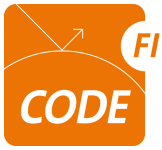
der Bundeswehr
Universität München

Masterarbeit

Varianten von blinden Signaturen und ihre Implementierung in CrypTool 2

Eingereicht von:	Axel Wehage 1150503
Aufgabensteller:	Professor Dr. Arno Wacker
Betreuer:	Professor Bernhard Esslinger Dr. Nils Kopal
Abgabedatum:	09. September 2019



Forschungsinstitut
Cyber Defence
Universität der Bundeswehr München

Eidesstattliche Erklärung

Hiermit versichere ich, Axel Wehage, dass ich die vorliegende Arbeit selbstständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, insbesondere keine anderen als die angegebenen Informationen.

Neubiberg, den 09.09.2019

Der Speicherung meiner Masterarbeit zum Zweck der Plagiatsprüfung stimme ich zu. Ich versichere, dass die elektronische Version mit der gedruckten Version inhaltlich übereinstimmt.

Neubiberg, den 09.09.2019

Inhaltsverzeichnis

Abkürzungsverzeichnis

ggf.	gegebenenfalls
bspw.	beispielsweise
bzw.	beziehungsweise
d.h.	das heißt
UniBw	Universität der Bundeswehr
CT	CrypTool
CT2	CrypTool 2
WPF	Windows Presentation Foundation

Danksagung

An dieser Stelle möchte ich mich bei Prof. Dr. Esslinger und Dr. Kopal bedanken, welche mich bei der Erstellung dieser Arbeit betreuten und trotz mehrerer Probleme mit sehr vielem und gutem Rat halfen und viel Geduld bewiesen.

Für die hilfreichen Anregungen, Lösungs- und Literaturvorschläge, ausführlichen Hilfestellungen und die konstruktive Kritik bei der Erstellung dieser Arbeit möchte ich mich in aller Form bedanken.

Ich habe viel gelernt dabei ...

Neubiberg, den 09.09.2019,

Axel Wehage

1 Einleitung

Blinde Signaturen – im Englischen blind signatures genannt – ermöglichen, dass eine digitale Signatur geleistet wird, ohne dass das zu unterzeichnende Dokument vom Signaturleistenden gelesen werden kann. Diese Art von Signatur und die dazugehörigen Verfahren bieten damit eine technisch sichergestellte Anonymität gegenüber allen, außer jenem, der die Signatur leistet.

Das entsprechende Forschungsgebiet der Kryptografie entstand 1982 mit der Arbeit von David Chaum [6]. Es wurde mehrfach theoretisch aufgearbeitet, allerdings konnte sich, aufgrund mangelnder Akzeptanz in der Bevölkerung, kein System praktisch langfristig durchsetzen. Derzeit werden blinde Signaturen weder in der europäischen eIDAS-Verordnung [24] noch in einem US-amerikanischen NIST-Standard definiert.

1.1 Motivation

Die steigende Vernetzung sowohl der Arbeitswelt als auch der Politik mit elektronischen Systemen resultiert darin, dass immer mehr Prozesse mit sensiblen Daten digital verarbeitet werden. Hierbei stellt sich jedoch das Problem, dass sowohl Alltagsprozesse als auch staatlich wichtige Vorgänge, die eine Anonymität für Einzelpersonen voraussetzen oder traditionell als gegeben sehen, nicht länger anonym ablaufen. Die Abwicklung von Geschäften über das Internet ist in der Mitte der Gesellschaft angekommen und alltäglich geworden. Hierbei kann jedoch ein Nutzer oftmals sehr leicht identifiziert und seine Handlungen können nachvollzogen werden. Derzeitige Umsetzungen von elektronischen Zahlungsmitteln wie EC und Kreditkarten bieten nur begrenzte Anonymität. Jeder Kauf kann nachverfolgt und zurückgeführt werden. Selbst bei fast komplett anonymen Kryptowährungen ist zum Einstieg ein Kauf dieser Währung nötig, welcher nachverfolgt werden kann. Schon allein daran, dass die Öffentlichkeit weiterhin Bargeld als beliebte Bezahlungsmethode verwendet, erkennt man jedoch, dass anscheinend eine Option, welche Anonymität bietet, ohne an Effizienz und Komfort einzubüßen, willkommen ist.

Deshalb muss nach Wegen gesucht werden, Werte wie die Anonymität, die die Öffentlichkeit nicht völlig aufgeben will, zu erhalten und auch in digitalen Systemen zu realisieren. Das Forschungsfeld der blinden Signaturen befasst sich damit, bei der elektronischen Kommunikation die Anonymität zu gewährleisten. Aus diesem Grund ist dieses Themengebiet trotz geringer realer Umsetzung der Konzepte von großer Bedeutung. Um die Thematik greifbarer und verständlicher zu machen, soll daher das Open-Source-Projekt CrypTool 2 (CT2) , welches sowohl für Forschung als auch für Lehre von kryptographischen und kryptoanalytischen Verfahren eingesetzt wird, entsprechend erweitert werden.

1.2 Aufgabenstellung

Die Aufgabenstellung der Arbeit lautet wie folgt:

Beschreiben und charakterisieren Sie die Varianten verschiedener blinder Signaturen. Implementieren und visualisieren Sie hierbei blinde Signaturen in CrypTool 2 (CT2). Die Arbeit unterteilt sich in einen praktischen Teil und einen theoretischen Teil. Der praktische Teil ist die Implementierung und visuelle Aufbereitung mit der Windows-Presentation-Foundation (WPF) in CT2. Als Vorlage können die bereits vorhandenen visuellen Darstellungen von Verfahren in CT2 dienen. Der theoretische Teil ist das Beschreiben und Charakterisieren von kryptographischen Signaturen im Allgemeinen.

Die Implementierung von blinden Signaturen ist auf mehreren Wegen möglich. Hierbei wird ein Verschlüsselungsverfahren so verändert, dass die Eigenschaften von blinden Signaturen, die in den Kapiteln 3.1 und 3.3 erläutert werden, erfüllt sind. Es sollen mindestens zwei Verfahren gewählt und blinde Signaturen für diese implementiert werden. Ziel dieser Arbeit ist daher, für das RSA- und für das Paillier-Verfahren eine Komponente zu entwickeln, die das Erstellen blinder Signaturen ermöglicht. In der Komponente soll man zwischen den Verfahren wählen können und eine gültige Signatur erhalten, ohne dass dem Signierenden der zu signierende Text bekannt gegeben wird. Des Weiteren soll mindestens eine Vorlage (Template) für zukünftige CT2-Nutzer erstellt werden, die die Nutzungsweise der blinden Signaturen als fertigen Prozess (Workflow) demonstriert.

Innerhalb der Arbeit soll die Idee der blinden Signatur sowie deren Konzept und Anwendbarkeit diskutiert werden. Alle in der Umsetzung getroffenen Entscheidungen werden in dieser Arbeit beschrieben. Die entwickelte Software ist in den Programmcode des CT2-Projektes einzupflegen.

1.3 Zielsetzung

In Absprache mit den Betreuern wurden gemäß Aufgabenstellung folgende Ziele definiert:

1. Die übergeordnete Idee hinter blinden Signaturen soll anschaulich und verständlich erklärt werden.

2. Der Ablauf eines blinden Signier-Prozesses soll anhand von Protokollen beschrieben werden.

3. Der Ablauf der Verifikation und der Umgang mit Betrugsversuchen soll anhand von Protokollen beschrieben werden.

4. Es müssen mindestens zwei verschiedene Algorithmen zur Auswahl in CT2 angeboten werden, die in einer Komponente das Erstellen blinder Signaturen unterstützen. In einer weiteren Komponente soll die entsprechende Verifikation ausgeführt werden.

2 Grundlagen

In diesem Kapitel werden die grundlegenden Begriffe, die für das Verständnis der restlichen Arbeit notwendig sind, eingeführt. Des Weiteren werden einige Varianten von Signaturverfahren genannt und ihr Verwendungszweck beschrieben.

2.1 Notationen und Begriffe

Im folgenden werden die im Verlauf der Arbeit verwendeten mathematischen Notationen und Begriffe aufgeführt:

M	ein Nachrichtentext
M*	der Hashwert des Nachrichtentext
A bzw. Alice	der Sender einer Nachricht
B bzw. Bob	der Empfänger einer Nachricht
I	Identität von A in Form einer Zahl (bspw. Hashwert)
C bzw. Charlie/die Bank	der Signaturleistende bei blinden Signaturen
S	die Signier-Transformation
V	die Verifikations-Transformation
s	die Signatur
h	eine Hashfunktion
k	ein Zufallswert
p,q	zwei verschiedene Primzahlen

2.2 Anforderungen an eine Signatur

Gemäß 3GPP [16], einer Kooperation von Standardisierungsgremien für den Mobilfunkbereich, muss eine digitale Signatur die folgenden Eigenschaften erfüllen:

Authentizität (Authenticity) Eine gültige Signatur muss implizieren, dass der Signierende willentlich die Nachricht signiert hat.

Unfälschbarkeit (Unforgeability) Nur der Signierende kann eine gültige Signatur für eine damit verbundene Nachricht erschaffen.

Unanfechtbarkeit (Non-repudiation) – Der Signierende kann nicht leugnen ein Dokument signiert zu haben, welches eine gültige Signatur besitzt.

Integrität (Integrity) – Die Signatur muss sicher stellen, dass der Inhalt der Nachricht nicht modifiziert wurde.

Darüber hinaus ergeben sich aus kryptographischer Sicht noch die beiden folgenden Anforderungen:

Korrektheit (Correctness) – Der Algorithmus zur Erstellung und Verifizierung einer Signatur muss zu jedem Zeitpunkt eine gültige Signatur erschaffen und verifizieren können. [21, S. 369]

Nicht-Wiederverwendbarkeit (Non-re-usability) – Eine für eine Nachricht erstellte Signatur kann nur für die ursprüngliche Nachricht verwendet werden und nicht bei weiteren, von dieser sich unterscheidenden Nachrichten wiederverwendet werden. [30, S. 2]

Des Weiteren ist laut der Bekanntmachung zur elektronischen Signatur nach dem Signaturgesetz und der Signaturverordnung durch die Bundesnetzagentur vom 18.02.2013 nötig, dass „es praktisch nicht möglich“ ist, „den Signaturschlüssel aus dem (öffentlichen) Signaturprüfschlüssel zu berechnen.“ [5, S. 4]

2.3 Eine digitale Signatur

Eine digitale Signatur ist das Ergebnis eines kryptographischen Verfahrens, um die Authentizität von digitalen Nachrichten oder Dokumenten zu beweisen. Es ist laut Punkt

10 der eIDAS-Verordnung vom 01. Juli 2016 definiert als „Daten in elektronischer Form, die anderen elektronischen Daten beigefügt oder logisch mit ihnen verbunden werden und die der Unterzeichner zum Unterzeichnen verwendet.“ [24, Seite L 257/84]

Eine Signatur wird mittels kryptographischer Verfahren erstellt. Für das Signieren einer Nachricht wird der private Schlüssel verwendet; ein dazugehöriger öffentlicher Schlüssel für das Verifizieren der Signatur. Der private Schlüssel definiert hierbei eine Menge von Signier-Transformationen $\mathbf{S_A}$ oder, wenn Zufall hinzukommt wie bei Paillier, die Menge $\mathbf{S_{A,k}}$**:** $\mathbf{k} \in R$. $\mathbf{V_A}$ bezeichnet die Verifikations-Transformationen, die den öffentlichen Schlüssel von A nutzen.

Die Benutzung einer Signatur zwischen zwei Personen kann mit dem folgenden, abstrakten Protokoll beschrieben werden (siehe Tabelle 2.1).

Schritt	Akteur	Operation
1	A	A wählt eine Zufallszahl k
2	A	A berechnet M* mit M* = h(M) Hierdurch hasht A die Nachricht M
3	A	A berechnet s mit $s = S_{A,k}(M^*)$ Hierdurch signiert A den Hashwert von M*
4	A	A sendet M und s an B
5	B	B berechnet M* mit M* = h(M) B berechnet hierdurch den Hash aus der erhaltenen Nachricht M
6	B	B verifiziert die erhaltene Signatur und prüft $V_A(s) = M^*$ B überprüft hiermit, ob der berechnete Hash mit dem signierten Text übereinstimmt
7	B	B akzeptiert die Signatur s, wenn $V_A(s) = M^*$ wahr ist

Tabelle 2.1: Abstraktes Protokoll zum Erstellen und Verifizieren einer Signatur [20, S. 434]

In dem in Tabelle 2.1 dargestellten Ablauf wählt **A** in Schritt 1 eine Zufallszahl **k** und berechnet in Schritt 2 den Hashwert **h(M)** der zu sendenden Nachricht. Eine Hashfunktion **h()** ist hierbei definiert als eine Funktion, die eine beliebig große Eingabemenge auf eine (kleinere) Zielmenge konstanter Länge abbildet. Dabei ist zu beachten, dass die Länge des Eingabetexts variieren kann. Es kann dazu kommen, dass zwei verschiedene Eingabetexte den gleichen Ausgabetext haben, dies wird als Kollision bezeichnet. Wie gut ein Hashverfahren ist, kann daran gemessen werden, wie schwer es ist, absichtlich Kollisionen herbeizuführen.

In Schritt 3 wird die Nachricht gehasht und dieser Hashwert wird unter Nutzung des privaten Schlüssels von **A** signiert. Hierbei wird die Signier-Transformation $\mathbf{S}_{A,k}$ **(M*)** (englisch: signing transformation) genutzt. Das Ergebnis dieser Operation ist die Signatur für den Nachrichtentext von **M**. Die erstellte Signatur wird an die Nachricht angehängt. Die Nachricht und die Signatur werden in Schritt 4 zusammen an **B** gesendet. **B** berechnet den Hashwert der Nachricht **M** in Schritt 5 und prüft die Signatur mittels **A**'s öffentlichem Schlüssel in Schritt 6. Hierbei wird die Verifikations-Transformation $\mathbf{V}_A$ **(M*,s)** genutzt. Die Verifikations-Transformation ist so konstruiert, dass sie ohne Wissen des privaten Schlüssels von **A** genutzt werden kann. Aus dieser Operation erhält **B** in Schritt 7 entweder das Ergebnis, dass **s** die zum Nachrichtentext **M** passende Signatur ist oder dass **s** keine gültige Signatur für **M** darstellt. Daraus kann **B** schließen, ob **M** oder **s** verändert oder gefälscht wurden beziehungsweise ob die Nachricht überhaupt von **A** stammt. [20, S. 434] Es gibt auch Signaturverfahren, bei denen die Nachricht direkt aus der Signatur wiederhergestellt wird, anstatt eine Signatur zusätzlich zur Nachricht anzuhängen. Dies wird als „message recovery" bezeichnet. Hierauf wird im Rahmen dieser Arbeit jedoch nicht weiter eingegangen. [21, S. 35 f.]

2.4 Das RSA-Kryptosystem

Das RSA-Kryptosystem ist ein asymmetrisches Kryptosystem, das von Ronald L. Rivest, Adi Shamir und Leonard M. Adleman 1977 vorgeschlagen wurde. Das Verfahren macht sich trapdoor (Falltür) Funktionalität zunutze. Dies bedeutet, dass sich die Inverse des Algorithmus mit Hilfe einer geheimen Information sehr leicht berechnen lässt, während es ohne diese geheime Funktion sehr schwer ist, die Umkehrfunktion zu finden. [11, S. 13]

Bei RSA werden mit zwei großen, zufällig gewählten Primzahlen ein öffentlicher und ein privater Schlüssel generiert (anschließend werden die Primzahlen verworfen). Im öffentlichen Schlüssel ist das Produkt **N** der beiden Primzahlen enthalten. Die Sicherheit des Verfahrens besteht darin, dass die Primfaktorzerlegung von **N** besonders zeitaufwändig ist. Es ist nicht unmöglich, diese Berechnung durchzuführen: Die längste bisher faktorisierte RSA-Zahl besaß eine Länge von 768 bits [19]. Nach dem sogenannten Algorithmenkatalog des BSI gelten derzeit RSA-Schlüssel mit einer Mindestlänge von 2000 Bit als sicher. [25, S. 38]

Das RSA-Verfahren bietet sowohl die Möglichkeit, Nachrichten asymmetrisch zu verschlüsseln als auch als digitales Signaturverfahren eingesetzt zu werden. Die Funktion hierbei hängt von der Nutzungsweise des gewählten Schlüssels ab.

Beim Verschlüsseln einer Klartextnachricht nutzt der Sender den öffentlichen Schlüssel des Empfängers. Die Nachricht kann nur mit dem privaten Schlüssel des Empfängers entschlüsselt werden. Fließen in die RSA-Operationen der private Schlüssel und die Klartextnachricht ein, so kann man die Nachricht mit dem öffentlichen Schlüssel wiederherstellen. Diese Nutzungsweise stellt ein digitales Signaturverfahren dar. [21, S. 336 f.]

