns, since they don't change depending on communication patterns: We assume that there are n clients that all have ν communication slots[3]. Thus, in the first phase, $n \cdot \nu$ requests are send from clients to (pseudo-)random dead drops. Due to the use of a strong cryptographic hash function, each dead drop is only accessed once with overwhelming probability.

In the second phase, each of the dead drops that was deposited to in phase one is again accessed once (for **M/AC**-Type requests by the communication partner and for **IC**-Type requests by the original sender).

If the adversary is able to observe dead drop access patterns, he has corrupted the last server. Since he is only allowed to corrupt all but one servers, there must be a remaining honest inner server. This honest inner server ensures, through onion-decryption and shuffling, that $\mathcal{A}$ cannot determine whether a dead drop was withdrawn from by the same client which deposited into it or a different one.

As with all other Vuvuzela variants, the message itself is always protected by at least one (IND-CPA secure) encryption layer for which the key is only known to sender and receiver of the request, which $\mathcal{A}$ is not allowed to corrupt.

9.1. Active Adversaries

2P-VCP as it is described above is *not* sufficiently protecting user privacy against active adversaries. An active adversary can use *any* of his additional abilities (i.e., modifying, delaying, dropping and replaying) to determine the dead drop a given client sends his request to. If done in both phases, the adversary can determine if two suspected clients are indeed communicating with each other (if Alice is withdrawing from the dead drop that Bob deposited to). This also implies that $\mathcal{A}$ can

[2]which identical in both scenarios and can already be determined by counting the number of requests at any point in the server chain.

[3]This is a simplification. In the real protocol, ν is a variable that can be set by each client individually.

determine if Alice is sending an **IC**-Type request, since Alice is effectively communicating with herself in this case.

We present concrete example attacks for each kind of alteration and introduce possible measures of protecting against them.

Replay. $\mathcal{A}$ can replay the request in question and corrupt the last server to observe which dead drop is accessed twice with the same request. Luckily, we already constructed a mechanism to detect and discard replayed requests for *f*-VCP (see Section 8.1). This mechanism can also be used in *2P*-VCP.

Modification. If $\mathcal{A}$ modifies a request on the way to the last server, it will reach a random dead drop due to the diffusion property of the used encryption scheme. One possible approach is for $\mathcal{A}$ to modify all requests except for Alice's in the first phase. In the second phase, $\mathcal{A}$ modifies all requests except for Bob's. $\mathcal{A}$ corrupts the last server and observes the dead drop accesses. If there is a successful access, it is highly likely that Alice and Bob are communicating with each other.

$\mathcal{A}$ could also just modify Alice's requests in both phases. If Alice send an **IC**-Type request, $\mathcal{A}$ observes one withdrawal from an empty dead drop. If Alice send an **M/AC**-Type request, $\mathcal{A}$ observes two withdrawals from empty dead drops (one by Bob at the dead drop that Alice meant to deposit in and one by Alice at a random dead drop).

To detect modified requests, we can make use of *digital signatures*, or more precisely *linkable ring signatures* (Liu et. al., [19]). We present an informal definition based on [20, Sec. 4.1]:

Definition 6 (Linkable Ring Signature)**.** A *Linkable Ring Signature* is a tuple (`Sign, Verify, Link`) of three poly-time algorithms:

1. $\texttt{Sign}(m, \{pk_1, \dots, pk_n\}, sk_i) \to \sigma$: generates a signature σ on input of a message m, n public keys $\{pk_1, \dots, pk_n\}$ and one secret key sk_i, which corresponds to public key pk_i
2. $\texttt{Verify}(m, \sigma, \{pk_1, \dots, pk_n\}) \to 0/1$: verifies on input of a message m, a signature σ and public keys $\{pk_1, \dots, pk_n\}$ if σ is a valid signature for m generated by one of the holders of the public keys
3. $\texttt{Link}(\sigma_0, \sigma_1) \to 0/1$: Outputs 1 if σ_0 and σ_1 are linked, i.e., were made using the same secret key.

The client includes a signature of the current ciphertext inside of each encryption layer[4]. In the following, we justify the use of relatively obscure and inefficient linkable ring signature over more common digital signature schemes.

Naively, we could employ a standard signature scheme and let the client sign the request with his long-lived secret key. This is not sufficient for our use case, since it would allow any server to determine the sender of a given request. A ring signature has the advantage of allowing the server to verify that the request was signed by *a* client (and not by the adversary himself) without revealing which client in particular is the sender.

We require the *linkability* property to prevent any corrupted client from sending too many requests: $\mathcal{A}$ could drop an arbitrary number of requests and then fill the freed spots with validly signed requests to chosen dead drops generated by a corrupted client. To prevent this from happening, we restrict the clients to only being able to send one request per round. If the signatures are linkable, the honest inner server can verify that each request was signed by a unique client. If this is not the case, it stops the current round.

Having the signatures be *linkable* presents a challenge in itself, if sender privacy is required. For example, the adversary could corrupt the first and last server. If the same key is used by the clients to generate signatures for both servers, $\mathcal{A}$ can link requests at the first server (where they can be linked to clients) to requests at the last server, effectively circumventing the mixnet altogether. If

[4]To do so, **W**-Type requests may need to contain a small dummy ciphertext, which would be exchanged by the last server for the dead drop content.

the same key is used to sign in both phases, $\mathcal{A}$ can tell whether a given dead drop was accessed by the same or different clients.

To solve this issue, the clients generates multiple signing keys: For each server, the client generates one key pair used in the deposit phase and another key pair for the withdrawal phase. The ith server than needs access to two sets of keys: $\mathbf{pk}^{\sigma}_{i,1}$ for the first phase and $\mathbf{pk}^{\sigma}_{i,1}$ for the second phase. Of course, these public keys need to be submitted to the servers in a way that unlinks them from the clients.

The adversary can still determine if two requests in different rounds were send by the same sender. He could compromise the system by blocking Alice for one round and saving all signatures from this round. In the next round, the signature that can't be linked to the previous round must be Alice's. With this knowledge, he can identify Alice's requests in all future rounds.

