Dezentralisierte asymmetrische Verschlüsselung über Tor

Die Lösung für sicheres Messaging?

Hendrik Lind

Facharbeit

Windthorst-Gymnasium Meppen

Seminarfach Informatik

17. Februar 2024

Inhaltsverzeichnis

1 Einleitung

Russland, China, Iran. In all diesen totalitären Staaten herrscht eine starke Zensur [vgl. Am23]. Rund 1,7 Milliarden Menschen sind allein nur in diesen drei Staaten von der Einschränkung der Meinungsfreiheit betroffen [vgl. Un22]. Wie können Bürger dieser Staaten ihre Meinung also verbreiten und andere Staaten auf staatskritische Probleme aufmerksam machen, ohne sich selber in Gefahr zu bringen?

Bei herkömmlichen Messengern, wie WhatsApp, Signal und Co., braucht die Außenwelt die Telefonnummern der im totalitären Staat wohnenden Bürgern und Reportern, um diese zu kontaktieren. Allerdings könnte ein totalitärer Staat, sich als Empfänger ausgeben, sodass Bürger/Reporter ihre private Nummer an den Staat überreichen und dieser somit jene Nummer rückverfolgen kann [vgl. Fä23]. Und genau hier liegt das Problem: Bürger und Reporter können nicht durch alltägliche Messenger mit der Außenwelt kommunizieren, da der Staat deren Nummer zurückverfolgen kann und somit weiter die Meinungsfreiheit einschränkt und unterbindet [vgl. ebd.].

Durch die zentrale Infrastruktur, welche die meisten Messenger, wie zum Beispiel WhatsApp und Signal verwenden, ist es für totalitäre Staaten, wie China, möglich, die IP-Adressen jener Server zu blockieren und somit für Bürger und Reporter unzugänglich zu machen [vgl. Wu+23; Bh23].

Ein dezentralisierter Messenger, welcher Ende-zu-Ende verschlüsselt ist und über das Tor-Netzwerk kommuniziert, könnte bei diesen Problemen eine Lösung sein. Die Frage, ob ein solcher Messenger die Lösung für Bürger eines totalitären Staates ist, soll in dieser Arbeit geklärt werden.

Um diese Frage beantworten zu können, beschäftigt sich diese Arbeit in dem zweiten Kapitel mit der asymmetrischen Verschlüsselung, welche benötigt wird um die Ende-zu-Ende-Verschlüsselung (E2EE) umzusetzen und die Definition der E2EE, sowie der Sicherheitsbetrachtung der asymmetrischen Verschlüsselung [vgl. LB21]. Die Arbeit geht dabei nicht auf weitere Padding-Verfahren ein. Eine mögliche Lösung, um eine Anonymität über das Internet zu gewährleisten wird in Kapitel drei vorgeschlagen, wobei das Tor-Netzwerk eine wichtige Rolle spielt. Diese Arbeit befasst sich im vierten Kapitel mit einer Dezentralisierung der Infrastruktur, um eine weitere Sicherheitsebene zu schaffen. Zuletzt werden im fünften Kapitel die Vor-

und Nachteile eines solchen Messengers betrachtet, im sechsten Kapitel wird eine mögliche Umsetzung des Messengers beschrieben und im siebten Kapitel wird ein Fazit gezogen.

2 Asymmetrische Verschlüsselung

Um einen sicheren Nachrichtenaustausch zu gewährleisten, wird in dieser Arbeit die E2EE implementiert. Bei der E2EE wird von dem Sender die Nachricht, bevor sie an den Empfänger geschickt wird, verschlüsselt [vgl. Gr14]. Dazwischenliegende Akteure, wie zum Beispiel Server oder mögliche Angreifer, können demzufolge die Nachricht nicht lesen [vgl. ebd.]. **Nur** der Empfänger der Nachricht kann diese auch entschlüsseln. Als Ent- und Verschlüsselungsverfahren der Nachrichten wird die asymmetrische Verschlüsselung verwendet [vgl. ebd.]. Diese Arbeit beschränkt sich bei der asymmetrischen Verschlüsselung auf das RSA-Verfahren.

2.1 Grundlagen

Grundsätzlich gibt es bei der asymmetrischen Verschlüsselung ein Schlüsselpaar (Keypair), welches aus einem privaten Schlüssel (private key) und einem öffentlichen Schlüssel (public key) besteht [vgl. BSW15b]. Diese beiden Schlüssel hängen mathematisch zusammen, sodass der öffentliche Schlüssel Nachrichten **nur** verschlüsseln aber nicht entschlüsseln kann [vgl. ebd.]. **Nur** der zum Schlüsselpaar dazugehörige private Schlüssel ist in der Lage, die verschlüsselte Nachricht wieder zu entschlüsseln (siehe Abb. 1) [vgl. ebd.].

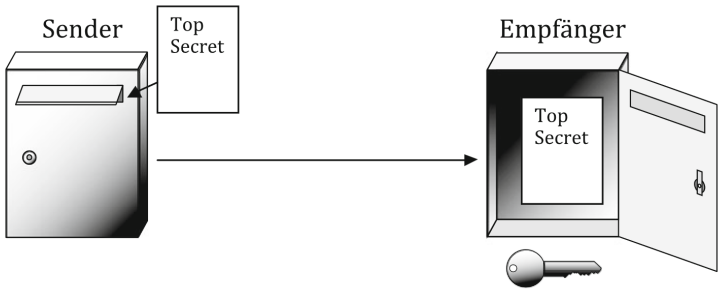

Abbildung 1: Jeder Sender kann mit dem öffentlichen Schlüssel die Nachricht „verschlüsseln“ (also eine Nachricht in den Briefkasten werfen), aber nur der Empfänger kann den Briefkasten mit seinem privaten Schlüssel öffnen und somit die Nachricht herausnehmen[vgl. BSW15a]

2.2 Mathematische Betrachtung

Alle Variablen der folgenden Berechnungen liegen im Bereich $\mathbb{N}$ [vgl. Is16].

Für die Generierung des Schlüsselpaares benötigen wir zuerst zwei große zufällige Primzahlen, P und Q [vgl. ebd.]. Daraus ergibt sich $n = P*Q$, wobei $P \neq Q$, sodass P bzw. Q nicht durch $\sqrt{n}$ ermittelt werden kann [vgl. ebd.]. Der private Schlüssel besteht aus den Komponenten $\{n, d\}$ währenddessen der öffentliche Schlüssel aus $\{n, e\}$ besteht [vgl. Wa+13].

2.2.1 Eulersche Phi-Funktion

Die Eulersche Phi-Funktion spielt eine wichtige Rolle in dem RSA-Verfahren [vgl. Tu08]. Grundsätzlich gibt $\phi(x)$ an, wie viele positive teilerfremde Zahlen bis x existieren (bei wie vielen Zahlen der größte gemeinsamer Teiler (gcd) 1 ist) [vgl. Ta13]. Somit ergibt $\phi(6) = 2$ oder bei einer Primzahl $\phi(7) = 7-1 = 6$ somit $\phi(x) = x-1$, wenn x eine Primzahl ist, da jede Zahl kleiner als x teilerfremd sein muss [vgl. Tu08].

$$\phi(n) = \phi(P \cdot Q)$$
$$\phi(n) = \phi(P) \cdot \phi(Q)$$
$$\phi(P) = P - 1 \qquad \phi(Q) = Q - 1$$
$$\phi(n) = (P-1) \cdot (Q-1)$$

2.2.2 Generierung des Schlüsselpaares

Sowohl der private als auch der öffentliche Schlüssel besteht unter anderem aus folgender Komponente: $n = P \cdot Q$ [vgl. Ta96]. Für den öffentlichen Schlüssel benötigen wir die Komponente e, die zur Verschlüsselung einer Nachricht verwendet wird [vgl. ebd.]. e ist hierbei eine zufällige Zahl, bei welcher folgende Bedingungen gelten [vgl. ebd.]:

$$e = \begin{cases} 1 < e < \phi(n) \\ \text{gcd}(e, \phi(n)) = 1 \\ e \text{ kein Teiler von } \phi(n) \end{cases}$$

Mit der errechneten Komponente e, welche Nachrichten verschlüsselt, kann der öffentliche Schlüssel nun an den Sender übermittelt werden.