Mathematisch sieht das RSA-Verfahren so aus:

1. Schlüsselgenerierung: Zur Generierung der Schlüssel werden zwei zufällige Primzahlen p und q von annähernd gleicher Größe gewählt und der Modulus n = p*q berechnet. Des Weiteren wird ein Integer **e** mit **1 < e <** $\varphi(n)$ mit $\varphi(n)$ **= (p-1)*(q-1)** und **e** teilerfremd zu $\varphi(n)$ berechnet. Anschließend wird ein zu **e** inverses Integer **d** mit **1 < d <** $\varphi(n)$ und **de** $\equiv$ **1 (mod** $\varphi(n)$**)** berechnet.

Das Ergebnis dieser Berechnungen sind der öffentliche Schlüssel **(n,e)** und der dazugehörige private Schlüssel **(n,d)**.

2. Verschlüsselung: Für die Verschlüsselung gilt $\mathbf{c = M^e \bmod n}$ mit **c** als verschlüsselte Nachricht; für die Entschlüsselung gilt: $\mathbf{M = c^d \bmod n}$.

3. Signieren: Das Signieren bei RSA erfolgt, indem folgende Operation auf einen Text mit dem privaten Schlüssel angewandt wird: $\mathbf{s = M^d \bmod n}$. Die so erstellte Signatur kann mittels des öffentlichen Schlüssels durch $\mathbf{M = s^e \bmod n}$ geprüft werden. Meist wird hierbei nicht die Nachricht selbst signiert, sondern ein Hash davon.

Das Signieren einer Nachricht durch $\mathbf{s = M^d \bmod n}$ entspricht hierbei der Signatur-Transformation wie in Schritt 3 von Tabelle 2.1 verwendet. Hingegen entspricht die Prüfung der Signatur durch $\mathbf{M = s^e \bmod n}$ der Verifikations-Transformation wie in Schritt 6 von Tabelle 2.1 verwendet.

2.5 Das Paillier-Kryptosystem

Das Paillier-Kryptosystem wurde 1999 von Pascal Paillier erfunden. Hier liegt ebenfalls ein asymmetrisches Kryptosystem vor. Die Funktionen des Verfahrens lassen sich ebenfalls als „trapdoor functions“ klassifizieren und nutzen Primzahlen zur Schlüsselerzeugung.

Die Formeln zur Erzeugung der Schlüssel sowie zur Ver- und Entschlüsselung sind in Abb. 2.1 zu sehen.

Schlüsselgenerierung	Große Primzahlen p und q selber Länge $(p \neq q) : n = pq$, Residuenbasis $g \in_R \mathbb{Z}^*_{n^2}$ mit $ggT(L(g^\lambda \bmod n^2), n) = 1$ Öffentlicher Schlüssel: (n, g), Privater Schlüssel: (p, q) bzw. $\lambda = kgV(p-1, q-1)$
Verschlüsselung	Klartext: $m \in \mathbb{Z}_{n^2}$, $m_1 = m \bmod n$ und $m_2 = m \; div \; n$, Chiffretext: $c = g^{m_1} m_2^n \bmod n^2$
Entschlüsselung	Schritt1: $m_1 = \frac{L(c^\lambda \bmod n^2)}{L(g^\lambda \bmod n^2)} \bmod n$ Schritt 2: $z = cg^{-m_1} \bmod n$, Schritt 3: $m_2 = z^{1/n \bmod \lambda} mod \; n$, Schritt 4: Klartext $m = m_2 n + m_1$

Abbildung 2.1: Schlüsselgenerierung, Verschlüsselung und Entschlüsselung beim Paillier-Verfahren [4, S. 5]

Die Formeln für den Vorgang des Signierens und Verifizierens einer Nachricht mit dem Paillier-Verfahren sind in Abb. 2.2 zu sehen.

Signaturalgorithmus	Klartext: $m \in \mathbb{Z}^*_{n^2}$, $s_1 = \frac{L(m^\lambda \bmod n^2)}{L(g^\lambda \bmod n^2)} \bmod n$, $s_2 = (mg^{-s_1})^{1/n \bmod \lambda} \bmod n$, Signatur: $\sigma(m) = (s_1, s_2)$
Verifikationsalgorithmus	$m \stackrel{?}{=} g^{s_1} s_2^n \bmod n^2$
Korrektheit	$g^{s_1} s_2^n = g^{s_1} m g^{-s_1} = g^{s_1} g^{-s_1} m = m$

Abbildung 2.2: Signatur- und Verifikationsalgorithmus beim Paillier-Verfahren [4, S. 7]

Die Funktion **L** ist definiert als **L(x) = (x-1) / n**.

Ein einfaches Zahlenbeispiel für das Paillier-Kryptosystem:

a) Schlüsselgenerierung: **p = 23**; **q = 29**, wodurch sich der Modulus **n = 667** ergibt. Mit dem zufällig gewählten, öffentlichen Schlüssel **g = 668 < n^2 = 444889** ergibt sich der private Schlüssel **lambda = 308**.

b) Signieren: Für eine Nachricht $\mathbf{M = 123}$ ergibt sich die Signatur als Tupel ($\mathbf{s_1 = 17}$, $\mathbf{s_2 = 487}$). Dazu wurden als Zwischenwerte $\mathbf{M^{lambda} = 378190}$, $\mathbf{g^{lambda} = 205437}$ und $\mathbf{g^{-s_1} = 1}$ berechnet.

c) Verifizieren: Mit $\mathbf{g^{s_1} = 11340}$ und $\mathbf{s_2{}^n = 384982}$ ergibt sich $\mathbf{g^{s_1} * s_2{}^n \bmod n^2 = 123}$. Dieses Ergebnis stimmt überein mit der Eingabe $\mathbf{M = 123}$, daher ist die Signatur korrekt.

Die semantische Sicherheit des Paillier-Verfahrens gegen Angriffe mit gewähltem Klartext kann gezeigt werden. Dies geschieht unter der Annahme, dass für einen zusammengesetzten Modul $\mathbf{n}$ nicht effizient geprüft werden kann, ob ein Element in $\mathbf{(}\mathbb{Z}/\mathbf{n^2}\ \mathbb{Z}\mathbf{)}$ eine $\mathbf{n}$-te Wurzel modulo $\mathbf{n^2}$ besitzt oder nicht. Diese Annahme wird als „decisional composite rediuosity“ bezeichnet. [22, S. 3]

2.6 CrypTool 2[1]

Das Projekt „CrypTool“ (CT) wurde 1998 ins Leben gerufen. Seit 2003 ist es ein Open-Source-Projekt. Die maßgeblichen Resultate des Projekts sind die E-Learning-Programme CrypTool 1 mit den Nachfolgern CrypTool 2 (CT2) und JCrypTool (ab 2007/2008). Inzwischen zählen sie zu den weltweit am weitesten verbreiteten Lern-Programmen im Bereich Kryptografie und Kryptoanalyse und finden Anwendung sowohl in Lehre als auch in Aus- und Fortbildung. [27]

CT2 ist eine Windows-Anwendung und wird in der Programmiersprache C# [26] entwickelt. CT2 basiert auf dem derzeit aktuellen .NET-Framework 4.7.2 und der Windows Presentation Foundation WPF [14], die eine Vektorgrafik-basierte Oberfläche erlaubt. Die Architektur der Software ist modular, so dass sie ohne großen Aufwand um neue Funktionen ergänzt werden kann. [27] Die Programme werden pro Monat rund 10.000 mal von der CT-Webseite herunter geladen.

Die Oberflächenelemente *Startcenter*, *Arbeitsbereich* und *Komponenten* werden vom Benutzer am häufigsten verwendet. Im Folgenden wird ihre Bedienung kurz beschrieben.

2.6.1 Das Startcenter

Beim Starten von CT2 öffnet sich das Startcenter (Abb. 2.3). In diesem gibt es vier Bereiche: *Hauptfunktionen*, *Vorlagen*, *Neues* und *Zuletzt geöffnete Vorlagen*:

- Über die *Hauptfunktionen* kann man bspw. mit dem Wizard das Programm erkunden, einen neuen Arbeitsbereich (Workspace) mit dem eingebauten grafischen Editor erstellen, das CrypTool-Buch oder die Online-Dokumentation lesen sowie die offizielle Webseite oder die Facebook-Seite zu CrypTool 2 besuchen.

- Im Bereich *Neues* sieht man die aktuellsten Änderungen am Projekt.

- Eine Vorlage (Template) ist ein grafisches Programm, das ein ganzes Szenario enthält. Dazu nutzt es eine oder mehrere Komponenten, die miteinander verbunden sind. Vorlagen werden als Ganzes in den Arbeitsbereich geladen. Alle verfügbaren Vorlagen findet man über die eingebaute Vorlagen-Suchfunktion oben im Bereich *Vorlagen*.

[1] Dieser Abschnitt wurde größtenteils von [12] übernommen.

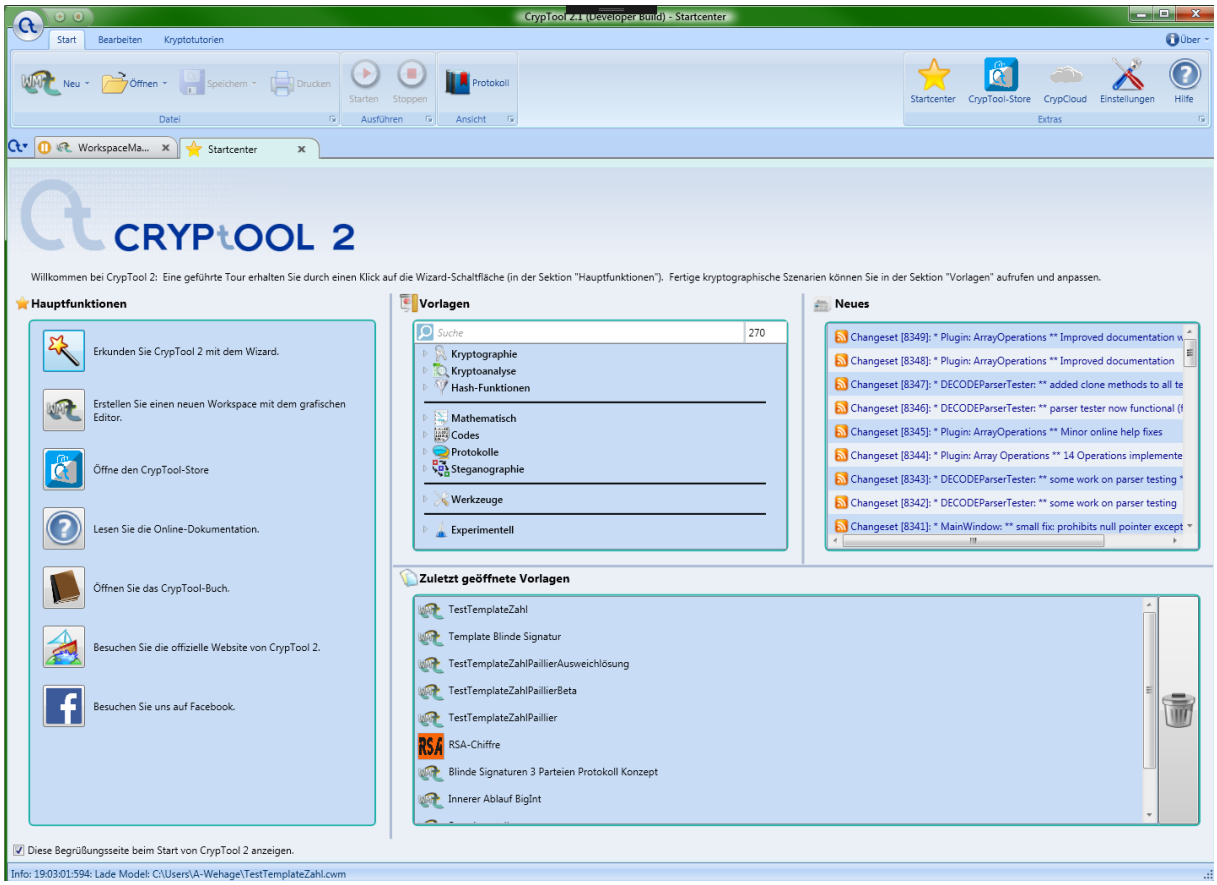


Abbildung 2.3: CT2-Startcenter

- Bereits vom Benutzer geöffnete Vorlagen werden in eine Liste unter *Zuletzt geöffnete Vorlagen* geschrieben.

2.6.2 Der Arbeitsbereich

Wenn man eine Vorlage im Arbeitsbereich öffnet, zeigt CT2 **sechs** Bereiche, wie in Abb. 2.4 dargestellt:

1. Im Bereich mit der Nummer 1 (oben, rot umrahmt) ist die Titelleiste, das Hauptmenü und die Ribbonbar zu sehen. Hier kann man unter anderem die geöffneten Programme starten oder stoppen.

2. Im zweiten Bereich (gelb umrahmt) sind alle verfügbaren Komponenten zu sehen, eingeteilt in Kategorien (Klassische Verfahren, Moderne Verfahren, Stegano-

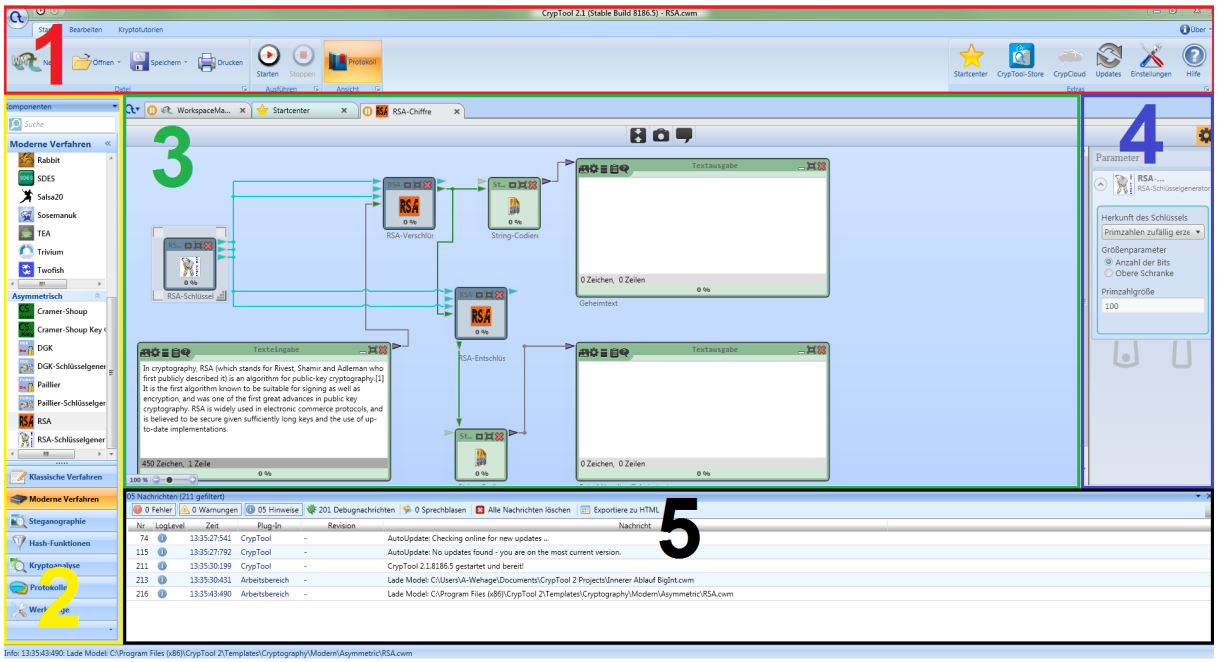


Abbildung 2.4: CT2-Arbeitsbereich mit geöffneter RSA-Chiffre-Vorlage

grafie, Hash-Funktionen, usw.). Zusätzlich können diese über die Komponenten-Suchfunktion oben schnell gefunden werden. Die in dieser Masterarbeit neu zu implementierenden Komponenten werden in die Kategorie *Moderne Verfahren – Asymmetrisch* eingeordnet.

3. Der dritte Bereich (grün umrahmt) ist die Arbeitsfläche, in der fertige Vorlagen geladen und ausgeführt, oder eigene Workflows und Vorlagen erstellt werden können. Die Komponenten auf der Arbeitsfläche können über ihre Ein- und Ausgänge miteinander verbunden werden. Hier ist exemplarisch die RSA-Komponente zu sehen, die eine Nachricht als Texteingabe annimmt und verschlüsselt bzw. entschlüsselt (unter zu Hilfenahme folgender Parameter: des öffentlichen Schlüssels, eines Modulus und eines privaten Schlüssels).