To combat this vulnerability, the clients could update their signing keys *every* round, which would add considerable overhead. Alternatively, *epochs* spanning multiple rounds can be introduced. Clients would use the signing keys for one epoch. If compromised, they can be linked to their requests for the remainder of the epoch, but no longer.

Remark (Modification versus Replacement)**.** There is no need for inefficient linkable ring signatures, if the requests only need to be protected from modification: In that case, a simple digital signature (or MAC) using a random key can be used. If a request is modified, the verification will no longer be successful. This does however not protect against the replacement of requests, which enables the same kinds of attacks. To stop clients from being identified by their signatures, random short-lived signing keys have to be used. In that case, $\mathcal{A}$ cannot be prevented from generating requests with valid signatures by himself. It might be possible to add additional mechanisms to detect replacements, but this would likely be no more efficient than using linkable ring signatures, since the same requirements have to be met.

Drop. Drop-based attacks can be executed analogously to modification attacks. The honest inner server can recognize drops by the number of requests it receives. Naively, it could stop the current round if it receives less requests than in the previous round.

This does not account for clients leaving/joining the protocol or loosing network connection. A more sophisticated solution requires more overhead:

Clients could be required to perform an authenticated handshake with each server at the beginning of the epoch. Only after successful handshakes with all servers is the client allowed to participate during this epoch. This approach assumes that the clients do not loose network connection or disconnect for any other reason during the epoch.

If this assumption is too strong (e.g., in a mobile network), the honest inner server could also ignore minor changes in request numbers and add a *small* amount of cover traffic to hide any changes in dead drop access patterns. This would require each server to have a repertoire of signing keys so that it is able to generate valid signatures from unique signers.

Delay. If a request is delayed and released within the same phase, it has no effect on the protocol execution. If it is delayed until a later round, the effect is the same as dropping it.

9.2. Discussion

At the beginning of this chapter, we set out to improve Vuvuzela's efficiency (especially with greater variation in communication patterns) by removing server cover traffic. While we have achieved the removal of server cover traffic (at least with a less resilient approach to drop protection), we were *not* able to improve efficiency when accounting for active adversaries.

The steppingstone variant of *2P*-VCP, which only protects against passive adversaries, is actually quite efficient: If Alice and Bob want to exchange messages in VCP, two requests are send from clients through the mixnet to the last server and two requests back. *2P*-VCP only requires two additional deposit requests from client to the last server (but not back).

Where *2P*-VCP falls short is the protection against active adversaries. While drop and replay protection add a manageable additional load to clients and servers (we have discussed the performance implications of the proposed replay protection in Section 8.5), the situation differs with the modification protection:

To the best of our knowledge, there is no linkable ring signature scheme that is able to handle millions of concurrent users (as is the case with Vuvuzela) with reasonable overhead. Having said that, there have been advances in making linkable ring signatures more efficient [20]–[23]. Further, there is some precedence of linkable ring signatures being used in anonymous communication networks, namely [24]. [25, Sec. 4] also suggests using linkable ring signatures in the same way as we propose to limit the number of requests the adversary can inject. There also are linkable ring signature schemes where the number of required computations is independent from the number of clients in the group (see e.g., [24, Sec. 5.3]), which comes in handy with growing numbers of clients.

Due to the way signatures in *2P*-VCP are used, we can make another efficiency improvement: Under normal circumstances, the ring signature σ includes the public keys of all senders [26, Sec. 2.1]. This makes the signatures prohibitively long for growing group sizes. Fortunately, in our case, the group of possible senders is known to the honest server, i.e., all clients connected to the system. Thus, the public keys do not need to be included with the signatures.

If one is willing to shrink the sender anonymity set, the clients could be distributed into smaller *signing groups*. This would improve efficiency, since the number of public keys needed to compute the signature could be drastically smaller. But it would also decrease sender privacy, since each server (and therefore the adversary) can reduce the set of possible senders of any given request to the members of the corresponding signing group.

Still, as it stands now, it is likely that *2P*-VCP increases the computational overhead for both clients and servers significantly compared to VCP. Due to these inefficiencies, we refrain from doing a full formal analysis of *2P*-VCP at this point.

10. Inspecting Vuvuzela Derivatives

There are multiple anonymous communication networks inspired by Vuvuzela ([3] and [4] directly, but also e.g., [27] and [28]), which promise further performance improvements. In this section, we examine *Stadium* [3] and *Karaoke* [4], since both cite Vuvuzela as a direct predecessor. Our goal is not to do a full analysis of each protocol but rather determine if they are still susceptible to the same attacks as Vuvuzela and if our proposed protection mechanisms from Chapter 8 can be adapted without major changes. We present the papers in chronological order: Vuvuzela was published in 2015, Stadium in 2017 and Karaoke in 2018.

10.1. Stadium

Protocol Overview. Stadium's basic communication structure is the same as Vuvuzela's: Clients send requests through a mixnet to dead drops. If two clients want to communicate with each other, they send their requests to the same dead drop. The content of the requests gets exchanged and each client is returned their own request. Stadium's main contribution lies in replacing Vuvuzela's single server chain by multiple parallel chains. The authors claim improved scalability since no server has to handle all requests.

Stadium's mixnet consists of a multiple parallel *input chains* and multiple parallel *output chains* with an all-to-all connection in between (see Figure 10.1)[1]. Stadium's adversary model assumes, as Vuvuzela, that at least one server per *mixchain* (i.e., one server in every input chain *and* one server in every output chain) is honest. If this where not the case, a client might pick a path through the network where *no* server is honest, enabling the adversary to track his request. Thus, Stadium requires *two* honest servers in the path between client and dead drop compared the Vuvuzela's single honest server. Because of this two stage approach, the authors assume a greater combined chain length than Vuvuzela ([3, Sec. 4] mentions $\sim 10-25$ servers versus 3 for Vuvuzela). Thus, the way the server cover traffic is added needs to be updated to stay efficient:

Where each honest Vuvuzela server adds enough cover traffic while processing the requests, Stadium has additional servers that add *just* the right amount of cover *prior* to mixing. These noise servers need to be trusted. The honest servers need to verify that the submitted messages where not altered by the adversary and that no server noise was removed by corrupted servers prior to them. For this, [3] introduces *hybrid verifiable shuffling.* One component that enables hybrid verifiable shuffling is a non-interactive zero knowledge proof (NIZK) that is appended to each request.