Um den privaten Schlüssel zu berechnen, benötigen wir die Komponente d, wel-

che zur Entschlüsselung verwendet wird [vgl. MS13].

$$\phi(n) = (P-1)(Q-1)$$
$$e \cdot d = 1 \mod \phi(n)$$

2.3 Sicherheit

Um die Sicherheit des RSA-Verfahrens betrachten zu können, müssen wir nun den Ver-/Entschlüsselungsvorgang betrachten.

$c = m^e \mod n$	Verschlüsselung zu c mit m als Nachricht
$m = c^d \mod n$	Umkehroperation (Entschlüsslung) von c zu m

Um die verschlüsselte Nachricht c zu entschlüsseln, bräuchte ein Angreifer die Komponente des privaten Schlüssels d. d ist allerdings mit einem starken Rechenaufwand verbunden, da, wie schon vorher bereits gezeigt, dafür $\phi(n)$ kalkuliert werden müsste. Somit wird eine Primfaktorzerlegung von n benötigt wird [vgl. Ta13]. Bei der Verschlüsselung von m zu c liegt eine Trapdoor-Einwegfunktion vor [vgl. Kr16a]. Das bedeutet, dass es zwar leicht ist $f(x) = i$ zu berechnen (bei RSA: Verschlüsselung), es jedoch unmöglich ist von i auf den Ursprungswert x zu schließen, ohne dass weitere dafür notwendigen Komponente bekannt sind (bei RSA wäre die benötigte Komponente d) [vgl. ebd.].

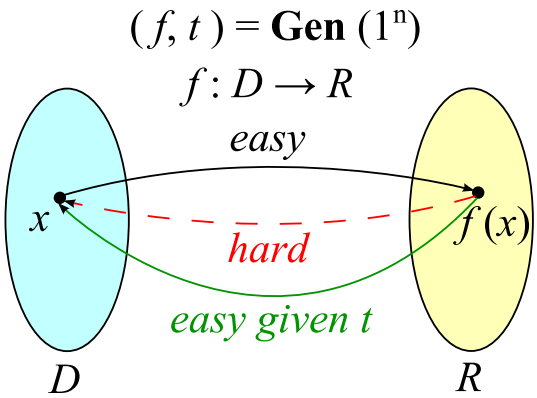

Abbildung 2: Die Trapdoor-Einwegfunktion bildlich dargestellt[vgl. Kr16b]

Wichtig bei dem RSA-Verfahren ist, dass die Länge von n (die Schlüssellänge) mindestens 3000 Bit betragen sollte, da sonst die Primfaktorzerlegung von n mit modernen Computern möglich sein könnte [vgl. Si23].

2.4 Vergleich zur symmetrischen Verschlüsselung

Bei der symmetrischen Verschlüsselung wird der gleiche Schlüssel sowohl für die Verschlüsselung als auch für die Entschlüsselung verwendet [vgl. IBM21]. Im Vergleich zu der asymmetrischen Verschlüsselung, ist die symmetrische Verschlüsselung schneller und keine Beschränkung des Chiffretextes [vgl. HK20; Op24]. Jedoch muss der Schlüssel der symmetrischen Verschlüsselung sicher an den jeweils anderen Kommunikationspartner übermittelt werden, um Nachrichten zu entschlüsseln [vgl. ebd.].

3 Das Tor-Netzwerk

Die Anonymität des Messengers ist ein weiterer zentraler Aspekt, um die jeweiligen Kommunikationspartner zu schützen. Das Tor-Netzwerk ist hierbei ein möglicher Lösungsansatz, da normalen Routing, wie wir es tagtäglich nutzen, der Client kommuniziert direkt mit dem Zielserver, somit der Zielserver die IP-Adresse des Clients einsehen kann [vgl. El16]. Eine Rückverfolgung der IP-Adresse ist somit möglich [vgl. Fä23]. Und genau bei diesem Problem setzt das Tor-Netzwerk an. Das Tor-Netzwerk besteht hierbei aus vielen *Nodes* , welche eingehende Tor-Verbindungen akzeptieren und weiterverarbeiten. Damit ein Client eine Anfrage über das Netzwerk verschicken kann, sucht er zunächst einen Pfad durch das Netzwerk, genannt *Circuit* [vgl. To24d]. Der *Circuit* besteht dabei meist aus drei *Nodes* und ist für 10 Minuten gültig, bis der Client das *Circuit* erneuert (also einen neuen Pfad „sucht“) [vgl. To24c]. Die drei *Nodes* werden klassifiziert in einer *Entry Node* , einer *Relay Node* und einer *Exit Node* , worüber später Anfragen an die Zielserver geschickt werden können [vgl. ebd.]. Dabei verschlüsselt der Client verschlüsselt die eigentliche Anfrage mehrmals, hüllt sie also in ein mehrere „Schalen“ ein, welche eine *Onion* bilden, und leitet diese über den *Circuit* an den Zielserver weiter[vgl. DMS04]. Anfangs wird die *Onion* von dem Tor-Client an die *Entry Node* geschickt [vgl. ebd.]. Bei jeder *Node* , wie der *Entry Node* , wird eine Schale der *Onion* „geschält“ (die *Onion* also einmal entschlüsselt), welche Informationen zu dem nächsten Knotenpunkt enthält [vgl. Au18]. Die Node leitet nun die um eine Schale „geschälte“ *Onion* an den nächsten Knotenpunkt weiter [vgl. ebd.]. Sobald die Anfrage bei der *Exit Node* angekommen ist, entfernt diese die letzte „Schale“ der Anfrage, welche nun

vollständig entschlüsselt ist und an den Zielserver geschickt werden kann [vgl. ebd.]. Dieses Routing-Verfahren ist auch als *Onion-Routing* bekannt [vgl. DMS04].

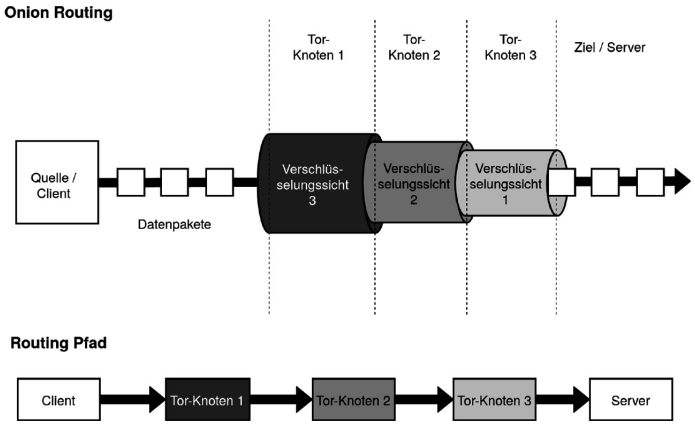

Abbildung 3: Das Prinzip des Onion-Routings [vgl. Wi+22a]

Nur die *Entry Node* weiß somit die reale IP-Adresse des Clients und **nur** die *Exit Node* weiß, an welchen Zielserver die Anfrage geschickt wurde [vgl. Au18]. Dazwischenliegende *Nodes* , wie die *Relay Node* erkennen nur eine verschlüsselte Anfrage und haben keinerlei Informationen über den eigentlichen Client oder den Zielserver. Die *Exit Node* schickt allerdings die Anfrage ohne Verschlüsselung des Tor-Netzwerkes an den Zielserver, wodurch Informationen der Nutzer ausgelesen werden könnten (wenn die Anfrage nicht über das HTTPS-Protokoll versendet wurde) und zusätzlich die IP-Adresse des Zielservers speichern [vgl. Ch+11]. Durch Etablieren von *Exit Nodes* in dem Tor-Netzwerk könnten Angreifer somit die Anonymität des Netzwerkes bzw. der Nutzer gefährden [vgl. ebd.].