4. Rechts daneben (blau umrahmt) liegt der vierte Bereich, in dem die Parameter angezeigt werden, die sich bei der selektierten Komponente einstellen lassen. Parameter können nur verändert werden, wenn das Programm gestoppt ist. Die Parameterleiste einer Komponente wird mit Strg+I oder mit einem Klick auf das Zahnrad rechts oben geöffnet.

5. Im fünften Bereich (unten, schwarz umrahmt) sieht man die Nachrichtenkonsole (Log): In dieser werden alle Fehler, Warnungen Hinweise oder sonstige Nachrich-

ten angezeigt, die während der Nutzung von CT2 entstehen.

6. Ganz unten ist die Statuszeile.

Mit F11 werden der gelbe und der schwarze Bereich (Nr. 2 und Nr. 5) ausgeblendet. Mit F12 werden die Ribbonbar mit ihren großen Ikonen (im roten Bereich, Nr. 1) und die Statuszeile ganz unten ausgeblendet. Beide Funktionstasten arbeiten alternierend. Damit kann man schnell mehr Platz für den Arbeitsbereich (Bereich Nr. 3) schaffen.

2.6.3 Komponenten

Entwickler, die Erweiterungen für CT2 schreiben, bauen ein Plugin. Ein Plugin ist eine .NET-Assembly, die eine oder mehrere Komponenten beinhaltet. [27]

Eine Komponente ist eine Funktion, die auf dem Arbeitsbereich liegt und nach dem Starten (Drücken des Starten-Buttons; im Englischen Play-Button) berechnet und ausgeführt wird. Im minimierten Zustand wird eine Komponente als Icon mit Ein- und Ausgängen dargestellt. Abb. 2.5 zeigt eine typische Komponente im maximierten (aufgeklappten) Zustand, bei der im Präsentationsbereich die Einstellungen angezeigt werden.

Die Komponente in Abb. 2.5 wurde in vier Bereiche unterteilt:

- Im oberen, rot umrahmten Bereich sind links die drei Ikonen, mit denen man zwischen den Ansichten, die im Präsentationsbereich der Komponente dargestellt werden, wechseln kann. Mit der 4. Ikone kann man die zugehörige Onlinehilfe aufrufen. Mittig steht der Name der Komponente und rechts hat man die Möglichkeit, das Fenster zu minimieren, zu maximieren oder zu schließen.

- Im grün umrahmten Bereich (Präsentationsbereich) wird die eingestellte Ansicht angezeigt, in diesem Fall die Einstellungen. Parameter der Einstellungen können auch über die Eingänge einer Komponente gesetzt werden. Normalerweise hat eine Komponente Default-Werte für ihre Parameter. Gibt es einen Eingang dazu, überschreibt er den Defaultwert. Während der Ausführung einer Komponente können die Parameterwerte innerhalb der Komponente nicht geändert werden; sind sie jedoch mit Eingängen verbunden, können Sie auch während der Ausführung geändert werden.

- Der blaue Bereich beinhaltet eine Fortschrittsanzeige.

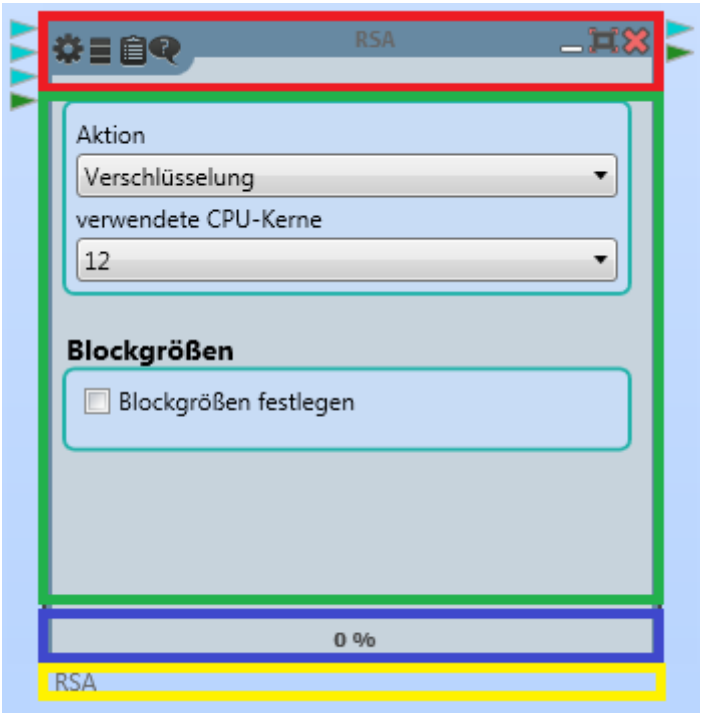


Abbildung 2.5: RSA-Komponente mit Einstellungen

- Unterhalb der Komponente, im gelben Bereich, steht eine kurze Beschreibung, die man frei wählen kann.

Über die ein- und ausgehenden Konnektoren (Dreiecke), links und rechts des roten Bereichs, kann die Komponente mit anderen Komponenten verbunden werden. Wie viele Ein- und Ausgänge eine Komponente hat, wird vom Programmierer festgelegt. Konnektoren können mit der Flag *mandatory* versehen werden, womit vom Programmierer festgelegt wird, dass ein Konnektor verbunden sein muss. Ohne diese Flag ist der Default, dass ein Konnektor nicht verbunden sein muss.

2.6.4 Der Lebenszyklus einer CT2-Komponente

Der Lebenszyklus einer Komponente (siehe Abb. 2.6) beginnt damit, dass sie – beim Laden in den Workspace – initialisiert wird. Dabei wird die Methode *Initialize()* aufgerufen. In dieser Methode können initiale Einstellungen vorgenommen werden, zum Beispiel das Laden einer Ansicht für eine Präsentation. Sobald man die Komponente mit einem Klick auf *Starten* (im Englischen *Play*) ausführt, startet CT2 alle Komponenten auf der Arbeitsfläche. Dabei wird als erstes bei allen Komponenten auf der Arbeitsfläche die

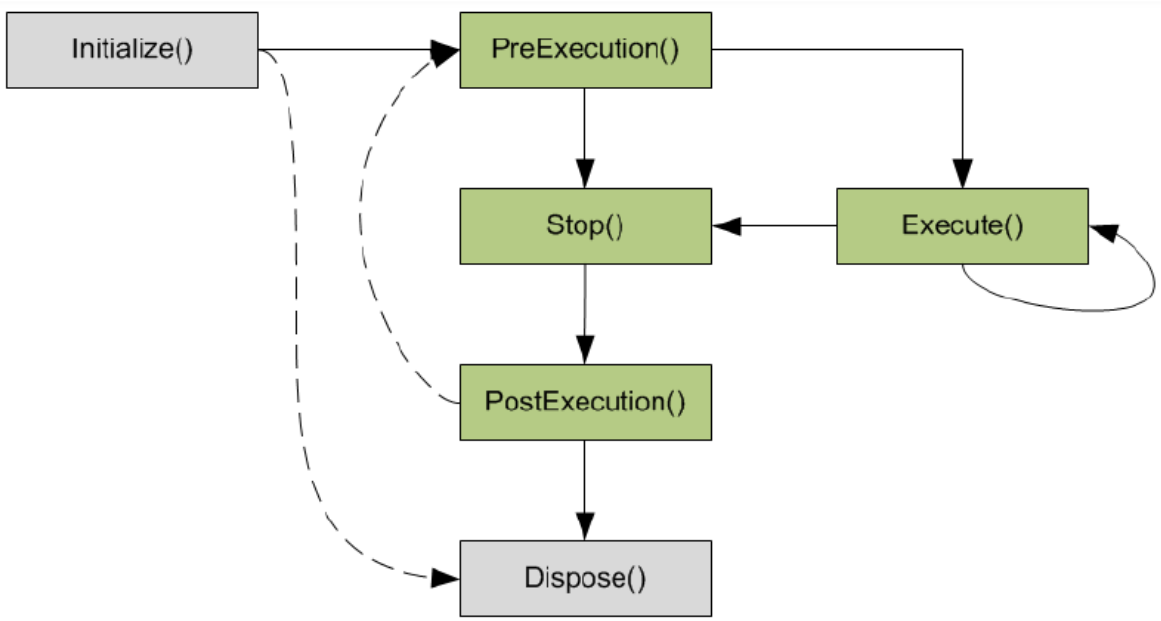


Abbildung 2.6: Lebenszyklus einer Komponente [1]

Methode *PreExecution()* ausgeführt. In dieser Methode können gewisse Vorarbeiten geleistet werden, wie das Aufbauen einer Verbindung zu einem Server oder das Prüfen bestimmter Einstellungen.

Anschließend wird die Methode *Execute()* ausgeführt. Hier finden die eigentlichen Berechnungen der Komponente statt. Diese wird immer wieder neu aufgerufen, sobald sich einer der Eingänge verändert.

Mit Betätigung der Schaltfläche *Stoppen* wird die laufende Ausführung der Arbeitsfläche beendet. Dadurch wird in jeder Komponente auf der Arbeitsfläche die Methode *Stop()* ausgeführt, welche interne Berechnungen unterbricht. CT2 beendet alle laufenden Threads und ruft abschließend die Methode *PostExecution()* auf. Diese Methode ist das Äquivalent zu *PreExecution()*, hier kann die Komponente in ihren Ursprungszustand zurück versetzt werden.

Entfernt man eine Komponente oder schließt die Arbeitsfläche, wird die Methode *Dispose()* ausgeführt, in der noch verwendete Ressourcen freigegeben werden können. Damit ist der Lebenszyklus einer Komponente beendet. Der Prozess startet von Neuem, sobald eine Komponente neu auf die Arbeitsfläche gezogen wird.

2.6.5 CT2 als Programm

Das CT2-Projekt wird derzeit unterteilt in einen Nightly Build (NB) mit aktuellen Änderungen und einen Stable Build (SB) mit halbjährlich freigegebenen und geprüften Änderungen. Die SB-Version ist hierbei die offizielle Releaseversion. Beide Varianten sind als ZIP-Datei und als Setup-Executable verfügbar und beide verfügen über einen automatischen Update-Mechanismus. Die aktuelle Releaseversion ist als Setup-Datei [CrypTool 2.1 (Stable Build 8186.5) vom 25.6.2019] knapp 150 MB groß und enthält rund 220 Templates und ca. 180 verschiedene Komponenten.

Für diese Arbeit sind nur die grundlegenden Funktionen von CT2 von Bedeutung. Diese sind unter anderem die Online-Hilfe, Templates, Auswählen und Nutzen von Plugins mit ihren Einstellungen, Starten und Stoppen von Workspaces sowie die Präsentationsmodi von Komponenten. Auf Bestandteile wie den CT-Store (zum Nachladen weiterer Plugins oder großer Statistik-Dateien) oder auf die CrypCloud (zum verteilten Rechnen in Ad-hoc-Netzen) wird nicht weiter eingegangen.

3 Blinde Signaturen

In diesem Kapitel werden blinde Signaturen genauer erläutert. Ihr Verwendungszweck sowie Vorteile und Nachteile werden beschrieben. Die existierenden Arten von blinden Signaturen werden erklärt und gegen weitere Signaturarten abgegrenzt. Abschließend wird eine Klassifikation der verschiedenen Varianten blinder Signaturen vorgestellt. Die in diesem Kapitel verwendeten Notationen für Akteure in Protokollen stimmen überein mit den in Kapitel 2.1 eingeführten Notationen.

3.1 Idee nach David Chaum

Der Begriff „blinde Signaturen“ beschreibt eine spezielle Art von kryptographischen Signaturen. Diese Art von Signaturen wurde erstmals von David Chaum im Jahr 1982 vorgeschlagen. [6] Anlass des Vorschlags war die steigende Verwendung elektronischer Zahlungssysteme, welche bislang keinerlei Möglichkeit boten, anonym Transaktionen durchzuführen. Die vorgeschlagene Signatur sollte automatisierte Zahlungen ermöglichen und folgende Eigenschaften erfüllen:

1. Dritten soll es unmöglich sein, den Zahlungsempfänger, Zeit und Summe von Zahlungen eines Individuums zu bestimmen. [6]

2. Individuen sollen fähig sein, einen Beweis der Zahlung vorzuweisen oder unter besonderen Umständen die Identität des [...][Zahlungsleisters] zu bestimmen. [6]

3. Es soll möglich sein, die Verwendung von als gestohlen gemeldeten Zahlungsmitteln aufzuhalten. [6]

Nach Chaum soll ein Kryptosystem für blinde Signaturen drei Funktionen enthalten (im Originaltext wird als Nachricht ein zufälliger Wert **x** verwendet, in dieser Arbeit wird dies durch einen Nachrichtentext **m** ersetzt):

1. Eine Signierfunktion **S**, die nur dem Signaturleistenden C bekannt ist und eine öffentlich bekannte inverse Funktion besitzt. [6]

2. Eine kommutative Funktion **d** mit einer zugehörigen inversen Funktion **d'**, welche nur dem Sender bekannt sind. Es muss gelten, dass **d'(S(d(M)))=S(M)** und **d(M)** sowie **S** keinen Hinweis über M geben. [6]

3. Ein Prädikat **r**, welches auf ausreichende Redundanz prüft und ein Suchen nach gültigen Signaturen unbrauchbar macht. [6] Ein Prädikat beschreibt hierbei eine Funktion, die für einen Eingabewert entweder true oder false zurückgibt.

Die in Punkt 2 genannte Funktion **d(M)** wird in späteren Anwendungsfällen zumeist in Form einer Multiplikation mit einem zufälligen Faktor **k** realisiert. Dieser Faktor wird Blendfaktor genannt, während die kommutative Funktion als Blendoperation bzw. Blendfunktion bezeichnet wird.

Nach dem von Chaum vorgeschlagenen Protokoll wird eine blinde Signatur gemäß den Schritten in Tabelle 3.1 erstellt. Unter der Spalte „Akteure“ steht derjenige, der den jeweiligen Schritt durchführt und damit agiert.

Schritt	Akteur	Operation
1	A	A wählt M, sodass r(M) = true gilt
2	A	A berechnet d(M). A blendet dadurch M
3	A	Sendet d(M) an C. A sendet nur die geblendete Nachricht d(M) an C, die ungeblendete Nachricht M erhält C nicht
4	C	C signiert d(M) mit S durch $S_C(d(M))$. C signiert hierbei eine geblendete Nachricht und kann deren Inhalt daher nicht lesen
5	C	C sendet $S_C(d(M))$ an A
6	A	A wendet d' an mit $d'(S_C(d(M)))=S_C(M)$ wodurch d(M) mit d' invertiert wird.
7	A	A erhält eine gültige Signatur S_C für M, da sie durch die Invertierung von d(M) entblendet wurde.

Tabelle 3.1: Abstraktes Protokoll zur Erstellung einer blinden Signatur nach Chaum

Nach Durchführung dieser Schritte erhält **A** also ein $\mathbf{S_C(m)}$, welches eine gültige Signatur für **M** ist. Für diese Signatur kann nun öffentlich geprüft werden, dass sie von

C geleistet wurde, indem $\mathbf{V_C(S_C(M))}$ sowie $\mathbf{r(V_C(S_C(M)))}$ angewandt wird und die Ergebnisse auf Gleichheit geprüft werden (Validierung).

Chaum hat für ein nicht-nachverfolgbares Zahlungssystem unter Nutzung blinder Signaturen die in Tabelle 3.2 beschriebenen Schritte vorgestellt. Für das Szenario, dass eine Person einen digitalen Geldschein von einer Bank erhalten möchte, um einen Händler damit zu bezahlen ist folgendes Protokoll gegeben. Hierbei ist **A** der Zahlungsleistende, **B** der Zahlungsempfänger, **C** die Bank. $\mathbf{S_C(M)}$ stellt einen Geldschein **M** dar.