Attack Potential. With the NIZK, the sender can proof knowledge of the *authentication key.* Stadium actually also implements replay protection using this mechanism: Requests with duplicated proofs are removed and the authentication key is also bound to the current round number; The verification fails for requests generated in previous rounds. Compared to our proposed replay protection (see Section 8.1), this approach is quite similar in its basic idea: Remove same round replays by deduplication and later round replays by being able to deduce the round the request was generated in from the request. However, Stadium's approach introduces significantly more overhead: Not only does each server need to verify NIKZ, but those NIKZ also need to be re-randomized to prevent adversaries from linking messages through the mixnet. Having said that, this overhead might be justified, since the NIKZ are not only used for replay protection, but also to make sure that enough server cover traffic reaches the dead drops.

[1]Since there are multiple last servers, who each have their own set of dead drops, Alice and Bob have to make sure that their requests end up at the same output chain.

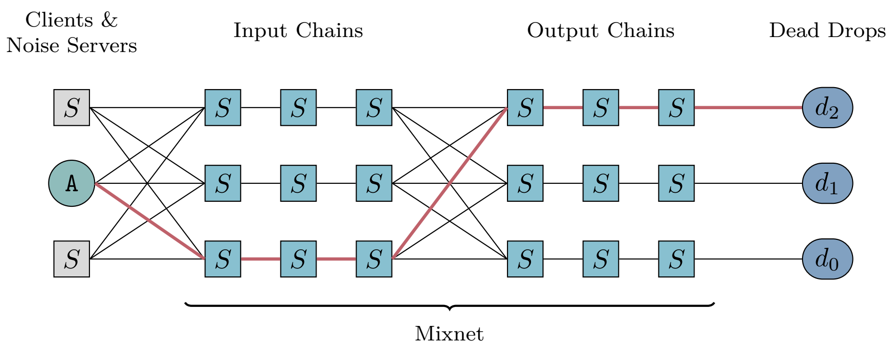


Figure 10.1: Simplified Stadium network structure.

[3, Sec. 5] states that users send a dummy request if they are not in active communication, just as Vuvuzela. The paper does not go into greater detail on how those dummy messages are generated. Looking at the general protocol architecture, we find it likely that random public keys have to be chosen. If this is indeed the case, the same attacks based on sender timing as in Vuvuzela are possible.

Applicability of Fixes. As mentioned before, Stadium has already built in replay protection. It might be possible to improve efficiency by exchanging it for our approach, but additional mechanisms to cover traffic generation would need to be introduced.

Since our strategy to avoid sender timing attacks is very nonspecific ("remove timing differences by executing the same computations in all cases"), it should be applicable to Stadium without problems.

Expected Analysis Results. Based on our analysis of Vuvuzela, we can make some predictions about which privacy notions can be achieved with Stadium against different adversaries. For that, we assume that their replay protection mechanism works as intended and that there are no further exploitable weaknesses compared to Vuvuzela.

- Under the strong client cover assumption, we can expect $C\overline{O}$ to hold against $\mathcal{A}_{GCTM}$, just as Vuvuzela. This is due to the fact that under the strong client cover assumptions *no* **IC**-Type requests are send (which would be detectable by sender timing).
- Under the medium and weak client cover assumption, we can expect that the notions that can only broken by replay in Vuvuzela hold in Stadium, but that all timing based attacks from Vuvuzela can also be executed successfully in Stadium. Thus, $\mathcal{A}_{CT}$ and $\mathcal{A}_{GT}$ can break $S\overline{O} - P$ under the medium client cover assumption and *any* notion except $M\overline{O} - |m|$ and $(SR)\overline{L}$ under the weak client cover assumption.

10.2. Karaoke

Protocol Overview. Karaoke uses a similar structure to Stadium with a mixnet of multiple parallel servers. Unlike Stadium, where there is only one all-to-all connection between the different horizontal "layers" of the mixnet, Karaoke has an all-to-all connection after each "column". Requests take a random path from column to column, two communicating clients need to ensure that their requests are routed to the same last server (see Figure 10.2). They do so by deriving not only the dead drop id, but also the id of the last server from their shared secret.

Its main distinction to the other protocols is that changes client behavior to eliminate single dead drop accesses[2]: In their case, this is done by making each client send two requests per communication slot. If Alice and Bob are communicating with each other, they each send one request to dead drop d_i and the other request to d_j. The contents of the requests are exchanged and send back to the clients. If Alice has no active conversation partner, she sends both of her requests to the same dead drop. With this modification, all dead drops are accessed twice and

[2]Which is the same idea that we had for *2P*-VCP.

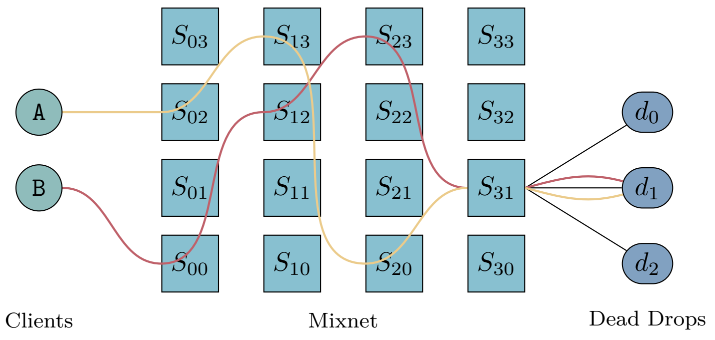


Figure 10.2: Simplified Karaoke network structure. Clients send their requests to a server of column 0 ($S_{00}, \dots, S_{03}$), requests are passed consecutively through the columns. The servers of the last column each have access to a set of dead drops.

dead drop access patterns are made independent from communication patterns. See Figure 10.3 for a visualization. Karaoke servers still add random noise requests to protect dead drop access patterns in case of dropped messages.