3.1 Onion Services

Onion Services sind eine mögliche Lösung für den *Exit Node* -Angriff [vgl. To24g]. Diese nicht auf die *Exit Node* angewiesen sind und agieren nur innerhalb des Tor-Netzwerkes mit anderen Tor-Clients und verhalten sich wie normale Tor-Clients [vgl. ebd.]. Onion Services sind nicht über das normale Internet erreichbar, wie die Zielserver im vorherigen Beispiel, sondern nur über das Tor-Netzwerk [vgl. To24f]. Bei Onion Services müssen im Vergleich zu normalen öffentlichen Servern keine Ports geöffnet werden, damit ein Client sich mit dem Server verbinden kann, da der Onion Service direkt mit dem Tor-Netzwerk über ausgehende Verbindungen kommuniziert (auch bekannt als *NAT punching*) und darüber sämtliche Daten geleitet werden

[vgl. ebd.].

3.1.1 Verbindungsaufbau

Zunächst generiert der Onion Service ein Schlüsselpaar, bestehend aus einem öffentlichen und einem privaten Schlüssel [vgl. Ge23b]. Unter anderem wird nun aus dem öffentlichen Schlüssel die Adresse des Onion Services generiert und endet mit „.onion“ [vgl. ebd.]. Ein Beispiel für eine solche Adresse ist die der Suchmaschine DuckDuckGo:
duckduckgogg42xjoc72x3sjasowoarfbgcmvfimaftt6twagswzczad.onion [vgl. Du24]. Der Onion Service sich nun mit dem Tor-Netzwerk wie ein normaler Client über einen *Circuit* , welcher aus drei *Nodes* besteht [vgl. To24f]. Der Service sendet eine Anfrage an das letzte Relay im *Circuit* , sodass es als *Introduction Point* dient und etabliert eine Langzeitverbindung jenem (der *Circuit* erneuert sich also nicht alle 10 Minuten wie bei dem Tor-Client) [vgl. ebd.]. Dieser Vorgang wiederholt sich zweimal, bis der Onion Service drei *Introduction Points* auf drei verschiedenen *Circuits* gefunden hat [vgl. ebd.]. Damit andere Clients den Onion Service erreichen können, erstellt der Onion Service einen *Onion Service descriptor*, welcher die Adressen der *Introduction Points* und Authentifizierungsschlüssel enthält, signiert diesen mit seinem privaten Schlüssel und schickt den *Descriptor* an die Directory Authority [vgl. To24a; To24e]. Die Directory Authority ist ein Server, welcher Informationen über das Tor-Netzwerk, wie zum Beispiel die des Onion Services, speichert und verteilt [vgl. To24b]. Im Sinne dieser Arbeit, hat der Onion Service sich nun von Person A mit dem Tor-Netzwerk verbunden, jedoch braucht es noch Person B, welche sich über ihren Tor-Client mit dem Onion Service der Person B verbindet. Damit der Tor-Client von Person A sich allerdings verbinden kann, fragt der Client nun die Directory Authority an den signierten *Descriptor* des Onion Services von Person B an den Client zu schicken [vgl. K 23]. Der Client besitzt nun die Adressen der *Introduction Points* und die Signatur des *Descriptors*, sodass dieser mit dem öffentlichen Schlüssel der Onion Adresse die Signatur überprüfen kann [vgl. ebd.]. Der Client generiert nun 20 zufällige Bytes (Secret) und schickt diese an eine zufällig ausgewählte *Relay Node* , welche nun als *Rendezvous Point* dient [vgl. To23b]. Das Secret wird von dem Client auch an eine von den *Introduction Points* geschickt, sodass der Onion Service mit dem gleichen Secret eine Verbindung zu dem *Rendezvous*

Point aufbauen kann [vgl. To23a]. Der *Rendezvous Point* leitet, wenn der Client und der Onion Service über deren *Circuits* miteinander verbunden sind, die Nachrichten zwischen den beiden weiter [vgl. To23b]. Der Client und der Onion Service kommunizieren nun also nur über das Tor-Netzwerk miteinander und sind nicht mehr auf die *Exit Node* angewiesen.

3.2 Sicherheit

In der Sicherheitsbetrachtung beziehe ich mich ausschließlich auf die Verbindung zwischen Onion Services und Tor-Clients (also nicht zwischen Tor-Client und *Exit Node*).

Das Tor-Netzwerk in Verbindung mit den Onion Services liefert, wie bereits erläutert, durch *Circuits* und *Rendezvous Point* eine hohe Anonymität. Im Gegenzug sorgt die hohe Anonymität, welche durch die vielen *Nodes* und durch mehrfache Verschlüsselungen erreicht wird, auch dazu, dass das Tor-Netzwerk um das 120-fache langsamer ist im Vergleich zum normalen Routing [vgl. LSS10]. Das Tor-Netzwerk ist praktisch unmöglich in der Praxis zu blockieren, da über 6.500 verschiedene *Nodes* existieren und es somit keinen zentralen Hauptserver gibt, von welchem das Tor-Netzwerk abhängig ist [1] [vgl. Wi+22b]. Theoretisch ist eine Deanonymisierung eines Onion Services möglich, wenn der ISP (Internet Service Provider, also z.B. Telekom, EWE, etc.) eingehende und ausgehende Verbindungen zwischen Client und Onion Service bzw. zu deren *Relay Node* aufzeichnet und diese mithilfe von Machine Learning analysiert [vgl. Lo+24]. Dabei war die Umsetzung der theoretischen Grundlage bis jetzt mit idealisierten Bedingungen erfolgreich, jedoch braucht es noch weiterer Studien, um die möglichen Gefahren dieses Algorithmus einzustufen [vgl. ebd.]. Dabei könnten, wenn die deutsche, amerikanische und die französische Regierung zusammenarbeiten, 78,34 % der *Circuits* deanonymisieren [vgl. ebd.]. Dieses Szenario erscheint jedoch unrealistisch, da Bürger in Deutschland, in den USA und in Frankreich ein Recht auf Meinungsfreiheit besitzen und höchstwahrscheinlich keine Deanonymisierungsangriff starten, um die Privatsphäre zum Beispiel von Whistleblowern zu schützen [vgl. Am23; Re24].

[1] Es ist zwar möglich die IP-Adressen jeglicher *Relay Nodes* zu blockieren, jedoch ist dies durch Pluggable Transports umgänglich [vgl. Pa+16]

4 Dezentralisierung

Mit den vorgeschlagenen Konzepten ist der Messenger imstande anonym (durch das Tor-Netzwerk und Onion Services) und sicher (durch asymmetrische Verschlüsselung) zu kommunizieren, jedoch wurde die Infrastruktur des Messengers noch nicht behandelt. Dafür ist die Betrachtung und Abwägung eines zentrales bzw. dezentrales Netzwerkes notwendig.

Ein zentrales Netzwerk, zum Beispiel das Netzwerk von WhatsApp oder Signal, besteht aus einem Hauptserver, welcher Daten verarbeitet und mit welchem sich Clients verbinden können [vgl. La22; Si16; Ge21]. Der Client muss bei dieser Netzwerkstruktur dem Hauptserver „vertrauen“ und kann nicht überprüfen, ob der Server von einem Hacker kompromittiert wurde [vgl. Se19]. Dezentrale Netzwerke bestehen hingegen aus mehreren Servern, auf welche Rechenoperationen und oder Daten aufgeteilt werden [vgl. Li23].

4.1 Sicherheit

Im Beispiel eines Messengers bietet ein zentrales Netzwerk den Hauptserver als mögliche Angriffsfläche an, sodass diese entweder durch DDoS-Angriffe außer Betrieb gesetzt werden kann oder durch Kompromittierung des Servers sowohl der Empfänger als auch der Sender der Nachrichten ausgelesen werden kann (in Form von Onion-Adressen)[vgl. Bu24]. Ein dezentrales Netzwerk mit einer Peer-to-Peer-Architektur[2] hat hingegen eine größere Angriffsfläche, wodurch mehrere Knotenpunkte des Netzwerkes kompromittiert werden müssen, um das Netzwerk ausschalten zu können [vgl. Ge23a]. Am Beispiel des Messengers, sollten nur Gesprächsteilnehmer Teil des Netzwerkes sein, um zu verhindern, dass ein Angreifer durch simples Beitreten des Netzwerkes die Onion-Adressen der Gesprächsteilnehmer auslesen kann.