Schritt	Akteur	Operation
1	A	A wählt (ein beliebiges) M, sodass r(M) gilt
2	A	A berechnet d(M) und blendet dadurch M. A sendet nur die geblendete Nachricht d(M) an C
3	C	C signiert d(M) mit V durch $S_C(d(M))$. C signiert hierbei eine geblendete Nachricht und kann diese daher nicht lesen.
4	C	C sendet $S_C(d(M))$ an A
5	A	A wendet d' an mit $d'(S_C(d(M)))=S_C(M)$ und hat somit eine gültige Signatur S_C für M, da d(M) mit d' invertiert wird und die bereits signierte Nachricht entblendet wird
6	A	A überprüft $V_C(S_C(M))=M$; und stoppt, wenn falsch. Hierbei prüft A durch Entschlüsseln der Signatur, ob diese gültig ist
7	A	A bezahlt B durch Senden von $S_C(M)$ an B
8	B	B überprüft $r(V_C(S_C(M)))$; und stoppt, wenn falsch. Hierbei prüft B durch Entschlüsseln der Signatur, ob diese gültig ist
9	B	B sendet $S_C(M)$ an C
10	C	C überprüft $r(V_C(S_C(M)))$; und stoppt, wenn falsch. Hierbei prüft C durch Entschlüsseln der Signatur, ob diese gültig ist
11	C	C fügt Geldschein $S_C(M)$ zur Liste aufgebrauchter Scheine hinzu; und stoppt, wenn Schein bereits auf Liste
12	C	C zahlt Geldwert an B aus
13	C	C informiert B über Annahme der Zahlung

Tabelle 3.2: Protokoll nach Chaum für Finanztransaktion

Hierbei ist zu beachten, dass **C** in Schritt 9 einen (von A) selbst unterschriebenen Geldschein erhält, jedoch keinerlei Informationen darüber hat, dass dieser ursprünglich in Schritt 4 an **A** ausgestellt wurde. [6]

3.2 Anwendung bei RSA und Paillier

In dem ursprünglichen Vorschlag „Blind signatures for untraceable payments“ gibt Chaum kein konkretes Verfahren oder System an, bei dem diese Signaturen Anwendung finden sollen. In diesem Kapitel wird ein Protokoll für die Erstellung einer blinden Signatur mit Hilfe des RSA- und des Paillier-Verfahrens gezeigt.

Für das RSA-Verfahren sei **(n,e)** der öffentliche Schlüssel von **C**, **d** der private Schlüssel von **C** und **s** die Signatur. Das zuvor beschriebene zufällige **x** sei nun ein Zufallswert **k** und **M** eine Nachricht, mit **s** als Signatur. Die dazugehörige geblendete Nachricht sei **M***, mit **s'** als Signatur, zuvor als $\mathbf{S_C(d(x))}$ bezeichnet. Dies ergibt das in Tabelle 3.3 beschriebene Protokoll.

Schritt	Akteur	Operation
1	A	A wählt zufällige Zahl k
2	A	A berechnet $M' = M * k^e \bmod n$. Hiermit blendet A die Nachricht M
3	A	A sendet M' (die geblendete Nachricht) an C
4	C	C erstellt Signatur für M' mit $s' = M'^d \bmod n$. C signiert hiermit M'
5	C	C sendet s' an A und belastet das Konto von A mit dem Wert von M
6	A	A berechnet s aus $s' = (M * k^e)^d \bmod n = M^d * k^{ed} \bmod n$ $= M^d * k \bmod n$, und berechnet $s = s' / k = M^d \bmod n$. Hiermit entblendet A die Signatur
7	A	A verwendet s als Signatur für M

Tabelle 3.3: Protokoll einer blinden Signatur mit RSA [28, 8]

Die zuvor geforderte Funktion **d** mit dem Inversen **d'** ist hierbei durch die Multiplikation mit $\mathbf{k^e \bmod n}$ gegeben und der Umkehrung durch die Division durch **k mod n**, nachdem durch die Potenzierung mit dem privaten Schlüssel **d** die vorherige Potenzierung von **k** mit dem öffentlichen Schlüssel **e** die Potenz aufgehoben wurde. Bei Durchführung dieser Schritte fällt auf, dass das Herausrechnen von **k** aus **s'** nur aufgrund der homomorphen Eigenschaften[1] des RSA-Verfahrens gelingt. Des Weiteren fällt auf, dass **C** zu keinem Zeitpunkt wirklich weiß, welchen Wert **M** besitzt und die Signatur kann theoretisch mehrfach verwendet werden. Diese Probleme werden in Kapitel 3.4 genauer erörtert.

[1] RSA ist multiplikativ-homomorph, d.h. für eine RSA-Verschlüsselung ε gilt: $\varepsilon(\mathbf{a}) * \varepsilon(\mathbf{b}) = \mathbf{a^e b^e \bmod n = (ab)^e \bmod n = \varepsilon(a*b)}$. Durch diese Eigenschaft löst sich die Potenz vom Blendfaktor $\mathbf{k^e}$, beim Signieren, auf und es ist möglich, durch Dividieren mit **k** die Signatur zu entblenden.

Mit dem Paillier-Kryptosystem sieht das Erstellen von blinden Signaturen wie in Tabelle 3.4 dargestellt aus.

Schritt	Akteur	Operation
1	A	**Wählt zufällige Zahl k**
2	A	**Berechnet M' = M * k^n** (Blenden)
3	A	**Sendet M' an C**
4	C	**Erstellt Signatur für M' mit s'(M') = (s_1,s_2')**
5	C	**Sendet s' an A, belastet Konto von A mit Wert von M**
6	A	**Berechnet s aus s'(M') = (s_1,s_2') mit (s_1,s_2'k^{-1} mod n) und erhält s(M) = (s_1,s_2)** (Entblenden)
7	A	**Nutzt s als gültige Signatur für M**

Tabelle 3.4: Protokoll einer blinden Signatur mit Paillier [4, S. 7 f.]

In diesem Beispiel ist die von Chaum geforderte Funktion **d** mit dem Inversen **d'** gegeben durch die Multiplikation der Nachricht mit $\mathbf{k^n}$ und die Umkehrung mit der späteren Multiplikation von $\mathbf{s_2}$' mit $\mathbf{k^{-1}}$ **mod n**. Bei Durchführung dieser Schritte fällt auf, dass in diesem Fall erneut die homomorphen Eigenschaften des zugrundeliegenden Verfahrens genutzt werden. Der Beweis, dass das in Schritt 6 benutzte $\mathbf{s(M) = (s_1, s_2'*k^{-1} \bmod n)}$ gilt, und somit eine für eine geblendete Nachricht erstellte Paillier-Signatur entblendet werden kann, findet sich bspw. bei A. Tueno [4, S. 7 f.]. Die gleichen Probleme wie bei dem RSA-Verfahren, dass **C** der Wert von **M** unbekannt ist und die Signatur mehrfach verwendet werden könnte, bestehen fort.

3.3 Vorteile

Die blinde Signatur nach David Chaum besitzt durch die Unterschiede zu normalen Signaturen zusätzlich die vorteilhaften Eigenschaften Blindheit (englisch: Blindness), Nicht-Nachverfolgbarkeit (englisch: Untraceability) sowie Unverkettbarkeit (englisch: Unlinkability).

- **Blindheit** bedeutet, dass der Signaturleistende C den Inhalt des zu signierenden Dokuments nicht kennt und von einer geblendeten Nachricht M‘ keine Verbindung zu M herstellen kann. (nach [6, 17])

- **Nicht-Nachverfolgbarkeit** bedeutet, dass die erstellte Signatur nicht zu A zurückverfolgt werden kann. (nach [18])

- **Unverkettbarkeit** bedeutet, dass es nicht möglich ist eine Verbindung zwischen zwei Signaturen herzustellen, außer dass diese von C erstellt wurden. (nach [23, S. 13])

Diese Eigenschaften bieten daher bei der Verwendung von blinden Signaturen den Vorteil des Schutzes vor Nachverfolgung und Verkettung von Nachrichten durch Dritte, die keine direkten Kommunikationspartner sind. Der Kommunikationspartner von A selbst ist ebenfalls nicht in der Lage, Nachrichten von A nachzuverfolgen oder zu verketten. Aber im Rahmen der Kommunikation sind A und B sich gegenseitig bekannt.

3.4 Nachteile

Natürlich ergeben sich aus der Funktionsweise von blinden Signaturen auch Nachteile und Probleme. Mit den neuen Eigenschaften Blindheit, Nicht-Nachverfolgbarkeit sowie Unverkettbarkeit geht die Eigenschaft der Integrität verloren. Die Signatur stellt nämlich nicht länger eine Aussage dar, die garantiert, dass der Inhalt der Nachricht unverändert ist. Anstelle dessen ist die Signatur eine Aussage, dass zu einem bestimmten Zeitpunkt dem Signaturleistenden **C** eine geblendete Nachricht vorlag.

Im Folgenden werden 2 wichtige Probleme erläutert:

- **Blind Signing** In Kapitel 3.2 wurde bereits angesprochen, dass **C** nicht weiß, welchen Wert M besitzt. Um den Blendfaktor entfernen zu können, ist es notwendig, dass **M**' signiert wird und nicht ein Hashwert von **M**'. Im Fall von Chaums Beispiel eines Zahlungsprozesses könnte **A** eine Nachricht **M** erstellen und behaupten, diese hätte einen Wert, den sie tatsächlich nicht besitzt. Unterzeichnet **C**, im Beispiel die Bank, diese Nachricht **M** blind, könnten digitale Geldscheine unterzeichnet werden, deren Wert sich von dem unterscheidet, was die Bank vom Konto von A abbucht. Da es **C** nicht erlaubt oder möglich ist, den Wert zu überprüfen, muss **C** auf die Korrektheit der Nachricht vertrauen. Damit wird also eine gültige Signatur für etwas erstellt, das **C** nicht signiert hätte, wenn seine Kenntnis, über den tatsächlichen Wert, vorläge. Dies wird als Blind-Signing-Angriff bezeichnet. [3]

- **Double Spending** Ein weiteres Problem ist das Wiederverwenden der erstellten Signatur (dies sollte nicht verwechselt werden mit der Eigenschaft Nicht-Wiederverwendbarkeit). Beim Double Spending geht es nicht um das Wiederver-

wenden der blinden Signatur bei verschiedenen Nachrichten, sondern um das erneute Senden der Signatur, welche im Beispiel Chaums einen Geldschein darstellt. Es besteht also die Gefahr, dass ein Geldschein in Form dieser Signatur mehrfach ausgegeben wird.[11, S. 236 f.]

Lösungen für diese Nachteile werden in Kapitel 5 beschrieben.

3.5 Varianten

Für blinde Signaturen haben Matthew Franklin und Moti Yung 1994 drei verschiedene Arten von blinden Signaturen klassifiziert [10]:

- **Signaturen mit blinder Verifikation** – verhindern, dass der Signaturleistende später die Signatur wiedererkennen kann, ohne notwendigerweise die Nachricht vor ihm zu verbergen. [10]

- **Signaturen mit blinden Nachrichten** – verhindern, dass der Signaturleistende die zu signierende Nachricht erkennen kann, ohne notwendigerweise die Signatur vor ihm zu verbergen. [10]

- **Vollständig blinde Signaturen** – kombinieren blinde Verifikation und blinde Nachrichten in einem einzigen Signaturverfahren (beispielsweise Chaums ursprünglicher Vorschlag). [10]

Ebenso wurden im Jahr 1994 kurz darauf von Patrick Horster und Holger Petersen vier verschiedene Varianten vorgestellt [13, S. 2]:

- **Verdeckte Signaturen** – hierbei kennt der Signaturleistende nicht das Dokument, welches er signiert, allerdings die „Signaturparameter“. Sollte er diese speichern, kann er diese später wiedererkennen durch einen Vergleich mit einer gegebenen Signatur. [13, S. 2]

- **Schwach-blinde Signaturen** – hierbei kennt der Signaturleistende weder das Dokument noch die Signaturparameter, kann sie jedoch später wiedererkennen, weil eine Beziehung zwischen den Parametern der geblendeten und ungeblendeten Signatur besteht. [13, S. 2]

- **Interaktiv-blinde Signaturen** – hierbei ist die Erzeugung der Signatur gleich mit dem Fall von schwach-blinden Signaturen. Es besteht eine Beziehung zwischen geblendeten und ungeblendeten Signaturparametern, jedoch kann der Signaturleistende keine Informationen hierüber erlangen. Notfalls kann der Besitzer im Falle einer Beschwerde gezwungen werden, die Signaturparameter anzugeben, wodurch die Beziehung zwischen den Daten erkannt werden kann. [13, S. 2]

- **Stark-blinde Signaturen** – hierbei kann der Signaturleistende keine Beziehung zwischen jeglichen gespeicherten Parametern und den gegebenen Signaturparametern erkennen. Die Signatur ist hier komplett anonym. [13, S. 2]

Beide Arten von Einteilungen richten sich nach dem Grad der Anonymität, der gewonnen werden kann. Allerdings fokussierten Franklin und Yung sich mehr auf die Möglichkeiten, zuvor definierte schwache Signaturen zu blenden und sind daher recht speziell. Die Einteilung von Horster und Petersen ist detaillierter und betrachtet auch mögliche Verbindungen zwischen gespeicherten Daten beim Signaturprozess. Des Weiteren entsprechen die Arten interaktiv-blind und stark-blind gemäß dieser Einteilung am ehesten der ursprünglichen Idee Chaums. Insbesondere entspricht die stark-blinde Signatur dem anfänglichen Konzept von blinden Signaturen. Interaktiv-blinde Signaturen entsprechen der Signatur, die gemäß dem Protokoll in Kapitel 5.2.2 erstellt wird. Da diese Einteilung in 4 Arten also präziser und dem Konzept näher ist, sollte sie bevorzugt werden.

3.6 Weitere Signaturarten von Chaum

Chaum entwarf außer blinden Signaturen die folgenden weiteren Signaturarten: 1990 **Nicht-Abstreitbare Signaturen** (undeniable signatures) und 1991 sogenannte **Gruppensignaturen** (group signatures). Diese hatten teilweise ebenfalls das Ziel, eine Form von Anonymität für den Sender einer Nachricht zu bieten, erreichten dies jedoch auf andere Weise als blinde Signaturen und besitzen nicht die gleichen Eigenschaften.

Nicht-abstreitbare Signaturen beschreiben hierbei Signaturen, die nicht ohne Kooperation des Signaturleistenden verifiziert werden können. Die Gültigkeit einer Signatur kann durch ein Protokoll überprüft werden. Dabei wird zwischen „confirmation“ und „disavowal protocols“, d.h. zwischen Bestätigungs- und Bestreitungs-Protokollen, unterschieden. Diese Protokolle sollen dazu dienen, ohne Wissen über den öffentlichen Schlüssel des Signaturleistenden preiszugeben, mit einer exponentiell hohen Wahrscheinlichkeit Aussagen zu treffen, ob eine Signatur eine gültige Signatur des Signaturleistenden ist oder nicht. [7]

Bei dieser Art von Signaturen wird eine Art Anonymität dadurch geschaffen, dass es Empfängern einer Nachricht nicht möglich ist, den Absender einer Nachricht zu bestimmen, d.h. die Signatur zu überprüfen, wenn dieser es nicht möchte. Dies ist von blinden Signaturen zu unterscheiden, wo der Sender einer Nachricht diese nicht selbst signiert und somit nicht eindeutig identifiziert werden kann. Ebenso kann, sofern der Sender einer Nachricht bei der Überprüfung der Gültigkeit kooperiert, eine Verbindung zwischen mehreren Nachrichten hergestellt werden, wenn sie von dem gleichen Absender stammen. Die Signatur ist damit sowohl nachverfolgbar als auch verkettbar. [7]

Gruppensignaturen beschreiben Signaturen, bei denen – wie bei blinden Signaturen – die Unterschrift nicht vom Ersteller einer Nachricht geleistet wird. Es wird hier von einer Gruppe, der der Sender angehört, signiert. Jede Gruppe hat hierbei einen Manager, der Mitglieder der Gruppe hinzufügen und identifizieren kann. Das Signieren erfolgt im Namen der Gruppe. Der Sender einer solchen Nachricht ist daher anonym gegenüber Empfängern der Nachricht, welche nur einsehen können, dass die Nachricht von der Gruppe signiert wurde. Ebenso kann von Außenstehenden keine Verbindung zwischen zwei Nachrichten hergestellt werden, um zu bestimmen, ob der Sender der gleiche ist. [9]

Die Nachricht wird also nicht blind signiert. Die Signatur stellt also keine Aussage dar, dass die Nachricht zu einem bestimmten Zeitpunkt einer bestimmten Institution vorlag, sondern nur dass die Nachricht von einem Mitglied einer Gruppe stammt. Der Sender einer Nachricht ist jederzeit durch den Gruppen-Manager identifizierbar, welcher somit Nachrichten nachverfolgen kann und Verbindungen zwischen mehreren gesendeten Nachrichten erkennen kann.

4 Implementierung in CrypTool 2

In diesem Kapitel wird auf die Implementierung der blinden Signaturen in CT2 eingegangen. Dabei wird zuerst das Konzept erläutert und anschließend werden die notwendigen Schritte bei der Erstellung der Komponente aufgezeigt. In diesem Kapitel werden neben den Definitionen für Ein-/Ausgänge der zu implementierenden Plugins und zusätzlicher Dateien zur Dokumentation, vor allem der Ablauf der Berechnungen bei Aufruf der Execute-Methode im Rahmen des Lebenszyklus einer Komponente, wie in Kapitel 2.6.4 schematisch dargestellt, beleuchtet.