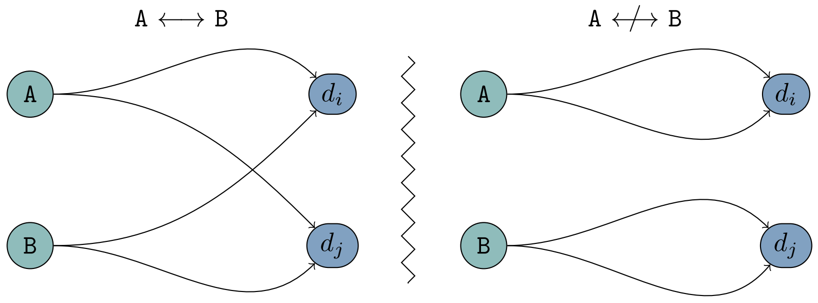


Figure 10.3: Dead drop accesses in Karaoke. Alice and Bob are communicating with each other on the left but have no communication partner on the right. In both cases, two dead drops are accessed twice.

Attack Potential. [4, Sec. 4.3] mentions that the inner servers need to deduplicate requests to prevent replays in the same round. There is however no protection mechanism against replays from previous rounds. The adversary could replay both of Alice's requests from round i in round $i + 1$ and save all accessed dead drop ids from round i. In round $i + 1$, $\mathcal{A}$ counts how many dead drops from the previous round are accessed again. Since the dead drop ids are chosen pseudorandomly, it is highly unlikely for any dead drop to be accessed in consecutive rounds without adversary involvement. If Alice had no communication partner, her requests are send to the same dead drop and $\mathcal{A}$ counts *one* recurring id. If she had a communication partner, her requests are send to two different dead drops and $\mathcal{A}$ counts *two* recurring ids.

Karaoke is arguably even more vulnerable against sender timing attacks than Vuvuzela: If the client is active, two requests are encrypted. If it is inactive, a random secret and two random ciphertexts are chosen. It can be assumed that these two cases do not take the same time to compute, the adversary can therefore tell whether the client is active or not by timing its sending behavior.

Applicability of Fixes. Our proposed protection against later-round replays can be implemented in Karaoke. Karaoke already has a *coordinator*, which announces the start of a new round (including the current round number). With that, it can be assumed that all honest clients have access to the current round number, which they can include in their onion layers. The servers can check if the include round number is valid and discard replayed requests from previous rounds.

The sender timing problems should also be fixable with our proposed strategy from *f*-Vuvuzela:

The random secret key can always be chosen and instead of random ciphertexts, real ciphertexts (containing dummy messages) can be computed.

Expected Analysis Results. Based on our analysis of Vuvuzela, we can make some predictions about which privacy notions can be achieved with Karaoke against different adversaries. For that, we assume that there are no further exploitable weaknesses compared to Vuvuzela. We think that it is reasonable to assume that the results would be at best identical to Vuvuzela: Timing weaknesses can be exploited in the same way. While Karaoke has protection against same round replays, all proposed attacks for Vuvuzela can also be successfully executed with later round replays. Thus, it is reasonable to assume that these adapted attacks work for Karaoke.

10.3. Conclusion

We have shown that neither Stadium nor Karaoke fix all of the issues we found with Vuvuzela. Stadium manages to prevent replay attacks but remains vulnerable to sender timing attacks. Karaoke is still susceptible to both replay (via later-round replay) and sender timing attacks.

During our analysis, we have shown that Vuvuzela can *only* be attacked by timing or replay. At this point, we cannot guarantee that this is also the case with Stadium and Karaoke. A full analysis of the protocols could reveal further weaknesses that are introduced by moving away from a single chain mixnet. Especially Stadium's hybrid verifiable shuffling adds quite a lot of complexity and could be source of yet unknown attacks.

11. Discussion

On the Use of Dead Drops. As we have seen, using dead drops to exchange messages comes with some additional complications. Dead drop access patterns can leak information about communication patters and costly protection measures have to be added. One can ask why dead drops are used in general and what their advantages are compared to other approaches.

If one were to remove the dead drops from Vuvuzela, one could end up with the following protocol (call it *ndd*-VCP[1]): Clients include the address of the intended receiver in place of Vuvuzela's dead drop id. Rather than depositing into dead drops, the last server sends the requests directly to the corresponding receivers. The rest of the protocol works analogously (communication happens in rounds, requests are onion-encrypted and shuffled, etc.). Looking at this protocol, one can find some advantages of dead drops:

- *Dead drops make active attacks a bit harder*: In *ndd*-VCP, $\mathcal{A}$ can replay Alice's request and directly observe which client receives a duplicated request (and therefore is her communication partner). With Vuvuzela, $\mathcal{A}$ at least has to replay the requests of two suspected communication partners to see if they reach the same dead drop. If his suspicion was wrong, $\mathcal{A}$ does not learn with whom the clients are communicating instead.
- *Dead drops simplify the use of server cover traffic.* *ndd*-VCP needs to detect all kinds of request alteration (e.g., $\mathcal{A}$ could drop Alice's request and observes which client does not receive a request). As can be seen in Chapter 9, these protection mechanisms can add significant overhead. The use of server cover traffic might be more efficient, but harder to implement without dead drops: In Vuvuzela, each inner server can add single and double dead drop accesses. Without dead drops, the cover requests have to be send to real clients, otherwise the last server could distinguish cover from real requests. This increases client load (which is not the case in Vuvuzela), since clients have to filter out cover themselves.