5 Umsetzung des Messengers

Um den Messenger umzusetzen, müssen zunächst die Anforderung des Messengers bekannt sein. Somit sollten folgende Kriterien erfüllt sein:

[2] Peer-to-Peer bedeutet, dass jeder Knotenpunkt im Netzwerk sowohl Client als auch Server sein kann, welche bidirektional kommunizieren [vgl. HD05]

Einfache Bedienung und Installation - Möglichst wenig Kenntnisse über Computer sollten benötigt werden, um diesen Messenger zu installieren und zu benutzen

Peer-to-Peer-Verbindung - Nur über eine direkte Verbindung miteinander kommunizieren

Anonymität - Kommunikation über das Tor-Netzwerk

Sicherheit - Verschlüsseltes Speichern von Nachrichten und Schlüsselpaaren, um Auslesen der Daten (z.B. durch totalitäre Staaten) zu verhindern

E2EE - Verschlüsselung der Nachrichten und Verifikation des Kommunikationspartners

Damit der Nutzer keine Konsole verwenden muss, um den Messenger zu benutzen, habe ich eine Desktopapplikation mit Rust und Tauri entwickelt. Durch Tauri sind keine weiteren Installationen von Bibliotheken nötig, wodurch eine einfache Installation gegeben ist. Zudem kann die Applikation für Windows, Linux und (theoretisch) MacOS kompiliert werden [vgl. Ru24]. Rust bietet eine hohe Performance und ist Memory Safe, wodurch viele Sicherheitslücken, wie Buffer Overflows, verhindert werden können [vgl. Ma23]. Das Tauri-Framework bietet hierbei zwischen User Interface (UI), welches ich in Typescript und React programmiert habe, und Rust-Backend eine Schnittstelle, um Daten (*payloads*) zwischen Frontend und Backend zu versenden [vgl. 23]. Ich werde in dieser Umsetzung nicht auf den Quellcode des Frontends eingehen, da dieser nur für das UI zuständig ist und keine weiteren Funktionalitäten bietet.

5.1 Peer-to-Peer-Verbindung

Damit Messenger in der Lage sind, sowohl als Client als auch als Server zu fungieren, startet jeder Messenger einen HTTP-Server, welcher einen HTTP-Endpunkt anbietet (in diesem Messenger */ws/*), mit welchem eine Websocket[3]-Verbindung durch Clients aufgebaut werden kann. Der Server ist hierbei also in der Lage sowohl Packets zu empfangen als auch zu senden.

[3] Ein Protokoll für bidirektionale Kommunikation zwischen Server und Client über HTTP [vgl. Mi23]

```
HttpServer::new(|| {
    return App::new()
        // Return the default message to tell other clients
        ↪  that this server is actually alive
        .service(hello)
        // The websocket endpoint
        .route("/ws/", web::get().to(ws_index));
})
// Bind just to localhost and run
.bind(("127.0.0.1", CONFIG.service_port()))?
.run()
.await?;
```

Analog dazu besitzt jeder Messenger auch einen Websocket-Client, welcher sich im direkten Weg (über das Tor-Netzwerk) mit den anderen Messengern verbinden kann. Der Verbindungsaufbau zu anderen Messengern sieht wie folgt aus:

```
/// `onion_hostname` is the onion address of the other
↪  messenger to connect to
pub async fn new(onion_hostname: &str) -> Result<Self> {
  // [...]
  let connect_host = onion_hostname.to_string();

  //[...]

  // The address which is used to connect to the websocket
  let onion_addr = format!("ws://{}.onion/ws/", connect_host);

  debug!("[CLIENT] Creating proxy...");

  // Creating the Socks5Proxy client which is used to connect
  ↪  to the tor network
  let proxy = SocksProxy::new()?;
  debug!("[CLIENT] Connecting Proxy...");
```

```
  let mut onion_addr = Url::parse(&onion_addr)?;
  onion_addr
      .set_scheme("ws")
      .or(Err(anyhow!("[CLIENT] Could not set scheme")))?;

  // Connecting to the destination host using the proxyy
  let sock = proxy.connect(&onion_addr).await?;

  //[...]
  // Connecting to the websocket with the client
  let (ws_stream, _) =
  ↪  tokio_tungstenite::client_async(&onion_addr, sock).await?;

  // [...]
}
```

Damit zwei Nutzer über den Messenger Nachrichten versenden können, muss ein Messenger die Rolle als Client und der ander als Server übernehmen. Dabei nimmt der Messenger, welcher die Verbindung zu einem anderen Messenger initiiert hat, die Rolle als Clients ein und der andere Messenger die Rolle als Server. Dadurch, dass jeder Messenger sowohl Client als auch Server sein kann, müssen empfangene Nachrichten von Client und Server zusammengeführt und an das Frontend übermittelt werden, welches in der Bibliothek *messaging* implementiert ist:

```
/// A Manager which holds the connections by receiver name
pub struct MessagingManager {
  /// The connections by receiver name
  pub(crate) connections: Arc<RwLock<HashMap<String, Connection>>>,
}

/// A generalized connection struct that can be used for both the client and the
#[derive(Debug, Clone)]
pub struct Connection {
    pub(super) info: Arc<RwLock<ConnInfo>>,
```

```
    read_thread: Arc<Option<ConnectionReadThread>>,
    pub(crate) self_verified: Arc<RwLock<bool>>,
    pub(crate) verified: Arc<RwLock<bool>>,
    pub(super) receiver_host: String,

    notifier_ready_tx: Sender<()>,
    notifier_ready_rx: Receiver<()>,
}
```

Der *MessagingManager* hält sowohl Verbindungen von Client zu Server und Server zu Client in einem generellen Konstrukt, der *Connection* fest. Diese *Connection*s werden in einer *HashMap* gespeichert. Der *MessagingManager* bietet eine Schnittstelle für das Frontend, um Nachrichten zu senden und zu empfangen. Wenn in die-

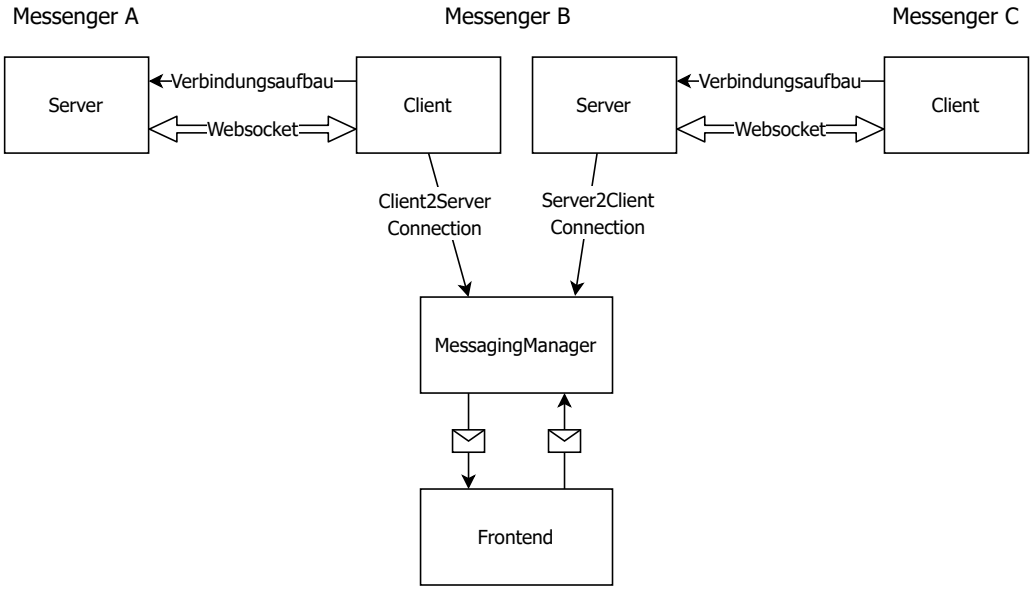

Abbildung 4: Mögliches Beispiel für Messenger B, welcher eine Verbindung zu Messenger A aufgebaut hat, und Messenger C, welcher mit Messenger B verbunden ist

sem Beispiel das Frontend eine Nachricht an Messenger A senden möchte, würde diese Nachricht zunächst an den *MessagingManager* geschickt, welcher diese verschlüsselte Nachricht an den dazugehörigen Client leitet. Der Client leitet nun die Nachricht über den Websocket an den Server des Messengers A weiter.