In den folgenden Beschreibungen der Implementierung ist darauf zu achten, dass zwecks Einheitlichkeit mit anderen CT2 Komponenten für den öffentlichen Modulus sowie die öffentlichen und privaten Schlüssel bei RSA und Paillier Großbuchstaben als Bezeichner verwendet werden, statt Kleinbuchstaben.

4.1 Konzept

Bei der Realisierung von blinden Signaturen in CT2 werden zwei Komponenten implementiert: 1. eine Signier-Komponente, die sowohl mit dem RSA als auch dem Paillier-Verfahren eine blinde Signatur erstellen kann. 2. eine Verifizierungs-Komponente, mit der die Korrektheit der blinden Signatur überprüft werden kann.

Hierbei sind die Eigenheiten der jeweiligen Signatursysteme zu beachten. Es werden zusätzliche Schlüsselpaare generiert, da die Signatur nicht vom Ersteller der Nachricht, sondern einer von diesem unabhängigen Partei geleistet wird. Die Komponente soll hierbei alle Schritte durchlaufen, welche in den Kapiteln zu RSA und Paillier beschrieben wurden.

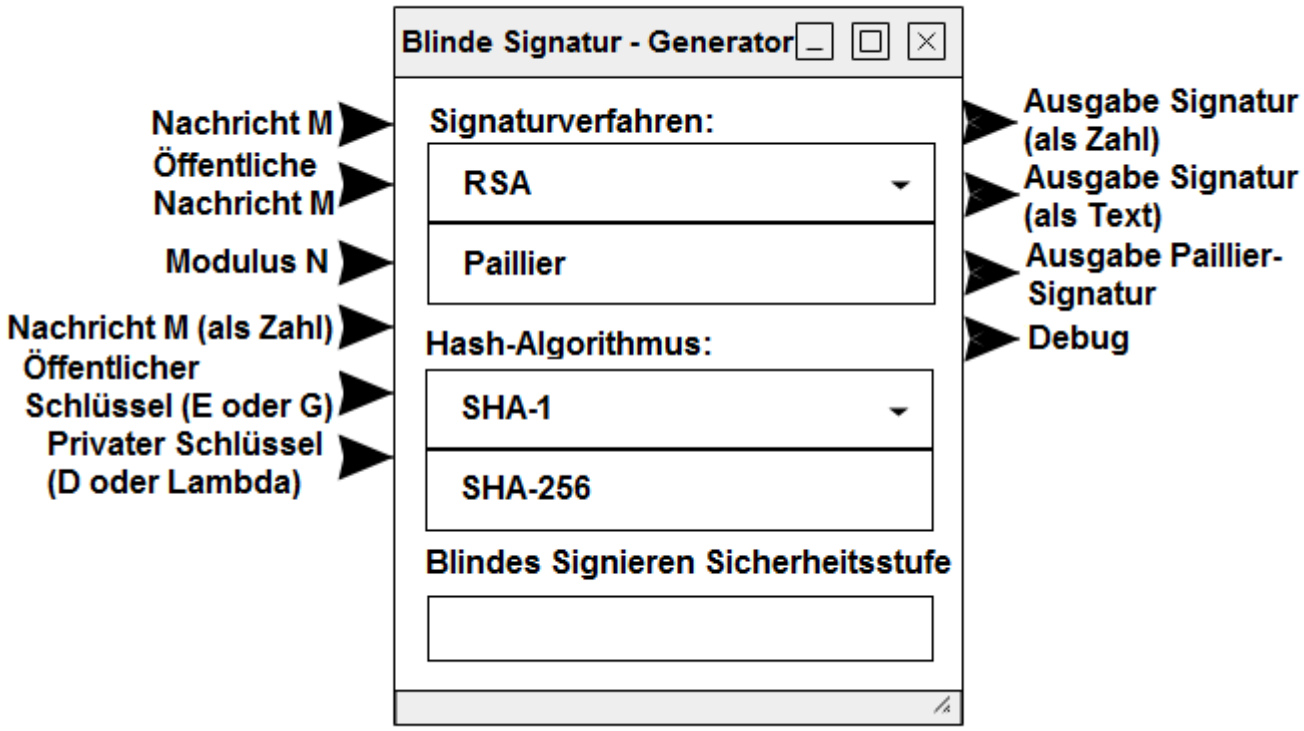


Abbildung 4.1: Mockup der Komponenten-Einstellungen für blinde Signaturen

Die in Abb. 4.1 dargestellten Bezeichnungen haben die folgenden Bedeutungen:

- **Nachricht M** — Hierüber soll der Nutzer seine Nachricht sowohl als Zahl, Text als auch als Byte-Array verbinden können. Diese Nachricht wird geblendet und signiert.

- **Öffentliche Nachricht M** — Hierüber soll der Nutzer seine öffentlich einsehbare Nachricht sowohl als Zahl, Text als auch Byte-Array verbinden können, wenn er einen Blind-Signing-Angriff simulieren möchte. Diese Nachricht ist optional und für den Signaturleistenden einsehbar.

- **Modulus N** — Hierüber wird der Modulus des RSA- bzw Paillier-Schlüssels verbunden.

- **Öffentlicher Schlüssel (E oder G)** — Hierüber wird der öffentliche Exponent e des RSA-Schlüssels verbunden bzw. der öffentliche Schlüssel G des erzeugten Paillier-Schlüsselpaares.

- **Privater Schlüssel (D oder Lambda)** — Hierüber wird der private Schlüssel d des RSA-Schlüssels bzw. der private Schlüssel Lambda des erzeugten Paillier-Schlüsselpaares verbunden.

- **Signaturverfahren** — Hierüber soll der Nutzer zwischen RSA und Paillier wählen können. Der Nutzer muss selbst darauf achten, die dazu passenden Schlüssel zu verbinden.

- **Hash-Algorithmus** — Hierüber soll der Nutzer zwischen 4 SHA-Hash-Algorithmen, SHA1, SHA256, SHA384 und SHA512 wählen können, sowie auch kein Hashverfahren anwählen können.

- **Blindes Signieren Sicherheitsstufe** — Hierüber soll der Nutzer eingeben können, wie sicher die Komponente gegen einen simulierten Blind-Signing-Angriff sein soll. Hierbei ist für eine eingegebene Zahl **T** die Erfolgswahrscheinlichkeit eines solchen Angriffs **1/T**. Als Standardwert ist 500 eingestellt.

- **Ausgabe Signatur (als Zahl)** — Hierüber wird die Signatur als Zahl ausgegeben.

- **Ausgabe Signatur (als Text)** — Hierüber wird die Signatur als Text ausgegeben.

- **Ausgabe Paillier-Signatur** — Hierüber wird die Paillier-Signatur als BigInteger-Array ausgegeben. Das Paillier-Verfahren benötigt als Spezialfall ein BigInteger-Array, da die Signatur ein Tupel von Werten ist.

- **Debug** – Hierüber werden alle Zwischenergebnisse der einzelnen Schritte als Text ausgegeben.

Der Input einer öffentlichen Nachricht M ist hierbei der einzige optionale Dateneingang.

Vorausgesetzt, dass die für die gewählten Verfahren korrekten Schlüssel erzeugt und verbunden wurden, führt die Komponente intern Folgendes durch:

Mit dem gewählten Verfahren wird ein Hashwert für die zu signierende Klartext-Nachricht berechnet. Daraufhin wird eine Zufallszahl gewählt und diese bei RSA mit dem öffentlichen Exponenten **e**, bei Paillier mit dem öffentlichen Schlüssel **n** potenziert. Der erhaltene Wert wird mit dem zuvor berechneten Hashwert multipliziert.

Dieser somit geblendete Hashwert wird mittels den bereits implementierten Verfahren RSA und Paillier signiert. Die erhaltene Signatur für den geblendeten Hashwert wird

nun entblendet durch die Multiplikation mit dem multiplikativen Inversen des Blendfaktors.

Anschließend wird die damit erstellte Signatur ausgegeben. Die Komponente kann damit eine blinde Signatur vollständig und korrekt erstellen. Intern muss vor allem auf die Typ-Konvertierungen geachtet werden, um Informationsverlust oder Informationsverfälschung zu verhindern.

Zur Verifizierung der erstellten Signatur wird eine weitere Komponente erstellt. Diese hat folgende Form:

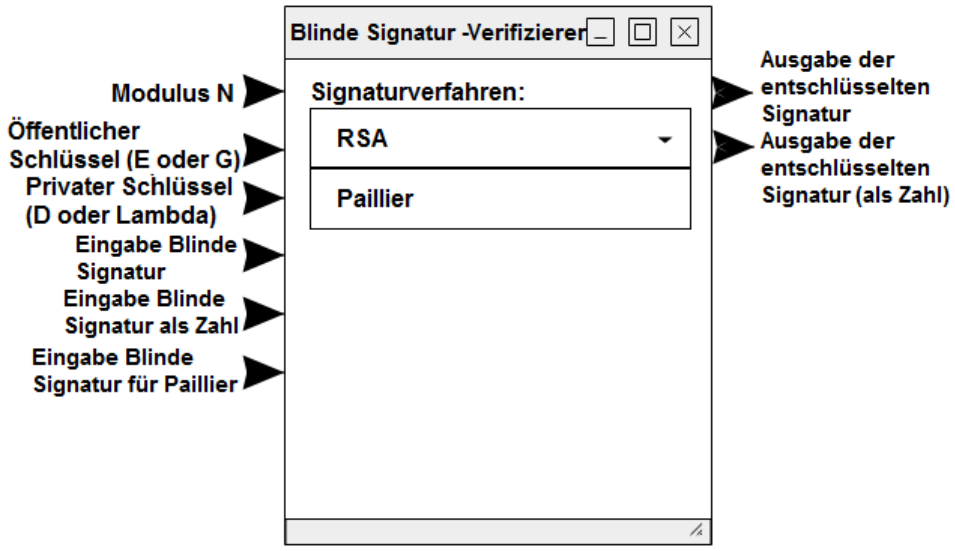


Abbildung 4.2: Mockup der Verifizierungs-Komponente

Die Eingaben der Komponente in Abb. 4.2 sind hierbei erneut der Modulus sowie die öffentlichen und privaten Schlüssel. Hierbei ist zu beachten, dass bei Auswahl von RSA die Signatur als BigInteger oder als Byte-Array vorliegen kann, bei Auswahl von Paillier aber als BigInteger-Array vorliegen muss. Das BigInteger-Array enthält den Tupel (s_1,s_2).

Die Ausgabe-Typen dieser Verifizierungs-Komponente stimmen mit den Ausgabetypen der Paillier- und RSA-Verschlüsselungs-Komponenten überein. Dabei ist zu beachten, dass zur Verifizierung der blinden Signatur bei RSA die bereits vorhandene RSA-Komponente genutzt werden kann. Allerdings ist dies für Paillier nicht der Fall, da die schon vorhandene Paillier-Komponente nur Verschlüsselung und Entschlüsselung, aber nicht Signaturerstellung und Verifikation enthält.

Für die WPF-Präsentation innerhalb der Signier-Komponente wurde folgendes Konzept (siehe Abb. 4.3) erstellt:

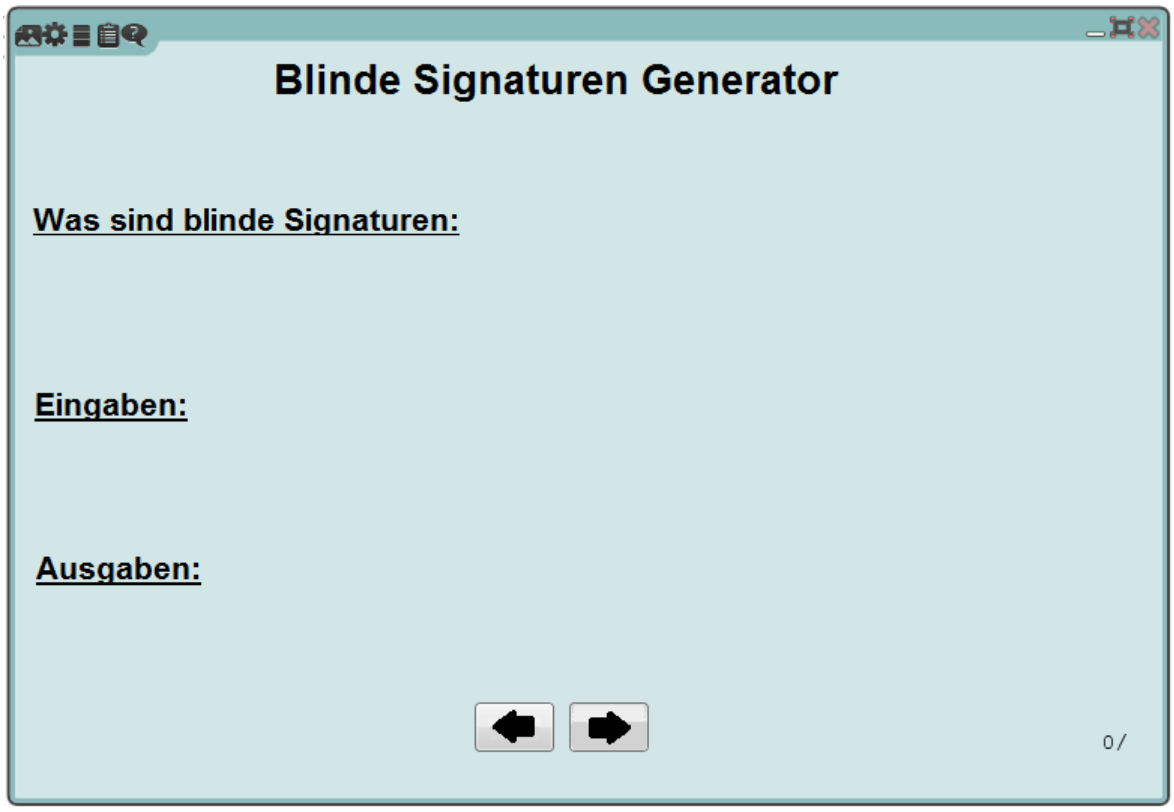


Abbildung 4.3: Mockup der Präsentation der Komponente

Die Komponente erklärt dem Nutzer in Form einer Slideshow, was blinde Signaturen sind und wofür sie dienen. Die Ein- und Ausgabe-Daten sowie die Einstellungen sollen beschrieben werden. Anschließend werden für die verwendeten Daten alle internen Zwischenschritte und Zwischenergebnisse dargestellt. Für den Nutzer soll somit ersichtlich und nachvollziehbar sein, wie die Komponente funktioniert und wofür er sie einsetzen kann.

4.2 Implementierung des Blinde-Signatur-Generators

Für die Implementierung wird zu den bereits vorhandenen CT2-Plugins ein neues Projekt unter Nutzung des öffentlich zugänglichen CT2-Templates [2] hinzugefügt.

Das Zielumgebung zur Zeit der Erstellung der Komponente ist das .NET Framework 4.7.2. Eingebundene Bibliotheken sind unter anderem CrypTool.Pluginbase, System.Numerics, System.Security.Cryptography.

Die Entwicklungsumgebung ist Visual Studio Community 2019 Version 16.1.6. Die Code-Beispiele in den Screenshots wurden hieraus entnommen.

```
[TaskPane("Signing Algorithm", "Choose the signing algorithm that you want to use.", null, 1, false, ControlType.ComboB
2 references
public SigningMode SigningAlgorithm
{
    get
    {
        return this.selectedSigningMode;
    }
    set
    {
        if (value != selectedSigningMode)
        {
            this.selectedSigningMode = value;
            OnPropertyChanged("SigningAlgorithm");
        }
    }
}
[TaskPane("Hash Algorithm", "Choose the hash algorithm that you want to use.", null, 1, false, ControlType.ComboBox, ne
1 reference
public HashMode HashAlgorithm
{
    get
    {
        return this.selectedHashMode;
    }
    set
    {
        if (value != selectedHashMode)
        {
            this.selectedHashMode = value;
            OnPropertyChanged("HashAlgorithm");
        }
    }
}
```

Abbildung 4.4: Definition der Einstellungsmöglichkeiten der Komponente

Wie in Abb. 4.4 dargestellt, wurden für die Einstellungen der Komponente, Enumerations für die wählbaren Signier-Verfahren und Hash-Verfahren erstellt. Diese können in Form einer DropBox in den Einstellungen vom Nutzer ausgewählt werden.