There are also inherent disadvantages to using dead drops:

- Dead drops in combination with a mixnet increase the protocol overhead, since every server has to process the requests *twice*. This assumes that the server chain length is identical in both protocols, which is reasonable since both variants require one honest server.
- With Vuvuzela's implementation of dead drops, Bob has to send a request to Alice if he wants to receive her request in turn. In the *ndd*-VCP, Alice can send a request to Bob without required sending on his side (he still needs to send some request to hide his sending patterns, but this can be send to another user). Note that this not necessary if sender and receiver use differing dead drops, as is the case with *2P*-VCP (see Chapter 9).

On the Use of Mixnets. Another component of Vuvuzela that introduces overhead is the mixnet itself. Since only one server has to be trusted, each inner server adds enough cover traffic to hide all changes in communication patterns. If multiple inner servers are honest, too much cover traffic is added, causing unnecessary overhead for later servers. While there are approaches that keep the mixnet, but move cover generation out of it (notably *Stadium* [3], which we discussed in Section 10.1), we'll consider removing the mixnet altogether in the following paragraphs.

If one were to remove the mixnet from Vuvuzela, one would have to add another mechanism to ensure that senders cannot be linked to receivers. *Pung* [18] is a protocol that is reasonably similar

[1] *no dead drop*-Vuvuzela Communication Protocol

to Vuvuzela, but replaces the mixnet by Private Information Retrieval (PIR) [29]. We present an informal definition of PIR based on [18, Sec. 3.3]: PIR allows clients to retrieve items from a dataset stored on a server without the server knowing which items the client is interested in. The client sends a request which prompts the server to compute an answer by performing cryptographic operations over *all* items in the dataset. The client can decode the answer to reveal the item she is interested in.

Pung uses a similar dead drop based strategy as Vuvuzela: Communication occurs in rounds, each clients sends one (real or random cover) request per round directly to a cluster of servers. Each request contains a label, which is Pung's analog to a dead drop id, and an encrypted message. Unlike Vuvuzela, communication partners do not use the same dead drop, but rather each users has a unique (but known to the communication partner) pseudo-random label[2]:

$$label_S(r) = H(k_{ab} \mid r \mid pk_a)$$
$$label_R(r) = H(k_{ab} \mid r \mid pk_b)$$

Clients retrieve messages from their communication partners from the cluster via PIR, using the label as an index.

The main advantage of this PIR-based approach is that the whole infrastructure can be untrusted (as long as sender and receiver are uncompromised). This is due to the fact that the server does not learn which dead drops are accessed by the clients. With mixnet based approaches, at least on server has to remain honest. Apart from trust, choosing a mixnet versus PIR is a question of computational overhead versus network overhead: In Vuvuzela, message exchange only requires computations on the two involved messages (onion decryption at the inner server and the exchange at the last server). PIR requires computations involving *all* requests for *each* retrieval. On the other hand, Vuvuzela requires a large number of extra server cover requests to be processed, where Pung only needs to handle requests from clients, reducing the network overhead. [18, Ch. 4] mentions that that the computational overhead can be reduced by having the clients retrieve multiple requests per query (e.g., for group chats). Still, according to [18, Ch. 9], Vuvuzela achieves significantly better performance than Pung.

[2]This is basically the same strategy as *2P*-VCP uses, see Chapter 9.

12. Conclusion

The main takeaway from this thesis is that Vuvuzela as proposed in [2] does not sufficiently protect user privacy. Only under the *strong* client cover assumption can Communication Unobservability be reached. While this seems like a positive result for Vuvuzela, one has to take into account that the strong client cover assumption is the strongest possible variant of client cover: If two clients are communicating with each other in one scenario, they also have to be communicating in the other scenario. Thus, there are no changes in communication patterns between scenarios and the only information that Vuvuzela needs to protect is the contents of the requests. Under more reasonable assumptions, flaws concerning sender timing and replay of requests can be exploited to break nearly all privacy notions:

- Under the *medium* client cover assumption, $\mathcal{A}_{CM}$ and $\mathcal{A}_{GNM}$ can break $(SR)\overline{L}$ and $(SR)\overline{O}$. $\mathcal{A}_{GT}$ and $\mathcal{A}_{CT}$ can break $S\overline{O} - P$.
- Under the *weak* client cover assumption, $\mathcal{A}_{CM}$ and $\mathcal{A}_{GNM}$ can break any notion except $M\overline{O} - |m|$. $\mathcal{A}_{GT}$ and $\mathcal{A}_{CT}$ can break any notion except $M\overline{O} - |m|$ and $(SR)\overline{L}$.

This is not only a theoretical inconvenience, but has serious consequences in real-life scenarios. Especially the double replay attacks are exploitable without much effort. If an oppressive government suspects that activists Alice and Bob are talking to each other, they can easily confirm their suspicion by replaying Alice's and Bob's requests *once.* The attack is invisible for Alice and Bob, who might face serious consequences.

Integrating Alpenhorn, which was introduced by the same authors to improve dialing and shared key generation, does also not have a positive impact on privacy protection. The dialing part of Alpenhorn is susceptible to the same attacks as the Vuvuzela Dialing protocol ($S\overline{O} - H$ and $(2S)\overline{L}$ can be broken via sender-timing attacks and *any* notion can be broken with replay attacks). Using Alpenhorn's evolving share keys with the communication protocol only prevents attacks, if the used encryption scheme is deterministic, which is not the case for Vuvuzela's prototype implementation.

With f-Vuvuzela, we have shown that these flaws can be fixed with reasonable overhead. Replay protection requires the clients to include the current round number inside of each onion layer and the honest inner server to check it. Further, the honest inner server has to deduplicate the received requests. Timing protection requires modifications to clients and last server behavior: The client has to additionally choose a random public key when sending **M/AC**-Type or **I**-Type requests. Client behavior for **IC**-Type requests does not have to change, **C**-Type requests now require the same steps as **I**-Type requests. The last server has to iterate over all requests twice instead of once. The resulting protocol reaches Communication Unobservability under *all* client cover assumptions.