5.2 Tor-Netzwerk

Eine mögliche Lösung für das Kriterium der Anonymität besteht im Tor-Netzwerk. Eine Verbindung muss also zum Tor-Netzwerk aufgebaut werden, welches durch die *tor-proxy* Bibliothek geschieht. Der HTTP-Server wird auch durch den Tor-Proxy in das Tor-Netzwerk eingebunden. Damit der Tor-Proxy sich jedoch mit dem Tor-Netzwerk verbinden kann, wird eine Konfigurationsdatei benötigt, welche durch die *tor-proxy* Bibliothek erstellt wird:

```
/// Converts the configuration to a `torrc` file format
async fn to_text(&self) -> Result<String> {
    let data = PathBuf::from(self.data_dir());

    let geo_ip = data.clone().join("geoip");
    let geo_ip6 = data.clone().join("geoip6");

    #[allow(unused_mut)]
    let mut config = format!(
        "SocksPort {}
HiddenServiceDir \"{}\"
HiddenServicePort 80 {}
DataDirectory \"{}\"
GeoIPFile \"{}\"
GeoIPv6File \"{}\"",
        self.get_socks_host(),
        self.service_dir().to_string_lossy().replace("\\", "/"),
        self.get_hidden_service_host(),
        self.data_dir().to_string_lossy().replace("\\", "/"),
        geo_ip.to_string_lossy().replace("\\", "/"),
        geo_ip6.to_string_lossy().replace("\\", "/"),
    );

    //[...]
```

```
    Ok(config)
}
```

Eingehende Verbindung zum Onion Service dieses Messengers werden also nun zum Port des HTTP-Servers weitergeleitet. Nun muss der Tor-Proxy nur noch gestartet werden:

```
let mut child = Command::new(TOR_BINARY_PATH.clone());
child.args(["-f", &get_torrc().to_string_lossy()]);
child.current_dir(TOR_BINARY_PATH.parent().unwrap());
child.stdout(Stdio::piped());
child.stderr(Stdio::piped());
//[...]
let child = child.spawn()?;
```

Der Messenger ist jetzt mit dem Tor-Netzwerk verbunden und der HTTP-Server ist über den Onion Service erreichbar. Andere Messenger können sich nun anonym über das Tor-Netzwerk mit diesem Messenger verbinden.

5.3 Sicherheit

Ein Datenspeicher, welcher symmetrisch verschlüsselt ist, wird durch die Bibliothek *secure-storage* implementiert. Mit dieser Bibliothek lassen sich beliebige *Structs* in Text umwandeln und verschlüsseln bzw. auslesen und entschlüsseln. Die *storage-internal* Bibliothek benutzt hierbei die Methoden der *secure-storage* Bibliothek und dient als Wrapper, um die Daten auf der Festplatte zu speichern und wieder auszulesen. Der Datenspeicher enthält hierbei eine *HashMap*, welche als Schlüssel die Onion Adresse anderen Messengers und als Wert die Chatinformationen enthält. Die Chatinformationen sind zum Beispiel gesendete oder versendete Nachrichten, Schlüsselpaare und öffentliche Schlüssel.

```
#[derive(Clone, Debug, Serialize, Deserialize, Zeroize,
 ↪  ZeroizeOnDrop)]
pub struct StorageChat {
  /// All messages sent to this receiver or received from this
   ↪  receiver
```

```
    pub messages: Vec<ChatMessage>,
    //[...]

    /// The public key which is used to encrypt messages when
    ↪  being sent to the receiver [...]
    #[zeroize(skip)]
    pub rec_pub_key: Option<PublicKey>,
    /// Private key of this messenger used to decrypt the
    ↪  messages that are being received [...]
    pub priv_key: PrivateKey,
}
```

Der *StorageChat* enthält die Nachrichten, welche an den Empfänger gesendet oder von diesem empfangen wurden (*messages*), den öffentlichen Schlüssel des Kommunikationspartner (*rec_pub_key1*) und den privaten Schlüssel des Messengers *priv_key*. Um die Daten des Nutzers vor weiteren möglichen Angriffen zu schützen, verwende ich die Bibliothek *zeroize*, welche sensible Daten (wie private Schlüssel) aus dem Arbeitsspeicher löscht (Zeroisation) [vgl. 24]. Zudem ist es durch die symmetrische Verschlüsselung des Datenspeichers für Angreifer nicht möglich (sofern die Applikation geschlossen ist), private Schlüssel oder Chatverläufe auszulesen, wodurch die Privatsphäre der Nutzer geschützt wird.

5.4 Ende-zu-Ende-Verschlüsselung

Das letzte Kriterium, welches der Messenger nun erfüllen muss, ist die Ende-zu-Ende-Verschlüsselung. Dafür muss zuerst die Identität des anderen Kommunikationspartners verifiziert werden.

5.4.1 Identity verification

Um das Konzept der E2EE umzusetzen, muss der Sender einer Nachricht sicher sein, dass der Empfänger tatsächlich der ist, für den er sich ausgibt, wofür die *Identity verification* stattfindet. Im Rahmen der Erklärung gehen wir nun davon aus, dass Messenger A an Messenger B eine Nachricht senden möchte. Messenger A wäre in diesem Fall der Client und Messenger B der Server. Messenger A und Messenger B generieren anfangs (wenn noch nicht vorhanden) private Schlüssel und erstellen

ein neues *StorageChat*-Konstrukt, in welchem die privaten Schlüssel gespeichert werden. Damit sowohl Client als auch Server sicher sein können, dass der Kommunikationspartner tatsächlich der ist, für den er sich ausgibt, wird zunächst von dem Client ein *Identity* -Packet versendet. Dies besteht aus dem eigenen Hostname, der Signatur des Hostnames und dem Zielhostname (durch den privaten Schlüssel des Chats signiert) und den öffentlichen Schlüssel des Schlüsselpaares (vom Chat). Das Identitätspacket wird also folgendermaßen generiert:

```rust
async fn identity(receiver: &str) -> Result<Self> {
  // Get the own hostname
  let own_hostname = get_service_hostname(true)
  .await?
  .ok_or(anyhow!("Could not get own hostname"))?;

  // Get the private key for the receiver (used to decrypt messages)
  let priv_key = StorageManager::get_or_create_private_key(receiver).await?;
  let pub_key = priv_key.clone().try_into()?;

  // Creating a signature for the receiver with the hostname
  let keypair = PKey::from_rsa(priv_key.0)?;
  let mut signer = Signer::new(*DIGEST, &keypair)?;

  signer.update((own_hostname.clone() + receiver).as_bytes())?;
  let signature = signer.sign_to_vec()?;

  // Return the identity packet
  Ok(C2SPacket::SetIdentity(Identity {
    hostname: own_hostname,
    signature,
    pub_key
  }))
}
```