Anschließend wird die Funktionalität der Komponente implementiert. Hierzu werden die für die Komponente zulässigen Daten-Eingänge und Daten-Ausgänge definiert, wie in Abb. 4.5 zu sehen. Da die Komponente intern die Verschlüsselungen für die verfügbaren Verfahren durchführen soll, benötigt sie, zusätzlich zu der Nachricht, den öffentlichen sowie privaten Schlüssel und den Modulus. Der Datentyp der Eingabe wird als Object definiert, um zu ermöglichen, dass eine Eingabe in unterschiedlichen Datentypen wie String, BigInteger oder Byte-Array möglich ist. Die Ausgabe der Signatur erfolgt sowohl als Byte-Array als auch als BigInteger. Der Debug-Ausgang wird bei allen internen

```
#region Data Properties

[PropertyInfo(Direction.InputData, "MessageM", "MessageMTooltip")]
4 references
public Object Message...

[PropertyInfo(Direction.InputData, "PublicMessage", "PublicMTooltip")]
8 references
public Object PublicMessage...

[PropertyInfo(Direction.InputData, "ModuloN", "ModuloNTooltip")]
43 references
public BigInteger Modulo
{
    get { return modulo; }
    set { modulo = value; }
}

[PropertyInfo(Direction.InputData, "PublicKey", "PublicKeyTooltip")]
6 references
public BigInteger PublicKey...
```

Abbildung 4.5: Definition der Ein- und Ausgänge der Komponente

Zwischenschritten aktualisiert, um alle Zwischenergebnisse als String aufzulisten und dem Nutzer zu ermöglichen, den Prozess nachzuverfolgen.

Der Ablauf der Ausführung der implementierten Komponente stellt ist wie folgt:

```
3 references
public byte[] DoHashes(byte[] m)
{
    byte[] calculatedhash = null;
    switch (settings.HashAlgorithm)
    {
        case BlindSignatureGeneratorSettings.HashMode.SHA1:
            using (SHA1 sha1Hash = SHA1.Create())
                calculatedhash = sha1Hash.ComputeHash(m);
            break;
        case BlindSignatureGeneratorSettings.HashMode.SHA256:
            using (SHA256 sha256Hash = SHA256.Create())
                calculatedhash = sha256Hash.ComputeHash(m);
            break;
        case BlindSignatureGeneratorSettings.HashMode.SHA384:
            using (SHA384 sha384Hash = SHA384.Create())
                calculatedhash = sha384Hash.ComputeHash(m);
            break;
        case BlindSignatureGeneratorSettings.HashMode.SHA512:
            using (SHA512 sha512Hash = SHA512.Create())
                calculatedhash = sha512Hash.ComputeHash(m);
            break;
        case BlindSignatureGeneratorSettings.HashMode.None:
            calculatedhash = m;
            break;
    }
    return calculatedhash;
}
```

Abbildung 4.6: Definition der switch-cases für den gewählten Hash-Algorithmus

Durch switch-cases (siehe Abb. 4.6) wird für den gewählten Hash-Algorithmus die korrekte Enumeration identifiziert und ein Hash für die eingegebene Nachricht erstellt. Die SHA-Algorithmen sind hierbei bereits durch die .NET-Bibliotheken implementiert und aufrufbar. Wenn **„None“** in den Einstellungen ausgewählt wurde, wird die Nachricht nicht gehasht, sondern nur in ein Byte-Array für die weitere interne Verwendung um-

gewandelt. Damit können Lehrbuchbeispiele und Zahlenbeispiele der Literatur, wie in Kapitel 2.5, wo als Nachricht oft ein Integerwert genommen wird, leicht nachvollzogen werden.

```
switch (settings.SigningAlgorithm)
{
    case BlindSignatureGeneratorSettings.SigningMode.RSA:
        //A creates its array of blinded messages and C checks for a cheating attempt.
        BlindSigningAttackDefenderRSA();
        //If A has not been found to be cheating the program resumes while using the c
        DebugBuilder.AppendLine(Resources.Hash_h_M__is__);
        DebugBuilder.AppendLine(ByteArrayToHexString(hash).Replace("-", " "));
        DebugBuilder.AppendLine(Resources.Random_Number_k_is__);
        DebugBuilder.AppendLine(blindingfactor.ToString());
        //signing-process with RSA is being carried out
        temp = BigInteger.ModPow(blindedmessage, PrivateKey, Modulo);
        temp = UnBlindingRSA(temp);
        signature = temp.ToByteArray();
        break;

    case BlindSignatureGeneratorSettings.SigningMode.Paillier:
        temp = new BigInteger(hash);
        //A creates its array of blinded messages and C checks for a cheating attempt.
        BlindSigningAttackDefenderPaillier();
        //calculate s1
        if (blindedmessage > Modulo) { GuiLogMessage("Message is bigger than N - this
        if (temp > Modulo) { GuiLogMessage("Message is bigger than N - this will produ
        BigInteger Modulo2 = Modulo * Modulo;
        BigInteger lambdainv = BigIntegerHelper.ModInverse(PrivateKey, Modulo);
        BigInteger s1 = (((BigInteger.ModPow(temp, PrivateKey, Modulo2) - 1) / Modulo)
        //calculate s2
        BigInteger InversePublicKey = BigIntegerHelper.ModInverse(PublicKey, Modulo);
        BigInteger s2 = BigInteger.ModPow(InversePublicKey, s1, Modulo);
        s2 = blindedmessage * s2;
        BigInteger R3 = BigIntegerHelper.ModInverse(Modulo, PrivateKey);
        s2 = BigInteger.ModPow(s2, R3, Modulo);
        // s2 is now being un-blinded with s2 = s2 * (k^-1 mod n) mod n
        s2 = UnBlindingPaillier(s2);
        //signature is (s1,s2)
        PaillierSignature = new BigInteger[2];
        PaillierSignature[0] = s1;
        PaillierSignature[1] = s2;
        break;
}
```

Abbildung 4.7: Definition der switch-cases für das gewählte Krypto-Verfahren

Im nächsten Schritt wird – erneut mittels switch-cases – zu sehen in Abb. 4.7, für das gewählte Krypto-Verfahren die korrekte Enumeration identifiziert und für das jeweilige Verfahren der Hash korrekt geblendet. Das Blenden erfolgt hierbei innerhalb der Hilfsmethode zur Abwehr eines Blind-Signing-Angriffs, welche zu Beginn der Verfahrensunterscheidung aufgerufen wird und in Kapitel 5 näher beschrieben ist.

Die Komponente kann die Funktionen der bisherigen RSA- und Paillier-Komponenten nicht einfach wiederverwenden:

1. Der vorhandene RSA-Code führt zu Fehlern, aufgrund der Zerlegung der geblendeten Nachricht in Segmente, die einzeln verschlüsselt werden. Aus diesem Grund wird für die Implementierung von RSA der geblendete Hash als BigInteger ohne weitere Zerlegung in Segmente mit dem privaten Schlüssel **d** potenziert und modulo dem öffentlichen Modulus **N** gerechnet.

2. Die Berechnung der Paillier-Signatur musste von Grund auf neu geschrieben werden, da die vorhandene Paillier-Komponente keine Signatur erstellen kann. Zu diesem Zweck wird der Signier-Algorithmus in mehrere Unterschritte unterteilt und in Zwischenergebnissen gespeichert. Die erstellten Signaturen für das Paillier-Verfahren werden in einem BigInteger-Array gespeichert und als solche ausgegeben. Eine Konvertierung in Byte-Arrays ist nicht nötig. Danach wird die entblendete Signatur als Ergebnis ausgegeben und alle Daten-Ausgänge aktualisiert. Im Rahmen der Implementierung konnte auch ein Fehler in der bisherigen Paillier-Schlüsselgenerator-Komponente von CT2 gefunden werden, bei dem für Lambda nur ein gemeinsames Vielfaches von **(p-1,q-1)**, aber nicht das *kleinste* gemeinsame Vielfache ausgegeben wurde.

4.3 Implementierung des Blinde-Signatur-Verifizierers

Bei der Implementierung der Verifizierungs-Komponente wird ebenso wie bei der Generator-Komponente vorgegangen. Nach der Definition der Einstellungen sowie der Ein- und Ausgänge des Plugins ist die Execute-Methode (zu sehen in Abb. 4.8) von besonderem Interesse.

In der Execute-Methode wird zuerst erkannt, welcher Daten-Eingang belegt wurde. Basierend darauf kann erneut über switch-cases je nach Signier-Algorithmus die Signatur verifiziert werden. Für RSA wird hierbei die Signatur durch Potenzierung mit dem öffentlichen Exponenten **e** modulo **N** entschlüsselt. Für Paillier wird – wie in Kapitel 2.5 beschrieben – $\mathbf{g^{s_1}}$ berechnet und das Ergebnis mit $\mathbf{s_2{}^n}$ multipliziert. Das gesamte Ergebnis wird anschließend **mod** $\mathbf{n^2}$ genommen. Beide switch-cases geben die somit

```
public void Execute()
{
    ProgressChanged(value:0, max:100);
    BigInteger temp = 0;
    if (BlindSignatureIn != null)
    {
        temp = new BigInteger(BlindSignatureIn);
    }
    if (BlindSignatureNumberIn != 0)
    {
        temp = BlindSignatureNumberIn;
    }

    BigInteger s1 = 0;
    BigInteger s2 = 0;
    if (BlindSignaturePaillierIn != null && BlindSignaturePaillierIn[0] != 0 && Blir
    {
        s1 = BlindSignaturePaillierIn[0];
        s2 = BlindSignaturePaillierIn[1];
    }
    switch (settings.SigningAlgorithm)
    {
        case BlindSignatureVerifierSettings.SigningMode.RSA:
            temp = BigInteger.ModPow(value:temp, exponent:PublicKey, modulus:Modulo);
            signature = temp.ToByteArray();
            break;
        case BlindSignatureVerifierSettings.SigningMode.Paillier:
            BigInteger Modulo2 = BigInteger.Pow(Modulo, exponent:2);
            temp = (BigInteger.ModPow(value:PublicKey, exponent:s1, modulus:Modulo2) *
            signature = temp.ToByteArray();
            break;
    }
```

Abbildung 4.8: Execute-Methode der Verifizierer-Komponente

entschlüsselte Signatur als Byte-Array zurück, welche dann über das Plugin als Byte-Array und BigInteger Zahl ausgegeben wird und anschließend verwendet werden kann, beispielsweise für weitere Konvertierungen und anschließende Vergleiche.

Die neue Komponente bietet die geforderten Funktionalitäten und ermöglicht es auf einfache Weise weitere Hash- oder Krypto-Verfahren zur Komponente hinzuzufügen. Dazu ist es notwendig die Enumerations und Drop-Down-Auswahl der Einstellungen zu erweitern, neue switch-cases hinzufügen und die benötigten Verschlüsselungs-Methoden von den vorhandenen CT2-Plugins zu übernehmen oder neu zu implementieren.

4.4 Implementierung des Präsentation-Modus der Komponente

Der Präsentations-Modus der CT2-Komponente wurde in WPF programmiert. Hierzu wurde für die Erstellung einer Slideshow eine Oberfläche definiert mit vorgegebenen Textfeldern, deren Inhalt für den jeweils darzustellenden Schritt aktualisiert wird. Der Nutzer navigiert zwischen den einzelnen Berechnungsschritten mittels der Buttons im unteren Bereich der Präsentations-Darstellung. Für die Buttons wiederum wird in C# die Funktionalität zur Aktualisierung des Textes, für den jeweils aktuellen Schritt, hinterlegt. Ein interner Zähler zeichnet den aktuell darzustellenden Schritt auf und bei jeder Bedienung der Buttons wird für den erkannten Schritt der Text aktualisiert. Hierbei ist zu beachten, dass die gesamte Präsentation nur genutzt werden kann, wenn die Komponente mindestens einmal gestartet wurde, da sonst keine Zwischenergebnisse existieren, die dargestellt werden könnten.

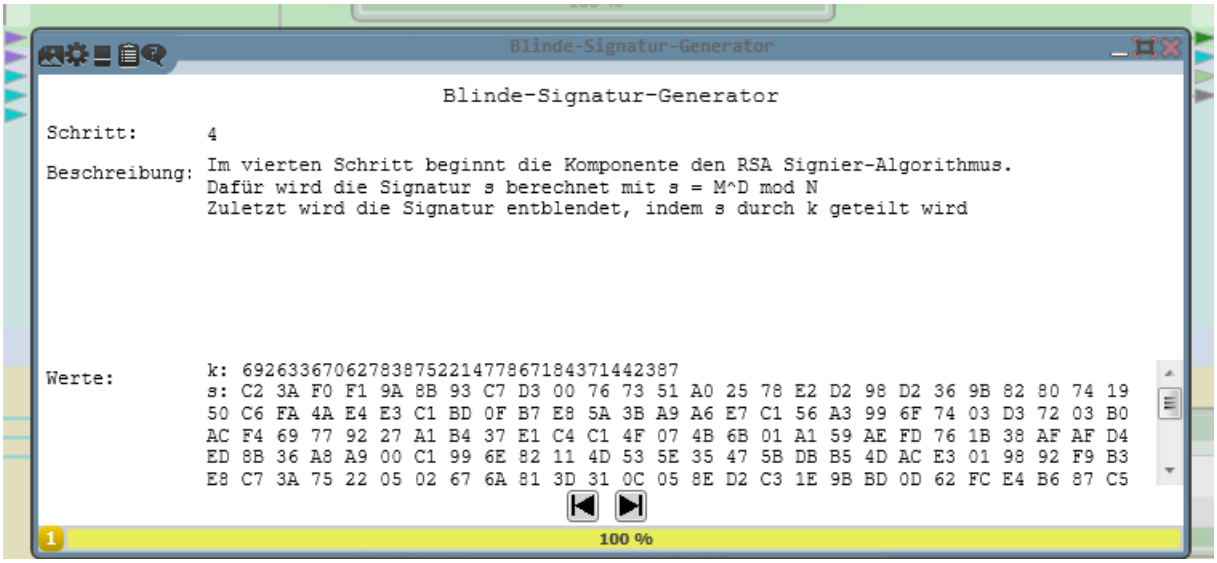


Abbildung 4.9: Präsentations-Modus der Komponente

Die Präsentation (Ausschnitt in Abb. 4.9) ist gleichzeitig so aufgebaut, dass ohne größeren Aufwand neue Schritte hinzugefügt werden können oder der darzustellende Text geändert werden kann. Dies ermöglicht zukünftige Erweiterungen.

Um die Mehrsprachigkeit der Komponenten zu gewährleisten, werden Namen und Beschreibungen der Bestandteile der Komponente in einer Ressourcen-Datei siehe Abb. 4.10 abgelegt. CT2 lädt für die eingestellte Sprache die jeweilige Übersetzung der Komponente und die dazugehörige Präsentation. Dazu wird dem Projekt eine .resx

Name	Wert
BlindSignature	Blinde Signatur S
BlindSignatureAsNumber	Blinde Signatur S als Zahl
BlindSignatureAsNumberIn	Blinde Signatur S als Zahl
BlindSignatureAsNumberInTooltip	Eingabe der blinden Signatur als Zahl.
BlindSignatureAsNumberTooltip	Ausgabe der blinden Signatur als Zahl.
BlindSignatureGeneratorCaption	Blinde-Signatur-Generator
BlindSignatureGeneratorTooltip	Erstellt eine blinde Signatur für eine Nachricht.
BlindSignatureIn	Blinde Signatur S
BlindSignatureInTooltip	Eingabe der blinden Signatur.
BlindSignaturePaillier	Blinde Signatur S für Paillier
BlindSignaturePaillierIn	Blinde Signatur S für Paillier
BlindSignaturePaillierInTooltip	Eingabe der blinden Signatur für Paillier.
BlindSignaturePaillierTooltip	Ausgabe der blinden Signatur für Paillier.
BlindSignatureTooltip	Ausgabe der blinden Signatur.
BlindSignatureVerifierCaption	Blinde-Signatur-Verifizierer
BlindSignatureVerifierTooltip	Verifiziert eine blinde Signatur
BlindSigningSecurity	Blindes Signieren Sicherheitsstufe
BlindSigningSecurityTooltip	Wählen Sie die Sicherheitsstufe gegen Blind-Signing-Angriffe.
Debug	Debug
DebugTooltip	Ausgabe aller Zwischenergebnisse für Debugging.
DecryptedBlindSignature	Entschlüsselte blinde Signatur S
DecryptedBlindSignatureAsNumber	Entschlüsselte blinde Signatur S als Zahl
DecryptedBlindSignatureAsNumberTooltip	Ausgabe der entschlüsselten blinden Signatur als Zahl
DecryptedBlindSignatureTooltip	Ausgabe der entschlüsselten blinden Signatur
Hash_h_M__is__	Hash h(M) ist:
HashAlgorithm	Hash-Verfahren
HashAlgorithmTooltip	Wähle das zu verwendende Hash Verfahren.
Message_M_is_	Nachricht M ist:
MessageM	Nachricht M
MessageMTooltip	Eingabe der zu signierenden Nachricht.
MessageNumber	Nachricht M als Zahl
MessageNumberTooltip	Eingabe der Nachricht M als Zahl
ModuloN	Modulus N

Abbildung 4.10: Externalisierte Strings in der deutschen Ressourcen-Datei

Ressource-Datei hinzugefügt, die für jeden verwendeten String einen Eintrag enthält. Auf diese Weise können Bezeichnungen von Komponenten-Eingängen und -Ausgängen sowie die dazugehörigen Beschreibungen externalisiert werden. Die deutsche Spezifikation der Ressourcen-Datei erfolgt durch den Dateinamenszusatz .de.resx, welche automatisch von CT2 als Sprachpaket erkannt und geladen wird, wenn CT2 auf Deutsch eingestellt wurde.