We have also inspected Stadium and Karaoke, which are follow-up protocols to Vuvuzela. Both protocols introduce measures against replay attacks, but only Stadium considers both same-round and later-round replays. Karaoke (which was published after Stadium) only protects against same-round replays. Both fail to address sender timing differences. As a consequence, we expect Stadium to only be susceptible to timing-based attacks (which still break any notion except $M\overline{O} - |m|$ and $(SR)\overline{L}$ under the weak client cover assumption). Karaoke is expected to be at most as secure as Vuvuzela, since all same-round replay attacks can be adapted to later-round replays.

Another lesson learned from this thesis is that designing an anonymous communication network is hard and that small oversights can have a big impact on privacy. This is actually where a formalized framework such as [1] can help significantly: During an analysis of the protocol against a well defined adversary model, flaws are bound to surface. Having to prove that a privacy notion can be reached against all adversaries should ensure that no issue is overlooked. A formal analysis also helps to find ambiguities and gaps in the protocol definitions, which can lead to bugs in the implementation.

13. Future Work

Challenge Client Corruption. In Section 4.5.1, we have shown that an adversary who is allowed to corrupt challenge clients can break *any* privacy notion. This issue is not addressed by *f*-Vuvuzela or *2P*-VCP, where the same attacks are still possible.

It would be useful to have a variant of Vuvuzela that improves on this aspect. This might be a very challenging task, since many existing protocols (all that where touched upon in this thesis) only make privacy guarantees for honest (i.e., not corrupted) clients. However, there is literature (e.g., [30]) that suggests reaching some sender privacy notions with a corrupted receiver is possible. For example, if the receiver of a message were not able to tell whether the message was send to him by Alice or Bob, Sender Unobservability ($S\overline{O}$) can be reached.

Improving the Efficiency of *2P*-VCP. As discussed in Section 9.2, the advantage of *2P*-VCP (i.e., removing the server cover traffic) is negated by the protection mechanisms against active adversaries. We still feel that there is value in the basic structure of *2P*-VCP.

It would be interesting to investigate alternative protection mechanisms against active attacks. If sufficiently efficient ways for the honest inner server to detect dropped, modified, delayed and replayed requests are found, the benefits of *2P*-VCP will outweigh the costs compared to VCP.

Examining Related Protocols. There are multiple protocols derived from Vuvuzela (e.g., *Stadium* [3] and *Karaoke* [4]). In Chapter 10, we found that both have some form of protection against replay attacks but are still susceptible to sender timing attacks. It would be interesting to do a full analysis of them to determine if the changes from Vuvuzela introduce any other weaknesses and if the replay protection mechanisms are effective under all circumstances.

Group Communication using Dead Drops. Vuvuzela and related protocols are focused on providing anonymous *one-on-one* communication. Popular chat apps such as *Signal* or *Whatsapp* also offer group communication capabilities. While there is existing literature on anonymous group communication (notably *Dissent* [31], but more recently also *Talek* [32]), it might be interesting to extend Vuvuzela. In that case, dead drops could be shared by multiple users, who all share a secret key generated with *Group Diffie-Hellmann* [33]. Alternatively, each group member could deposit into her own dead drop, which in turn is accessed by all other group members. With both approaches, one has to make sure that group size cannot be determined from dead drop access patterns, which might be difficult.

14. References

[1] C. Kuhn, M. Beck, S. Schiffner, E. Jorswieck, and T. Strufe, "On privacy notions in anonymous communication," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 2, pp. 105–125, Apr. 2019 [Online]. Available: `http://dx.doi.org/10.2478/popets-2019-0022`

[2] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich, "Vuvuzela: Scalable private messaging resistant to traffic analysis," in *Proceedings of the 25th symposium on operating systems principles*, 2015, pp. 137–152 [Online]. Available: `http://doi.acm.org/10.1145/2815400.2815417`

[3] N. Tyagi, Y. Gilad, D. Leung, M. Zaharia, and N. Zeldovich, "Stadium: A distributed metadata-private messaging system," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 423–440 [Online]. Available: `http://doi.acm.org/10.1145/3132747.3132783`

[4] D. Lazar, Y. Gilad, and N. Zeldovich, "Karaoke: Distributed private messaging immune to passive traffic analysis," in *13th {usenix} symposium on operating systems design and implementation ({osdi} 18)*, 2018, pp. 711–725.

[5] E. Snowden, *Permanent record.* New York: Metropolitan Books/Henry Holt; Company, 2019.

[6] D. L. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Communications of the ACM*, vol. 24, no. 2, pp. 84–90, 1981.

[7] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," Naval Research Lab Washington DC, 2004.

[8] M. K. Reiter and A. D. Rubin, "Crowds: Anonymity for web transactions," *ACM transactions on information and system security (TISSEC)*, vol. 1, no. 1, pp. 66–92, 1998.

[9] C. Kuhn and M. Beck, "Common adversaries," *Technical report*, 2019.

[10] C. E. Shannon, "Communication theory of secrecy systems," *Bell system technical journal*, vol. 28, no. 4, pp. 656–715, 1949.

[11] D. Bernstein, "Salsa20 security," *`https://cr.yp.to/snuffle/security.pdf`*, 2005.

[12] D. J. Bernstein, "Extending the salsa20 nonce," in *Workshop record of symmetric key encryption workshop*, 2011, vol. 2011.

[13] B. Dawson, "The surprising subtleties of zeroing a register." `https://randomascii.wordpress.com/2012/12/29/the-surprising-subtleties-of-zeroing-a-register/`; Random ASCII, 2012.

[14] S. Malladi, J. Alves-Foss, and R. B. Heckendorn, "On preventing replay attacks on security protocols," IDAHO UNIV MOSCOW DEPT OF COMPUTER SCIENCE, 2002.

[15] D. Lazar and N. Zeldovich, "Alpenhorn: Bootstrapping secure communication without leaking metadata," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 571–586 [Online]. Available: `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/lazar`

[16] M. Bellare, A. Boldyreva, A. Desai, and D. Pointcheval, "Key-privacy in public-key encryp-

tion," in *International conference on the theory and application of cryptology and information security*, 2001, pp. 566–582.