Ein wichtiges Detail ist hierbei die Signatur des *Identity* -Packets. Sie besteht sowohl

aus dem eigenen Hostname als auch dem Ziel des Hostnames, um zu verhindern, dass ein Angreifer das *Identity* -Packet abfängt und sich damit als Messenger des dazugehörigen Packets ausgibt. Nachdem der Server das Packet empfangen hat, überprüft er dieses. Dazu ruft dieser den öffentlichen Schlüssel des Kommunikationspartners auf, und verifiziert mit diesem die Signatur des *Identity* -Packets:

```
async fn verify(&self) -> Result<()> {
  let Identity {  hostname: remote_host, pub_key, signature} =
  ↪  self;
  // Get the own hostname
  let own_hostname =
  ↪  get_service_hostname(!remote_host.ends_with("-dev-client"))
      .await?
      .ok_or(anyhow!("Could not get own hostname"))?;



  debug!("Reading to verify...");
  // Check if there is a public key for the given receiver
  let local_pub_key = STORAGE.read().await.get_data(|e| {
      let key = e.chats.get(remote_host)
          .and_then(|e| e.rec_pub_key.clone());

      Ok(key)
  }).await?;

  debug!("Done");

  // If there is a public key, verify the signature
  if let Some(local_pub_key) = local_pub_key {
      info!("Verifying for hostname: {:?}", remote_host);
      let keypair = PKey::from_rsa(local_pub_key.0)?;
      let mut verifier = Verifier::new(*DIGEST, &keypair)?;
```

```
        // Verify the signature with the public key
        verifier.update((remote_host.to_string() +
        ↪  &own_hostname).as_bytes())?;
        let is_valid = verifier.verify(&signature)?;

        if !is_valid {
            warn!("[INVALID_SIGNATURE] Wrong signature was given!
            ↪  This may be an attack!");
            return Err(anyhow!("Wrong signature was given! This
            ↪  may be an attack!"));
        }

        Ok(())
    } else {
        // Adding public key to storage because it does  not
        ↪  exist
        info!("No chat with hostname '{}' yet. Adding new
        ↪  receiver...", remote_host);
        STORAGE.read().await.modify_storage_data(|e| {
            let res = e.chats.entry(remote_host.clone())
                .or_insert_with(||
                ↪  StorageChat::new(&remote_host));

            res.rec_pub_key = Some(pub_key.clone());

            Ok(())
        }).await?;
        debug!("Done.");

        Ok(())
    }
}
```

Falls die Identität valide ist, übersendet der Server seine Identität an den Client. Der gleiche Vorgang der Validation findet nun auch auf dem Client statt. Wenn dieser die *Identity* als valide betrachtet, ist eine sichere Verbindung zwischen Client und Server aufgebaut.

5.5 Nachrichten versenden

Gehen wir davon aus, dass der Client an den Server über die gesicherte Verbindung eine Nachricht versenden möchte. Dazu ruft er zunächst den öffentlichen Schlüssel des Empfängers aus dem Datenspeicher ab und verschlüsselt die Nachricht mit dem RSA-Verfahren. Anschließend übersendet er die verschlüsselte Nachricht an den Server.

```rust
/// Sends a message to the receiver with the given date and
↪  msg, internal function
async fn inner_send(&self, msg: &str, date: u128) -> Result<()>
↪  {
  let raw = msg.as_bytes().to_vec();

  let tmp = self.receiver_host.clone();
  debug!("Reading public key for {}...", tmp);

  // Firstly we need to get the public key of the receiver
  let pub_key = STORAGE
    .read()
    .await
    .get_data(|e| {
      e.chats
        .get(&tmp)
        .and_then(|e| e.rec_pub_key.clone())
        .ok_or(anyhow!("The pub key was empty (should never
        ↪  happen)"))
    })
    .await?;
```

```
    debug!("Sending");
    // And encrypt the message
    let bin = pub_key.encrypt(&raw)?;

    // And send it to the receiver, if we are the client, send a
    ↪  client packet if not, server packet
    match &*self.info.read().await {
      ConnInfo::Client(c) => {
        debug!("Client msg");
        let packet = C2SPacket::Message((date, bin));
        c.feed_packet(packet).await?;
      }
      ConnInfo::Server((_, s)) => {
        debug!("Server msg");
        let packet = S2CPacket::Message((date, bin));

        s.send(packet).await?;
      }
    };

    Ok(())
}
```

Der Server empfängt über die Websocket-Verbindung die verschlüsselte Nachricht des Clients und entschlüsselt diese anschließend mit dem privaten Schlüssel des Chats. Somit wurde die Nachricht empfangen, entschlüsselt und wird nun an das Frontend übermittelt, sodass diese angezeigt werden kann. Nutzer können nun sicher und anonym miteinander kommunizieren.

6 Nachteile und Lösungsmöglichkeiten

Der Messenger bietet zwar hohe Anonymität und Sicherheit, jedoch ist es unerlässlich auch die Nachteile des Messengers und deren Lösungsmöglichkeiten zu be-

trachten. Durch die Struktur des Tor-Netzwerkes können Datenmengen um das 120-fache langsamer übermittelt werden, welches bei simplen Textnachrichten nicht auffallen mag, bei größeren Datenmengen, wie zum Beispiel Bildern, jedoch ein Problem darstellen könnte [vgl. LSS10]. Dadurch, dass der Client eine Websocket-Verbindung mit dem Onion Service aufbaut, herrscht hier ein *long-term-circuit*, wodurch der *Circuit* zwischen Onion Service und Client nicht erneuert wird, somit für Angriffe anfälliger sein könnte [vgl. To24c]. Zusätzlich ist die Nachrichtenlänge durch das Padding der asymmetrischen Verschlüsselung begrenzt [vgl. Op24]. Eine hybride Verschlüsselung, also eine Kombination aus asymmetrischer und symmetrischer Verschlüsselung, könnte eine mögliche Lösung für die limitierte Nachrichtenlänge sein [vgl. HK07]. Dabei generiert bereits bei dem Verbindungsaufbau der Server einen zufälligen symmetrischen Schlüssel [vgl. ebd.]. Dieser wird mit dem öffentlichen Schlüssel des Clients verschlüsselt und an den Client übermittelt, welcher diesen entschlüsselt und abspeichert, sodass sowohl Server als auch Client den gleichen symmetrischen Schlüssel teilen [vgl. ebd.]. Somit werden die Vorteile der symmetrischen Verschlüsselung (schnell und keine Nachrichtenlängenbegrenzung) mit den Vorteilen der asymmetrischen Verschlüsselung (sicherer Austausch von Schlüsseln) kombiniert [vgl. 22]. Der Messenger muss zudem gestartet sein, damit der Onion Service im Tor-Netzwerk verfügbar ist, sodass ein Nutzer nur kontaktiert werden kann, wenn dieser auch den Messenger gestartet hat.

7 Fazit

In der heutigen Gesellschaft wird die Meinungsfreiheit der Bürger, welche in einem totalitären Staat leben, immer wichtiger. Das Tor-Netzwerk bietet hierbei eine Möglichkeit, anonym und sicher über Onion Services miteinander zu kommunizieren. Mit dem Messenger, welcher in dieser Arbeit umgesetzt wurde, ist es möglich, Menschen in totalitären Staaten ein Werkzeug zu geben, ihre Meinung zu äußern. Allerdings muss auch hierfür die Onion-Adresse der Person, welche den Messenger benutzt, sicher über eine andere Plattform übermittelt werden, damit zum Beispiel Reporter diese kontaktieren können. Zusätzlich müssen beide Kommunikationspartner deren Messenger gestartet haben.

Literatur

[Am23] Amnesty International. *Amnesty International Report 2022/23*. London WC1X 0DW, United Kingdom: International Amnesty Ltd, 2023, S. 307–312, 122–128, 196–201. ISBN: 978-0-86210-502-0. URL: https://www.amnesty.org/en/wp-content/uploads/2023/04/WEBPOL1056702023ENGLISH-2.pdf (besucht am 13. 01. 2024).

[Au18] Autumn. „How does Tor ∗really∗ work?“ In: (Feb. 2018). URL: https://hackernoon.com/how-does-tor-really-work-c3242844e11f (besucht am 23. 01. 2024).

[BSW15b] Albrecht Beutelspacher, Jörg Schwenk und Klaus-Dieter Wolfenstetter. „Ziele der Kryptographie“. In: *Moderne Verfahren der Kryptographie: Von RSA zu Zero-Knowledge*. Wiesbaden: Springer Fachmedien Wiesbaden, 2015, S. 1–7. ISBN: 978-3-8348-2322-9. DOI: 10.1007/978-3-8348-2322-9_1. URL: https://doi.org/10.1007/978-3-8348-2322-9_1.