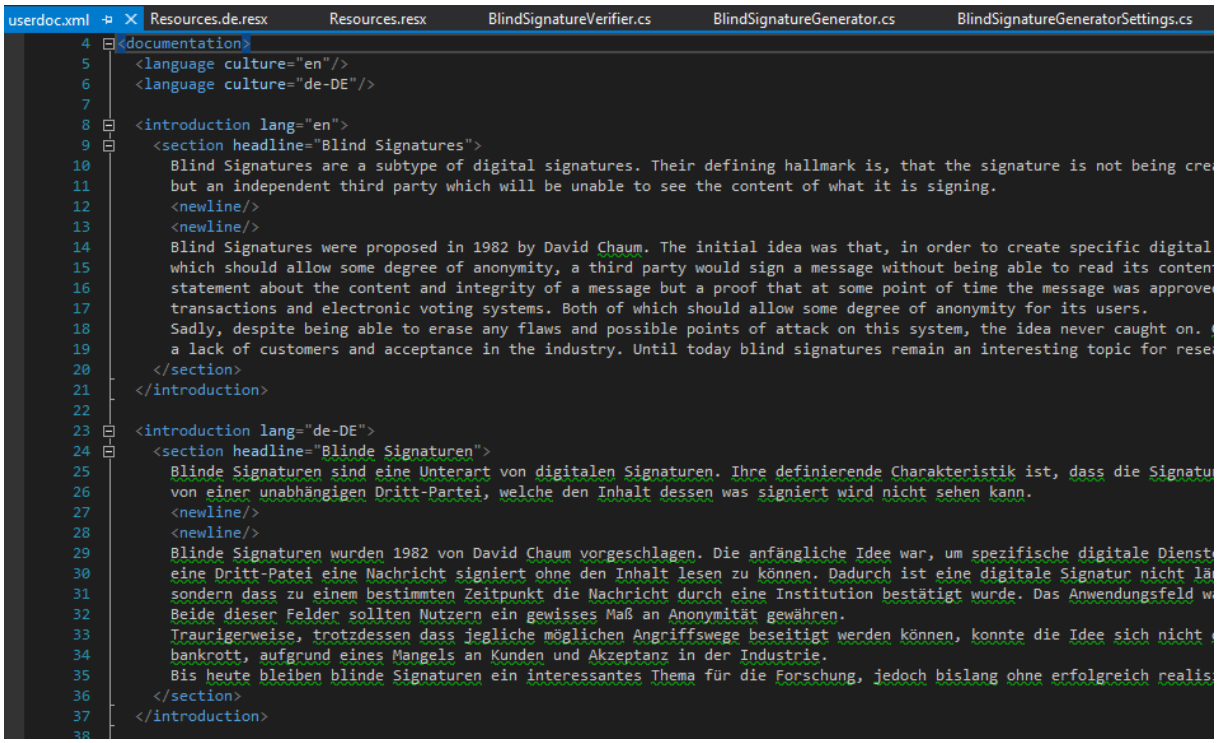


Abbildung 4.11: Code der XML Datei, aus der die Online-Hilfe generiert wird

Die Online-Hilfe der Komponenten wird durch eine lokale xml-Datei bereitgestellt. Diese ist in Abb. 4.11 zu sehen. In dieser Datei werden für die englische und deutsche Sprache die Thematik der blinden Signatur, die implementierten Komponenten und ihre Nutzungsweise kurz beschrieben. Auf diese Weise kann der Nutzer bei Bedarf eine ausführlichere Erklärung der Thematik, der Ein- und Ausgaben, sowie des Zwecks der Komponente erhalten.

CT2 ist auf Mehrsprachigkeit ausgelegt. Die Erstellung einer weiteren Sprache, wie bspw. russisch, kann leicht automatisiert werden.

4.5 Template zur Demonstration blinder Signaturen

Zu jeder neuen Komponente wird in CT2 mindestens ein Template erstellt, das die Verwendung beispielhaft zeigt und für den Nutzer ohne weiteres ausführbar ist. Hierfür wurde als Beispiel der Ablauf der Kommunikation mit 3 Parteien unter Nutzung von blinden Signaturen gewählt.

Das Template soll dem Nutzer ermöglichen einen Nachrichtentext einzugeben. Für diesen erhält er eine blinde Signatur von einer unabhängigen Instanz und kann die Signatur von einem Kommunikationspartner verifizieren lassen.

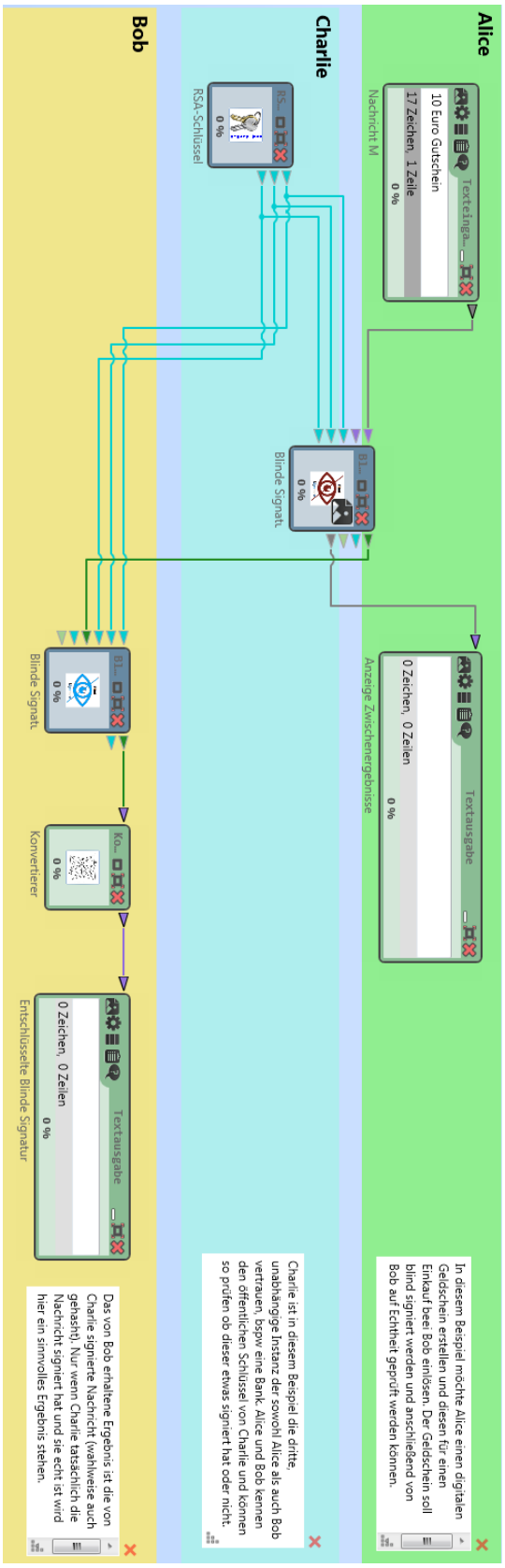


Abbildung 4.12: CT2-Template zur Erzeugung einer blinden Signatur mit 3 Parteien

Im Template (siehe Abb. 4.12) werden drei farblich unterscheidbare Parteien dargestellt. Diese werden als Alice, Bob und Charlie bezeichnet. Alice erstellt in dem Beispiel die Nachricht **10 Euro Gutschein**. Der von Charlie bereitgestellte RSA-Schlüsselgenerator wird genutzt, um eine blinde Signatur zu erstellen. Alice verwendet die erstellte blinde Signatur, um mit Bob zu kommunizieren. Anschließend kann Bob mit den RSA-Schlüsseln von Charlie die Signatur verifizieren. Auf diese Weise erhält Bob die Nachricht **10 Euro Gutschein** und weiß, dass diese Nachricht von Charlie signiert wurde. Nur Charlie besitzt den privaten Schlüssel, der zur Erstellung der Signatur notwendig war. Dies stellt das ursprünglich für blinde Signaturen vorgesehene grundlegende Protokoll dar.

5 Anwendungen blinder Signaturen und Abwehrmöglichkeiten gegen Angriffe

In diesem Kapitel sollen reale Anwendungsfälle von von blinden Signaturen und sowie Lösungen gegen eine konkrete Angriffsmöglichkeit genauer betrachtet werden.

5.1 Anwendungen

Blinde Signaturen können überall Einsatz finden, wo Anonymität wichtig ist. Dies ist derzeit vor allem bei digitalen Finanzmitteln sowie digitalen Wahlsystemen der Fall. David Chaum hat in seiner ursprünglichen Idee für blinde Signaturen den Einsatz bei elektronischen Finanztransaktionen als Hauptbeispiel genommen. Hierbei plante er, dass die Signatur durch eine Bank auf einen Betrag geleistet wird und somit ein digitaler Geldwertschein entsteht. Die Probleme dieser Idee wie Double-Spending und das Signieren eines falschen Betrags wurden in Kapitel 3.4 samt Lösungen thematisiert. Anwendung fand seine Idee bei DigiCash Inc, einem Unternehmen, das er 1989 gründete. Die erste Zahlung unter Nutzung des Systems fand 1994 statt. Allerdings meldete DigiCash 1998 Bankrott an und wurde verkauft. [29]

Weitere Möglichkeiten für anonyme Finanztransaktionen sind PrePaid-Karten und Kryptowährungen. Allerdings verwenden diese Systeme keine blinden Signaturen, sondern realisieren Anonymität auf andere Weise, wenn auch das Ziel gleich ist.

Bei digitalen Wahlsysteme ist es notwendig, dass die Identität eines Wählers in Verbindung zur abgegebenen Stimme anonym bleibt und gleichzeitig sichergestellt wird, dass seine Stimmberechtigung festgestellt, dass er nicht doppelt abstimmt und dass seine Stimme korrekt gezählt wird und unverfälscht bleibt. Die Implementierung ist hier analog zu der Realisierung mit Finanztransaktionen, da lediglich der Wert einer Nachricht nur anders interpretiert werden muss.

Eine konkrete Realisierung von elektronischen Wahlsystemen unter Nutzung von

blinden Signaturen lässt sich in „Secure E-Voting With Blind Signature“ [15] finden. Hier wurde ein Wahlsystem unter Nutzung von Java-Socket-Technologie und von BouncyCastle (einer etablierten Sammlung von kryptographischen Open-Source-Programmierschnittstellen, die auch in CT2 benutzt wird) als Prototyp umgesetzt.

Allerdings ist trotz zahlreicher elektronischer Wahlsysteme der bisherige Einsatz recht begrenzt. Ich konnte keinen Einsatz eines digitalen Wahlsystems, das blinde Signaturen nutzt, finden. Man kann also feststellen, dass sich das Konzept bislang in der Praxis nicht durchsetzen konnte.

5.2 Angriffe und Lösungsmöglichkeiten

In diesem Abschnitt werden die zwei bekannten Angriffsmöglichkeiten gegen blinde Signaturen beschrieben und mögliche Lösungen diskutiert. Für den Angriff in Kapitel 5.2.1 wird auch eine Lösung in der erstellten Komponente implementiert.

5.2.1 Blind-Signing-Angriff

Das Problem des in Kapitel 3.4 beschriebenen Blind Signing kann folgendermaßen gelöst werden: **A** wird gezwungen, eine Anzahl **T** von geblendeten Nachrichten **M*** von der Originalnachricht **M** zu erstellen. **C** wählt zufällig aus der Menge der geblendeten Nachrichten eine aus und verlangt, dass A alle Nachrichten außer der gewählten entblendet, indem **A** den Faktor **k** preisgibt. Wenn **C** nun bestätigen kann, dass **T-1** Nachrichten gleich sind, so kann mit einer Wahrscheinlichkeit von **1/T** davon ausgegangen werden, dass die zufällig ausgewählte Nachricht ebenfalls korrekt ist und den gleichen Inhalt hat, ohne diesen direkt gesehen zu haben. [6, 11]

Die bislang implementierte CT2-Komponente zur Generierung von blinden Signaturen wird daher um zwei Hilfsmethoden erweitert, jeweils eine für die beiden Signier-Verfahren RSA und Paillier. Abb. 5.1 zeigt einen Ausschnitt einer davon. In diesen Methoden wird eine vom Nutzer festgelegte Menge von geblendeten Nachrichten mit zufälligen Blendfaktoren erstellt. Anschließend wird eine der generierten Nachrichten sowie ihr Blendfaktor ausgewählt und alle übrigen entblendet. Hierzu signiert die Komponente die Nachrichten gemäß des zu verwendenden Signier-Verfahrens und entblendet sie, und kann anschließend den erhaltenen Wert mit der Signatur der öffentlich einsehbaren Nachricht vergleichen. Wird festgestellt, dass die Werte nicht übereinstimmen ist

```
//A creates a set of blind messages
for (int i = 0; i < securitylevel; i++)
{
    blindingfactor = Randomnumber();
    factors[i] = blindingfactor;
    blindedmessage = BlindingRSA(new BigInteger(hash));
    messages[i] = blindedmessage;
}
//C picks one at random.
int chosenOne = random.Next((int)securitylevel);
//C demands the unblinding of all other blind messages and checks if those are equal to what C has been told the message is.
for (int i = 0; i < chosenOne; i++)
{
    blindingfactor = factors[i];
    blindedmessage = messages[i];
    //signing-process with RSA is being carried out
    BigInteger temp = BigInteger.ModPow(blindedmessage, PrivateKey, Modulo);
    checkmessage = UnBlindingRSA(temp);

    checkmessage = BigInteger.ModPow(checkmessage, PublicKey, Modulo);

    if (checkmessage != checkpublicmessage)
    {
        GuiLogMessage("Cheating detected! Generator will immediately stop!", NotificationLevel.Warning);
        cheats = true;
        break;
    }
}
for (int i = (chosenOne + 1); i < securitylevel; i++)
{
    blindingfactor = factors[i];
    blindedmessage = messages[i];
    //signing-process with RSA is being carried out
    BigInteger temp = BigInteger.ModPow(blindedmessage, PrivateKey, Modulo);
    checkmessage = UnBlindingRSA(temp);

    checkmessage = BigInteger.ModPow(checkmessage, PublicKey, Modulo);

    if (checkmessage != checkpublicmessage)
    {
        GuiLogMessage("Cheating detected! Generator will immediately stop!", NotificationLevel.Warning);
        cheats = true;
        break;
```

Abbildung 5.1: Ausschnitt aus der Hilfsmethode zur Abwehr eines Blind-Signing-Angriffs für RSA

ein Betrugsversuch erkannt und durch das Werfen einer Exception wird die Komponente angehalten. Auf diese Weise kann ein Nutzer einen Blind-Signing-Angriff simulieren, erkennen und abwehren – mit einer Erfolgs-Wahrscheinlichkeit, die er selbst festlegen kann.

An dieser Stelle muss jedoch angemerkt werden, dass als Folge der Prüfung gegen diesen Angriff die ursprüngliche Idee und der Zweck von blinden Signaturen untergraben wird. Der Signaturleistende kann in Folge der Angriffsabwehr mit einer hohen Wahrscheinlichkeit davon ausgehen, dass die Nachricht ein ihm bekannter Wert ist. Daher ist der Ansatz, dass der Signaturleistende nicht wissen soll, was er signiert beinahe hinfällig, auch wenn keine absolute Gewissheit vorliegt, da die zufällig gewählte Nachricht nie entblendet und gelesen wurde. Somit schützt diese Lösung zwar vor dem Angriff und erlaubt technisch gesehen das Erstellen einer blinden Signatur, doch im Gegenzug

verliert die Signatur ihr hohes Maß an Anonymität.

5.2.2 Double-Spending-Angriff

Das Problem bzw. der Angriff des in Kapitel 3.4 beschriebenen Double Spending kann auf zwei Wegen gelöst werden.