[17] E. Wilde and A. Vaha-Siplia, "URI scheme for global system for mobile communications (gsm) short message service (sms)," RFC Editor; Internet Requests for Comments; RFC Editor, RFC 5724, Jan. 2010 [Online]. Available: `https://www.rfc-editor.org/rfc/rfc5724.txt`

[18] S. Angel and S. Setty, "Unobservable communication over fully untrusted infrastructure," in *12th {usenix} symposium on operating systems design and implementation ({osdi} 16)*, 2016, pp. 551–569.

[19] J. K. Liu, V. K. Wei, and D. S. Wong, "Linkable spontaneous anonymous group signature for ad hoc groups," in *Australasian conference on information security and privacy*, 2004, pp. 325–335.

[20] M. H. Au, S. S. Chow, W. Susilo, and P. P. Tsang, "Short linkable ring signatures revisited," in *European public key infrastructure workshop*, 2006, pp. 101–115.

[21] J. Camenisch and M. Stadler, "Efficient group signature schemes for large groups," *Lecture Notes in Computer Science*, pp. 410–424, 1997.

[22] P. P. Tsang and V. K. Wei, "Short linkable ring signatures for e-voting, e-cash and attestation," in *International conference on information security practice and experience*, 2005, pp. 48–60.

[23] X. Lu, M. H. Au, and Z. Zhang, "Raptor: A practical lattice-based (linkable) ring signature," in *International conference on applied cryptography and network security*, 2019, pp. 110–130.

[24] H. Zheng, Q. Wu, B. Qin, L. Zhong, S. He, and J. Liu, "Linkable group signature for auditing anonymous communication," in *Australasian conference on information security and privacy*, 2018, pp. 304–321.

[25] R. W. Lai, K.-F. Cheung, S. S. Chow, and A. M.-C. So, "Another look at anonymous communication," *IEEE Transactions on Dependable and Secure Computing*, vol. 16, no. 5, pp. 731–742, 2018.

[26] R. L. Rivest, A. Shamir, and Y. Tauman, "How to leak a secret," in *International conference on the theory and application of cryptology and information security*, 2001, pp. 552–565.

[27] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford, "Atom: Horizontally scaling strong anonymity," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 406–422.

[28] A. Kwon, D. Lu, and S. Devadas, "{XRD}: Scalable messaging system with cryptographic privacy," in *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)*, 2020, pp. 759–776.

[29] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *Proceedings of ieee 36th annual foundations of computer science*, 1995, pp. 41–50.

[30] G. Danezis and I. Goldberg, "Sphinx: A compact and provably secure mix format," in *2009 30th ieee symposium on security and privacy*, 2009, pp. 269–282.

[31] H. Corrigan-Gibbs and B. Ford, "Dissent: Accountable anonymous group messaging," in *Proceedings of the 17th acm conference on computer and communications security*, 2010, pp. 340–350.

[32] R. Cheng *et al.*, "Talek: Private group messaging with hidden access patterns," *arXiv preprint arXiv:2001.08250*, 2020.

[33] M. Steiner, G. Tsudik, and M. Waidner, "Diffie-hellman key distribution extended to group communication," in *Proceedings of the 3rd acm conference on computer and communications se-*

curity, 1996, pp. 31–37.

15. Appendix

15.1. Differential Privacy [2, Sec. 2.2]

Definition 7 (Differential Privacy)**.** A randomized algorithm M is (ε, δ)-differentially private for adjacent inputs x and y if, for all sets of outcomes S,

$$Pr[M(x) \in S] \leq e^{\varepsilon} \cdot Pr[M(y) \in S] + \delta$$

Inputs x and y are user actions (i.e., which users are communicating). Two inputs are adjacent, if they only differ in the message exchanged by one user.

[2, Sec. 6.4] suggests that reasonable values for ε are $\ln 2$ and 10^{-4} for δ.

15.2. Pseudocode for the Vuvuzela Communication Protocol

The pseudocode for the Vuvuzela communication protocol is split into algorithms for clients (Algorithm 5), inner servers (Algorithm 6) and last servers (Algorithm 7).

Algorithm 5 VCP: Client Protocol

```
1: function ConstructRequest
2:     if Client has no partner then
3:         choose random $pk_b$
4:     compute shared key $k_{ab} \leftarrow DH(sk_a, pk_b)$
5:     compute dead drop id $d^r_{ab} \leftarrow H(k_{ab} \mid r)$
6:     if Client has message $m$ to send then
7:         pad message $m$ if necessary
8:     else
9:         $m = \mathbf{0}$
10:    return $req = (d^r_{ab}, Enc_{k_{ab}}(m))$
11: function OnionWrap(req)
12:    for $i \in$ Servers.reverse do
13:        choose random temporary keys $pk^t_i, sk^t_i$
14:        compute shared key $k_{ai} \leftarrow DH(sk^t_i, pk_i)$
15:        $req = (pk^t_i, Enc_{k_{ai}}(req))$
16:    return $req$
17: function Main
18:    for $i \in \{1, \ldots, \nu\}$ do
19:        $req$ = ConstrucRequest
20:        $req$ = OnionWrap($req$)
21:        Send $req$ to first server
22:    Wait for requests to return
23:    Unpack requests
```