[Bh23] Jasdeep Bhatia. *Understanding System Design Whatsapp & Architecture*. Mai 2023. URL: https://pwskills.com/blog/system-design-whatsapp/#Client-Server_Architecture (besucht am 22. 01. 2024).

[Bu24] Bundesamt für Sicherheit in der Informationstechnik. *DoS- und DDoS-Attacken*. Feb. 2024. URL: https://www.bsi.bund.de/DE/Themen/Verbraucherinnen-und-Verbraucher/Cyber-Sicherheitslage/Methoden-der-Cyber-Kriminalitaet/DoS-Denial-of-Service/dos-denial-of-service_node.html (besucht am 08. 02. 2024).

[Ch+11] Sambuddho Chakravarty, Georgios Portokalidis, Michalis Polychronakis und Angelos D. Keromytis. „Detecting Traffic Snooping in Tor Using Decoys“. In: *Recent Advances in Intrusion Detection*. Hrsg. von Robin Sommer, Davide Balzarotti und Gregor Maier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, S. 222–241. ISBN: 978-3-642-23644-0.

[DMS04] Roger Dingledine, Nick Mathewson und Paul Syverson. „Tor: The Second-Generation Onion Router“. In: *13th USENIX Security Symposium (USENIX Security 04)*. San Diego, CA: USENIX Association, Aug. 2004. URL: https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router.

[Du24] DuckDuckGo. *duckduckgo onion at DuckDuckGo*. Feb. 2024. URL: https://duckduckgo.com/?q=duckduckgo+onion&atb=v160-7&ia=web (besucht am 02. 02. 2024).

[El16] Elektronik Kompendium. *TCP/IP*. Nov. 2016. URL: https://www.elektronik-kompendium.de/sites/net/0606251.htm (besucht am 23. 01. 2024).

[23] *Events | Tauri Apps*. März 2023. URL: https://tauri.app/v1/guides/features/events (besucht am 15. 02. 2024).

[Fä23] Jan Fährmann. „Rechtliche Rahmenbedingungen der Nutzung von Positionsdaten durch die Polizei und deren mögliche Umsetzung in die Praxis–zwischen Strafverfolgung und Hilfe zur Wiedererlangung des Diebesguts“. In: *Private Positionsdaten und polizeiliche Aufklärung von Diebstählen*. Nomos Verlagsgesellschaft mbH & Co. KG. 2023, S. 141–176. ISBN: 978-3-8487-5905-7.

[Ge23a] GeeksforGeeks. *Comparison Centralized Decentralized and Distributed Systems*. Sep. 2023. URL: https://www.geeksforgeeks.org/comparison-centralized-decentralized-and-distributed-systems (besucht am 09. 02. 2024).

[Ge23b] GeeksforGeeks. *What are onion services in Tor Browser*. Okt. 2023. URL: https://www.geeksforgeeks.org/what-are-onion-services-in-tor-browser (besucht am 04. 02. 2024).

[Ge21] Gemini. *Networks: Decentralized, Distributed, & Centralized | Gemini*. Juli 2021. URL: https://www.gemini.com/cryptopedia/blockchain-network-decentralized-distributed-centralized#section-what-is-a-centralized-network (besucht am 08. 02. 2024).

[Gr14] Andy Greenberg. „Hacker Lexicon: What Is End-to-End Encryption?“ In: *WIRED* (Nov. 2014). URL: https://www.wired.com/2014/11/hacker-lexicon-end-to-end-encryption (besucht am 16. 01. 2024).

[HK20] Aljaafari Hamza und Basant Kumar. „A Review Paper on DES, AES, RSA Encryption Standards“. In: *2020 9th International Conference System Modeling and Advancement in Research Trends (SMART)*. 2020, S. 333–338. DOI: 10.1109/SMART50582.2020.9336800.

[HD05] Manfred Hauswirth und Schahram Dustdar. „Peer-to-peer: Grundlagen und Architektur“. In: *Datenbank-Spektrum* 13.2005 (2005), S. 5–13.

[HK07] Dennis Hofheinz und Eike Kiltz. „Secure Hybrid Encryption from Weakened Key Encapsulation“. In: *Advances in Cryptology - CRYPTO 2007*. Hrsg. von Alfred Menezes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, S. 553–571. ISBN: 978-3-540-74143-5.

[22] *Hybrid Encryption*. Jan. 2022. URL: https://www.techopedia.com/definition/1779/hybrid-encryption (besucht am 15. 02. 2024).

[IBM21] *IBM Documentation*. März 2021. URL: https://www.ibm.com/docs/en/ztpf/2020?topic=concepts-symmetric-cryptography (besucht am 23. 01. 2024).

[Is16] Ni Made Satvika Iswari. „Key generation algorithm design combination of RSA and ElGamal algorithm“. In: *2016 8th International Conference on Information Technology and Electrical Engineering (ICITEE)*. 2016, S. 1–5. DOI: 10.1109/ICITEED.2016.7863255.

[K 23] Arun K. L. „Detailed Anatomy of the Tor Network | Structure of the Tor Network“. In: *Sec Master* (Okt. 2023). URL: https://thesecmaster.com/detailed-anatomy-of-the-tor-network-structure-of-the-tor-network (besucht am 23. 01. 2024).

[Kr16a] N. Krzyworzeka. „Asymmetric cryptography and trapdoor one-way functions“. In: *Automatyka / Automatics* 20.2 (2016), S. 39–51. ISSN: 1429-3447. DOI: 10.7494/automat.2016.20.2.39.

[La22] Lakhwinder. *Understanding WhatsApp Architecture*. Aug. 2022. URL: https://hackernoon.com/understanding-whatsapp-architecture (besucht am 08. 02. 2024).

[Li23] Max (Chong) Li. „What A Decentralized Infrastructure Is And How It Actually Works“. In: *Forbes* (Mai 2023). URL: https://www.forbes.com/sites/digital-assets/2023/05/07/what-a-decentralized-infrastructure-is-and-how-it-actually-works (besucht am 08. 02. 2024).

[LSS10] Tomáš Liška, Tomáš Sochor und Hana Sochorová. „Comparison between normal and TOR-Anonymized Web Client Traffic“. In: *Procedia - Social and Behavioral Sciences* 9 (2010). World Conference on Learning, Teaching and Administration Papers, S. 542–546. ISSN: 1877-0428. DOI: https://doi.org/10.1016/j.sbspro.2010.12.194. URL: https://www.sciencedirect.com/science/article/pii/S1877042810022998.

[Lo+24] Daniela Lopes, Jin-Dong Dong, Pedro Medeiros, Daniel Castro, Diogo Barradas, Bernardo Portela, João Vinagre, Bernardo Ferreira, Nicolas Christin und Nuno Santos. „Flow Correlation Attacks on Tor Onion Service Sessions with Sliding Subset Sum“. In: *Network and Distributed System Security Symposium*. Internet Society, Feb. 2024. ISBN: 1-891562-93-2. URL: https://www.ndss-symposium.org/ndss-paper/flow-correlation-attacks-on-tor-onion-service-sessions-with-sliding-subset-sum/ (besucht am 08. 02. 2024).

[LB21] Ben Lutkevich und Madelyn Bacon. „end-to-end encryption (E2EE)“. In: *Security* (Juni 2021). URL: https://www.techtarget.com/searchsecurity/definition/end-to-end-encryption-E2EE (besucht am 16. 01. 2024).

[MS13] Prerna Mahajan und Abhishek Sachdeva. „A study of encryption algorithms AES, DES and RSA for security“. In: *Global Journal of Computer Science and Technology* 13.15 (2013), S. 15–22.

[Ma23] Giorgio Martinez. „Rust: Exploring Memory Safety and Performance - Giorgio Martinez - Medium“. In: *Medium* (Okt. 2023). ISSN: 9598-0120. URL: https://medium.com/@giorgio.martinez1926/unlocking-the-power-of-rust-exploring-memory-safety-and-performance-9afd5980c120 (besucht am 11. 02. 2024).

[Mi23] Microsoft. *WebSockets - UWP applications*. Juli 2023. URL: https://learn.microsoft.com/de-de/windows/uwp/networking/websockets (besucht am 13. 02. 2024).