Algorithm 6 VCP: Inner server protocol

```
function DecryptRequest($req = (pk_i^t, c)$)
    save $pk_i^t$
    $k_{ai} \leftarrow DH(pk_i^t, sk_i)$
    $req \leftarrow Dec_{k_{ai}}(c)$
    return $req$
function GenerateCover
    sample $n_1, n_2$ from Laplace distribution
    for $i \in \{0, \dots, \lceil n_1 \rceil\}$ do                ▷ Generate $n_1$ single dead drop accesses
        choose random keys $sk_x, pk_x$ and $sk_y, pk_y$
        $k_{xy} \leftarrow DH(sk_x, pk_y)$
        $d_{xy}^r \leftarrow H(k_{xy}, r)$
        choose random $m$
        $req \leftarrow (d_{xy}^r, Enc_{k_{xy}}(m))$
        onion wrap $req$ (for all remaining servers)
        add $req$ to list
    for $i \in \{0, \dots, \lceil n_2/2 \rceil\}$ do              ▷ Generate $n_2$ double dead drop accesses
        choose random keys $sk_x, pk_x$ and $sk_y, pk_y$
        $k_{xy} \leftarrow DH(sk_x, pk_y)$
        $d_{xy}^r \leftarrow H(k_{xy}, r)$
        choose random $m_1, m_2$
        $req_1 \leftarrow (d_{xy}^r, Enc_{k_{xy}}(m_1))$
        $req_2 \leftarrow (d_{xy}^r, Enc_{k_{xy}}(m_2))$
        onion wrap $req_1$ and $req_2$ (for all remaining servers)
        add $req_1$ and $req_2$ to list
function ProcessRound
    collect requests
    for all $req \in$ requests do
        $req \leftarrow$ DecryptRequest($req$)
    GenerateCover
    choose random permutation $\pi_i$ and shuffle requests
    send requests to next server & wait for response
    reorder requests according to $\pi_i^{-1}$
    remove cover
    build up onion layer
    send to previous server                  ▷ or in case of the first server to the clients
```

Algorithm 7 VCP: Last server protocol

```
function ProcessRound
    collect requests
    for all $req \in$ requests do DecryptRequest($req$)              ▷ See Algorithm 6
    for all $req = (d, c) \in$ requests do                          ▷ Access dead drops
        if $\exists\, req' = (d, c')$ then                  ▷ other request to the same dead drop
            $req = (d, c')$
            $req' = (d, c)$
            remove $req$ and $req'$ from search
        else
            choose random $c'$
            $req = (d, c')$
            remove $req$ from search
    build up onion layers
    send requests to previous server
```

15.3. Pseudocode for the Vuvuzela Dialing Protocol

The pseudocode for the Vuvuzela dialing protocol is split into algorithms for clients (Algorithm 8), inner servers (Algorithm 9) and last servers (Algorithm 10).

Algorithm 8 VDP: Client Protocol

```
function ConstructRequest
    if Client has no partner to invite then
        Choose random $c$
        return $req = (\bar{d}, c)$
    else
        compute dead drop id $d \leftarrow H(pk_b) \mod \mu$
        compute nonce, MAC
        return $req = (d, Enc_{pk_b}(pk_a, \text{nonce, MAC}))$
function Main
    for $i \in \{1, \dots, \nu\}$ do
        $req$ ←ConstructRequest
        $req$ ←OnionWrap($req$)                              ▷ See Algorithm 5
        send $req$ to first server
```

Algorithm 9 VDP: Inner server protocol

```
function DecryptRequest($req = (pk_i^t, c)$)
    save $pk_i^t$
    $k_{ai} \leftarrow DH(pk_i^t, sk_i)$
    $req \leftarrow Dec_{k_{ai}}(c)$
    return $req$
function GenerateCover
    for $i \in \{1, \dots, \mu\}$ do
        sample $n$ from Laplace distribution
        for $j \in \{1, \dots, n\}$ do
            choose random $pk_x, pk_y$, nonce, MAC
            $req = (j, Enc_{pk_x}(pk_y, \text{nonce, MAC}))$
            onion wrap $req$ from remaining servers
function ProcessRound
    collect requests
    for all $req \in$ requests do
        $req \leftarrow$ DecryptRequest($req$)
    GenerateCover
    choose random permutation $\pi_i$ and shuffle requests
    send requests to next server & wait for response
```

15.4. Attack: $\mathcal{A}_{GT}$ vs. $(SR)\overline{L}$

Table 15.1: Challenge and cover communications for the attack on $(SR)\overline{L}$ by $\mathcal{A}_C$.

		Scenario 0	Scenario 1
Batch 0		$(alice, bob, m, aux)$	$(alice, bob, m, aux)$
		$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
		$(carol, bob, 0, aux)$	$(carol, bob, 0, aux)$
		$(bob, carol, 0, aux)$	$(bob, carol, 0, aux)$
		$(carol, dave, 0, aux)$	$(carol, dave, 0, aux)$
		$(dave, carol, 0, aux)$	$(dave, carol, 0, aux)$
		$(alice, dave, 0, aux)$	$(alice, dave, 0, aux)$
		$(dave, alice, 0, aux)$	$(dave, alice, 0, aux)$
Batch 1	Instance 0	$(alice, bob, m, aux)$	$(alice, dave, m, aux)$

	Scenario 0	Scenario 1
	$(carol, dave, m, aux)$	$(carol, bob, m, aux)$
	$(bob, alice, 0, aux)$	$(bob, alice, 0, aux)$
	$(carol, bob, 0, aux)$	$(carol, dave, 0, aux)$
	$(bob, carol, 0, aux)$	$(bob, carol, 0, aux)$
	$(dave, alice, 0, aux)$	$(dave, alice, 0, aux)$
	$(dave, carol, 0, aux)$	$(dave, carol, 0, aux)$
	$(alice, dave, 0, aux)$	$(alice, dave, 0, aux)$
Instance 1	...	...

15.5. Detecting Duplicates in Linear Time

Given list e of n elements, the goal is to detect and eliminate all duplicates from the list in time complexity $\mathcal{O}(n)$.

This can be done using a collision resistant hash function H:

```
1: $t \leftarrow [0, \ldots, 0]$
2: for all $i \in e$ do
3:     $h \leftarrow H(i)$
4:     if $t[h] = 0$ then
5:         $t[h] \leftarrow 1$
6:     else
7:         remove $i$ from $e$
```

This algorithm only needs to traverse the list once and thus has a time complexity in $\mathcal{O}(n)$.

Algorithm 10 VDP: Last server protocol

```
1: function ProcessRound
2:     collect requests
3:     for all $req \in$ requests do
4:         $req \leftarrow$ DecryptRequest($req$)
5:     GenerateCover
6:     choose random permutation $\pi_i$ and shuffle requests
7:     add each request to corresponding dead drop
```