[Op24] Openssl Foundation, Inc. */docs/man3.1/man3/RSA_public_encrypt.html*. Jan. 2024. URL: https://www.openssl.org/docs/man3.1/man3/RSA_public_encrypt.html (besucht am 04. 02. 2024).

[Pa+16] Ioana-Cristina Panait, Cristian Pop, Alexandru Sirbu, Adelina Vidovici und Emil Simion. „TOR - Didactic Pluggable Transport“. In: *Innovative Security Solutions for Information Technology and Communications*. Hrsg. von Ion Bica und Reza Reyhanitabar. Cham: Springer International Publishing, 2016, S. 225–239. ISBN: 978-3-319-47238-6.

[Re24] Reporter ohne Grenzen, e. V. *USA | Reporter ohne Grenzen für Informationsfreiheit*. Feb. 2024. URL: https://www.reporter-ohne-grenzen.de/usa (besucht am 08. 02. 2024).

[Ru24] Rust Foundation. *cargo build - The Cargo Book*. Feb. 2024. URL: https://doc.rust-lang.org/cargo/commands/cargo-build.html (besucht am 11. 02. 2024).

[Se19] Session. *Centralisation vs decentralisation in private messaging - Session Private Messenger*. Dez. 2019. URL: https://getsession.org/blog/centralisation-vs-decentralisation-in-private-messaging (besucht am 08. 02. 2024).

[Si23] Bundesamt für Sicherheit in der Informationstechnik. *BSI TR-02102-1 Kryptographische Verfahren: Empfehlungen und Schlüssellängen*. Bundesamt für Sicherheit in der Informationstechnik, Jan. 2023, S. 39–41. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-

TR-02102.pdf?__blob=publicationFile&v=9 (besucht am 21. 01. 2024).

[Si16] Signal Messenger. *Reflections: The ecosystem is moving*. Mai 2016. URL: https://signal.org/blog/the-ecosystem-is-moving (besucht am 08. 02. 2024).

[Ta96] D. Taipale. „Implementing the Rivest, Shamir, Adleman cryptographic algorithm on the Motorola 56300 family of digital signal processors". In: *Southcon/96 Conference Record*. Juni 1996, S. 10–17. DOI: 10.1109/SOUTHC.1996.535035.

[Ta13] Marius Tarnauceanu. *A generalization of the Euler's totient function*. 2013. DOI: 10.48550/arXiv.1312.1428. arXiv: 1312.1428 [math.GR]. URL: https://doi.org/10.48550/arXiv.1312.1428.

[To24a] Tor Project. *Deriving blinded keys and subcredentials [SUBCRED] - Tor Specifications*. Jan. 2024. URL: https://spec.torproject.org/rend-spec/deriving-keys.html (besucht am 04. 02. 2024).

[To24b] Tor Project. *directory authority | Tor Project | Support*. Feb. 2024. URL: https://support.torproject.org/glossary/directory-authority (besucht am 02. 02. 2024).

[To24c] Tor Project. *How can we help? | Tor Project | Support*. Feb. 2024. URL: https://support.torproject.org/about/change-paths/ (besucht am 04. 02. 2024).

[To24d] Tor Project. *Kanal | Tor Project | Hilfe*. Jan. 2024. URL: https://support.torproject.org/de/glossary/circuit (besucht am 31. 01. 2024).

[To24e] Tor Project. *src/core/or · main · The Tor Project / Core / Tor · GitLab*. Feb. 2024. URL: https://gitlab.torproject.org/tpo/core/tor/-/blob/main/src/feature/hs/hs_descriptor.c?ref_type=heads (besucht am 04. 02. 2024).

[To23a] Tor Project. *The introduction protocol [INTRO-PROTOCOL] - Tor Specifications*. Dez. 2023. URL: https://spec.torproject.org/rend-spec/introduction-protocol.html (besucht am 04. 02. 2024).

[To23b] Tor Project. *The rendezvous protocol - Tor Specifications*. Nov. 2023. URL: https://spec.torproject.org/rend-spec/rendezvous-protocol.html (besucht am 04. 02. 2024).

[To24f] Tor Project. *Tor Project | How do Onion Services work?* [Online; accessed 2. Feb. 2024]. Jan. 2024. URL: https://community.torproject.org/onion-services/overview.

[To24g] Tor Project. *Tor Project | Talk about onions*. Jan. 2024. URL: https://community.torproject.org/onion-services/talk (besucht am 31. 01. 2024).

[Tu08] Clay S Turner. „Euler's totient function and public key cryptography". In: *Nov* 7 (2008), S. 138.

[Un22] United Nations. *World Population Prospects - Population Division*. Jan. 2022. URL: https://population.un.org/wpp/Download/Files/1_Indicators%20(Standard)/EXCEL_FILES/1_General/WPP2022_GEN_F01_DEMOGRAPHIC_INDICATORS_COMPACT_REV1.xlsx (besucht am 13. 01. 2024).

[Wa+13] Hongjun Wang, Zhiwen Song, Xiaoyu Niu und Qun Ding. „Key generation research of RSA public cryptosystem and Matlab implement". In: *PROCEEDINGS OF 2013 International Conference on Sensor Network Security Technology and Privacy Communication System*. 2013, S. 125–129. DOI: 10.1109/SNS-PCS.2013.6553849.

[Wi+22b] Sandra Wittmer, Florian Platzer, Martin Steinebach und York Yannikos. „Deanonymisierung im Tor-Netzwerk – Technische Möglichkeiten und rechtliche Rahmenbedingungen". In: *Selbstbestimmung, Privatheit und Datenschutz : Gestaltungsoptionen für einen europäischen Weg*. Hrsg. von Michael Friedewald, Michael Kreutzer und Marit Hansen. Wiesbaden: Springer Fachmedien Wiesbaden, 2022, S. 151–169. ISBN: 978-3-658-33306-5. DOI: 10.1007/978-3-658-33306-5_8. URL: https://doi.org/10.1007/978-3-658-33306-5_8.

[Wu+23] Mingshi Wu, Jackson Sippe, Danesh Sivakumar, Jack Burg, Peter Anderson, Xiaokang Wang, Kevin Bock, Amir Houmansadr, Dave Levin und Eric Wustrow. „How the Great Firewall of China Detects and Blocks Fully Encrypted Traffic“. In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, S. 2653–2670. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/wu-mingshi (besucht am 14. 01. 2024).

[24] *zeroization - Glossary | CSRC*. Feb. 2024. URL: https://csrc.nist.gov/glossary/term/zeroization (besucht am 15. 02. 2024).

Anhang

[BSW15a] Albrecht Beutelspacher, Jörg Schwenk und Klaus-Dieter Wolfenstetter. „Kryptologische Grundlagen“. In: *Moderne Verfahren der Kryptographie: Von RSA zu Zero-Knowledge*. Wiesbaden: Springer Fachmedien Wiesbaden, 2015, S. 9–30. ISBN: 978-3-8348-2322-9. DOI: 10.1007/978-3-8348-2322-9_2. URL: https://doi.org/10.1007/978-3-8348-2322-9_2.

[Kr16b] N. Krzyworzeka. „Asymmetric cryptography and trapdoor one-way functions“. In: *Automatyka / Automatics* 20.2 (2016), S. 39–51. ISSN: 1429-3447. DOI: 10.7494/automat.2016.20.2.39.

[Wi+22a] Sandra Wittmer, Florian Platzer, Martin Steinebach und York Yannikos. „Deanonymisierung im Tor-Netzwerk – Technische Möglichkeiten und rechtliche Rahmenbedingungen“. In: *Selbstbestimmung, Privatheit und Datenschutz : Gestaltungsoptionen für einen europäischen Weg*. Hrsg. von Michael Friedewald, Michael Kreutzer und Marit Hansen. Wiesbaden: Springer Fachmedien Wiesbaden, 2022, S. 151–169. ISBN: 978-3-658-33306-5. DOI: 10.1007/978-3-658-33306-5_8. URL: https://doi.org/10.1007/978-3-658-33306-5_8.

Quellcode: https://github.com/sshcrack/enkrypton