Für **C** kann dieses Problem gelöst werden, indem zu jeder Nachricht **M** von **A** eine Seriennummer **R** hinzugefügt wird. Jedes Mal, wenn eine von **C** signierte Nachricht bei **C** eingereicht wird, entschlüsselt **C** diese und speichert sich die Seriennummer der Nachricht. Auf diese Weise kann **C** mehrfach eingereichte Nachrichten aufspüren. Dies erfordert, dass **C** bei jedem Einlösen von Nachrichten erreichbar sein muss, dass **C** eine möglicherweise sehr große Datenbank für die Seriennummern unterhalten muss und es erhöht den Kommunikationsaufwand, was zu höheren Latenzzeiten führen kann. [11]

Für **B** löst dies das Problem nicht. Hierfür muss das Protokoll geändert werden. Zusätzlich zu der Nachricht **M** muss **A** nun einen Wert **I**, welcher die Identität angibt, und drei gleich lange Zufallszahlenfolgen $\mathbf{a_i}$, $\mathbf{b_i}$, $\mathbf{c_i}$ erzeugen. Der Index **i** ist ein Laufindex über die Anzahl **T** der erzeugten Werte. Nun übermittelt **C** eine vorgegebene Hashfunktion **h** an **A**. Daraufhin bildet **A** die Werte $\mathbf{x_i}$ und $\mathbf{Y_i}$ mit $\mathbf{x_i = h(a_i,b_i)}$ und $\mathbf{y_i = h(a_i + I, c_i)}$.

A sendet eine Anzahl von **T** geblendeten Nachrichten an **C**, jeweils mit dem gleichen Nachrichtentext **M** sowie $\mathbf{x_i}$ und $\mathbf{y_i}$.

Nun kann **C** eine der Nachrichten zufällig auswählen und für den Rest der **T-1** Nachrichten fordern, dass **A** den Faktor, mit welchem die Nachricht geblendet wurde, sowie die Faktoren $\mathbf{a_i}$, $\mathbf{b_i}$, $\mathbf{c_i}$ preisgibt. Dies ermöglicht **C**, die berechneten Hashwerte zu überprüfen. Sollten diese korrekt sein, akzeptiert **C** die zuvor ausgewählte **T**-te Nachricht und signiert diese blind.

Wenn **A** nun später die signierte Nachricht nutzt und an **B** sendet, bspw. bei der Verwendung als Geldschein, so kann **B** nach Prüfung der Signatur von **C** eine **T**-Bit Zahlenfolge $\mathbf{z_i}$ an **A** senden. Hierauf antwortet **A** mit den Werten von $\mathbf{a_i}$, $\mathbf{b_i}$ und $\mathbf{y_i}$ für $\mathbf{z_i = 1}$ oder $\mathbf{(a_i + I)}$, $\mathbf{c_i}$ und $\mathbf{x_i}$ für $\mathbf{z_i = 0}$. Das ermöglicht **B**, die Hashwerte von $\mathbf{x_i}$ und $\mathbf{y_i}$ zu berechnen und somit nachzuprüfen. Hat nun **A** eine Signatur verwendet welche zuvor schon an **B** gesendet wurde, so entsteht ein Konflikt, da durch die zufällige Zahlenfolge $\mathbf{z_i}$ garantiert wird, dass bei wiederholtem Einreichen dieser Signatur **B** sowohl $\mathbf{a_i}$, $\mathbf{b_i}$ und $\mathbf{y_i}$ als auch $\mathbf{(a_i + I)}$, $\mathbf{c_i}$ und $\mathbf{x_i}$ für mindestens ein Bit erhält und somit die Identität **I** von

A berechnen kann. Auf diese Weise kann **B** auch ohne Kontaktieren von **C** bemerken, wenn eine Signatur bei **B** mehrfach an ihn eingereicht wird. [6, 11]

Dieses neue Protokoll verhindert das mehrfache Verwenden von gültigen Signaturen gegenüber einem gleichbleibenden Kommunikationspartner. Wenn A versucht die gleiche Signatur bei einem anderen Kommunikationspartner zu verwenden, wird dieser lokal die doppelte Verwendung nicht bemerken können, sondern kann erst bei Einreichung der Daten bei C den Angriff erkennen.

Als Resultat dieser Änderungen muss ebenso festgehalten werden, dass die Ankunft von Nachrichten bei **B** nicht länger anonym ist. Im Gegenzug kann ein solcher Angriff verhindert werden und Missbrauch verhindert werden.

Das zugehörige Protokoll mit sämtlichen Änderungen für die Analogie eines Bankprozesses sieht wie in Tabelle 5.1 aus.

Schritt	Akteur	Operation
1	A	A wählt I und zufällige Werte von k, R, a_i, b_i, c_i (R ist die Seriennummer)
2	A	A berechnet $x_i = h(a_i, b_i)$; $y_i = h(a_i + I, c_i)$; $M' = (M+R+x_i+y_i) * k$ Hiermit erstellt A die geblendeten Nachrichten
3	A	A sendet N Nachrichten mit M', R, x_i und y_i an C
4	C	C wählt zufällig N-1 Nachrichten und fordert k, a_i, b_i, c_i für diese von A an
5	A	A sendet k, a_i, b_i, c_i für N-1 Nachrichten an C Hiermit gibt A für N-1 Nachrichten die Blendfaktoren preis
6	C	C berechnet M = M' * k-1 ; xi = h(ai, bi) ; yi = h(ai + I, ci) für N-1 Nachrichten; stoppt, wenn falsches Ergebnis vorliegt. Hierdurch entblendet C die N-1 Nachrichten und prüft, dass kein Betrug vorliegt
7	C	C berechnet für die N-te Nachricht $s' = S_C(M')$ und sendet s' zurück an A. C signiert also nur die N-te, noch geblendete Nachricht
8	A	A berechnet s durch $s = s' * k^{-1}$ A entblendet die Signatur
9	A	A sendet s an B
10	B	B prüft s mit $V_C(s)$ B prüft die Gültigkeit der Signatur mit dem öffentlichen Schlüssel von C
11	B	B sendet zufällige N-Bit Zahlenfolge z_i an A B fordert damit für zufällig Indizes die Faktoren an, die indirekt die Identität von A beinhalten
12	A	A sendet für z_i=1 a_i, b_i und y_i und für $z_i = 0$ ($a_i + I$), c_i und x_i an B
13	B	B prüft durch Nachrechnen mit h Werte von xi und y_i; B stoppt, wenn für einen Index i sowohl x_i als auch y_i vorliegen und damit die Identität I von A bekannt wird
14	B	B akzeptiert s und sendet Daten weiter an C. B hat 2 Dinge geprüft: ob die Signatur gültig ist und ob sie zuvor bereits eingereicht wurde, und kann sie daher annehmen
15	C	Prüft s mit $V_C(s)$; speichert R in Datenbank; stoppt, wenn R bereits in Datenbank vorliegt
16	C	Sendet Wert von M an Konto von B. Da R in der Datenbank zuvor nicht vorlag und B ebenfalls die Signatur annahm, ist die Transaktion abgeschlossen

Tabelle 5.1: Protokoll einer blinden Signatur mit den Lösungen gegen Blind-Signing und Double-Spending

5.3 Evaluation

Die Performance (Laufzeit) der beiden blinde Signatur-Komponenten Generator und Verifier wird verglichen mit einem äquivalenten Aufbau zur Erstellung von nicht-blinden Signaturen (sowohl für RSA wie für Paillier). Damit kann man erkennen, ob durch den zusätzlichen Rechenaufwand die Signaturerstellung signifikant länger benötigt.

Die Berechnungen werden in CT2 durchgeführt, welches auf einem Windows7 PC mit 12 Intel i7-5820K CPUs mit 3,30 GHz ausgeführt wird.

Hierfür werden zwei Sets von Parametern gewählt:

Set 1 nutzt kleine Zahlen als Beispielwerte zwecks Verständlichkeit. Diese sind für RSA: Modulus **n = 299**, öffentlicher Exponent **e = 23** und privater Schlüssel **d = 23**. Für Paillier sind die Werte wie in Kapitel 2.5: Modulus **n = 667**, öffentlicher Schlüssel **g = 668** und privater Schlüssel **lambda = 308**.

Die erstellten Signaturen sind für RSA s=284, für Paillier s1=17 und s2=487.

Set 2 nutzt größere Zahlen von einer 1024 Bit-Länge für Praxis-nahe Werte.[1]

Die Nachricht für beide Sets ist **M = 123**, derselbe Wert wird zur Prüfung gegen einen Blind-Signing-Angriff verwendet. Die Sicherheitsstufe, das heißt die Zahl an Iterationen zur Prüfung gegen einen solchen Angriff, wird variiert auf 100, 1.000, 10.000 und 100.000.

Für diese Werte ergeben sich die Daten in Tabelle 5.2 (alle Werte in Spalte 2-5 in Millisekunden):

Einstellung	RSA-Parameterset 1	Paillier-Parameterset 1	RSA-Parameterset 2	Paillier-Parameterset 2
nicht-blinde Signatur in CT2	1.055	nicht verfügbar	1.030	nicht verfügbar
kein Angriff	1.059	1.054	1.027	1.037
Sicherheitsstufe 100	1.026	1.058	11.038	58.074
Sicherheitsstufe 1.000	2.130	1.103	100.045	573.066
Sicherheitsstufe 10.000	13.851	8.388	990.117	5.915.704
Sicherheitsstufe 100.000	132.166	85.483	10.493.952	nicht verfügbar[2]

Tabelle 5.2: Ergebnisse der Laufzeit bei Berechnung mit Beispielwerten

[1] Siehe Anhang Kapitel 5.3.1

[2] Abbruch der Berechnung nach über 14h

Die Ergebnisse können nun verglichen werden mit der Laufzeit für die Berechnung einer nicht-blinden RSA-Signatur. Für Paillier gibt es in CT2 bislang keine Komponente zur Signaturerstellung, daher kann nur die Laufzeit für die nicht-blinde RSA-Signatur als Ausgangswert genommen werden.

Der Vergleich von Parameterset 1 ergibt, dass, ohne Prüfung auf einen Blind-Signing-Angriff, ein vernachlässigbarer Laufzeit-Unterschied von 4-5 ms, also unter 0,004%, vorliegt. Wenn jedoch auf einen Angriff geprüft wird, steigt die Laufzeit exponentiell. Für eine Wahrscheinlichkeit von 1 zu 100.000, dass ein Angriff trotz Prüfung erfolgreich ist, steigt die Laufzeit bei RSA um das 125-fache an. Bei Paillier steigert sich die Laufzeit um das 82-fache. Dieser Anstieg ist dadurch zu erklären, dass bei der Prüfung auf den Angriff für die gewählte Sicherheitsstufe entsprechend viele Nachrichten erstellt und geblendet werden und für alle bis auf eine Nachricht wiederum auf Gleichheit geprüft wird. Das Blenden und Entblenden von entsprechend vielen Nachrichten ist daher die Ursache für diese Verzögerung.

Bei Parameterset 2 mit Parametern, die einem praktischen Einsatz weit näher kommen, ist der Anstieg der Laufzeit mit steigender Sicherheitsstufe weit stärker. Die letzte RSA Berechnung benötigte 2 Stunden, 54 Minuten und 54 Sekunden und 952 Millisekunden. Dies ist ein Anstieg um das 10188-fache.

Es lässt sich somit sagen, dass die Komponente „Blinde-Signatur-Generator“ Laufzeiten besitzt, die ohne weitere Sicherheitsvorkehrungen gleichzusetzen sind mit nicht-blinden Signaturerstellungen. Der bloße Einsatz von blinden Signaturen zieht also keinen signifikanten zeitlichen Nachteil mit sich. Wenn jedoch ein Nutzer entsprechende Sicherheit gewinnen will, so ist ein erheblicher Zeitverlust unvermeidlich.

5.3.1 Anhang

Die Werte für das Parameterset 2 sind:

p=1443*C*8888*AF*24*B*764*B*1530*C*7*C*15283*BA*39*E*3*E*9719*A*23852*CEC*315*FC*4*C*0904792*C*7*E*
55*E*98505182130*DAEA*119*FB*284*BEC*710*B*4876*E*5407*B*9*C*7*F*8*D*5*CF*5144*BAD*2*B*43170254
96*C*8*F*1*E*33*E*687302749*AEB*0*F*73*F*977*FC*26*B*7*B*7485*DB*5*B*5*AAC*272*A*3*EBA*581*F*27*E*5*DE*
*BA*89*DB*38*A*7607*D*011*E*34506*E*15*DCBCD*96011*DBF*89*D*5335584*A*919

q=77*A*215*A*449845681*D*72990823526*B*8*CAB*6*C*6*FCA*82*E*864*ED*7*EB*084*D*7*B*22*D*3061*CA*5
7*EDAC*17000*E*2*EF*6*D*907*CE*24*B*089*B*1*C*7*A*47*D*20*FC*6*C*5*CF*777*E*329*B*47*C*5268892*B*4*EC*9
6*CA*386844*EBC*915*A*61*F*2*A*9402*ACD*1*E*69*F*92*AF*9*B*4*DA*7*A*11*CB*0783*EF*427*A*82509*A*5*D*07
3*E*76*EC*4*DF*84385*E*9*A*85*CF*6*D*54*F*0*B*8*ECAF*50*ED*864*D*47*AF*4*D*1*C*90*FBBF*

n=97856*D*2*EFB*0074203*C*048*ABD*7*B*054*D*3*B*139*EC*5*BCA*17*BCF*0*DFD*44359*BFFB*42*F*8*F*
455*B*222*C*2448637*B*9*FA*4*A*8121*ECF*3289*CBF*028496*A*340*F*4*CC*528*B*0346*CC*3*F*707*AF*7*C*5
*DA*0*C*426*DC*07420*EEDA*7*CCE*6*B*462*F*9*EEB*8832*F*46*DE*18*DE*4*E*2914*F*1*DC*8*A*18*A*79*EED*
857*E*2*CAF*1*A*7*E*086*C*7*E*9*BDBD*46*DBDDB*7630*A*0*DBF*0*DC*08*D*0*F*6850*EBD*51*A*747*CA*1*C*8
1*D*80952201335*F*7*B*4328*AB*8*D*2*F*140*E*305885*E*8300*BCCDB*89*B*49*C*63*B*11*EE*86*CA*194*BF*
9*AB*0*B*2*E*87*E*1*E*9*AB*759*ED*13*DF*50527*DD*76135609*B*0337893*CDE*3*A*215*FE*8149*FBA*9*BF*
3*A*7775*B*9524*ECD*646*E*411*F*44*F*1586*C*2*F*044*C*0*D*91*CF*1*DA*5*CF*4929440*FE*7*FC*1*CDE*29
*EC*5*CCADB*57*EFD*0*C*48467*D*8041*A*74*F*7*B*6987*F*6*C*8*B*9*D*5*ACA*7

e=5

d=5*AE*9*A*7*E*8*FD*0045*ACF*0*CF*8671*B*0366189*D*89276*A*460*E*3*E*2*A*1*FE*5*C*202*A*6638*E*955*F*
669*E*14*DAF*5*EA*216*F*962*CB*3*E*127*C*5185*E*0*C*34*B*5*F*3*FB*8*D*5*FADCB*20352*A*7*A*8*C*76*B*02*E*
4382*D*427*DB*7378*E*08*F*4*FE*47*BD*9*F*6*E*95*F*5*A*1*E*84*F*90*EBA*8856218*A*62*AB*7860*ECAF*8*F*4*
E*9*B*21*ACF*764*B*9*EA*77*F*2*A*50*B*2*A*83*EB*6*E*08393*B*72*A*1*D*9*EE*3*C*71*CA*271975*F*1*F*40*EF*6*C*5
6*C*5*EB*2*B*8*F*92*BB*72*E*02*DC*3825724*B*40*BCB*73*F*4*D*1*F*8*B*822*C*4*A*55739*A*9*DB*5*B*85*E*4*C*4186
9*BBAFF*39*AE*32*B*711*BC*18542047177*D*73*ADA*1*B*6*A*030031331*E*418*D*0*FC*2*D*16*B*8*D*6492*C*
*C*8*C*1788371*F*1*ECC*5511*E*0*CF*2*B*8*A*5335*D*28*AB*62*C*3788*A*6*D*44*D*1*FECFCEC*26*D*8007077
8912*CF*12*C*9*C*5*AAC*3*DAB*0*FA*6*A*08137*A*0*AAF*82*B*0*CD*17*D*

g=97856*D*2*EFB*0074203*C*048*ABD*7*B*054*D*3*B*139*EC*5*BCA*17*BCF*0*DFD*44359*BFFB*42*F*8*F*
455*B*222*C*2448637*B*9*FA*4*A*8121*ECF*3289*CBF*028496*A*340*F*4*CC*528*B*0346*CC*3*F*707*AF*7*C*5
*DA*0*C*426*DC*07420*EEDA*7*CCE*6*B*462*F*9*EEB*8832*F*46*DE*18*DE*4*E*2914*F*1*DC*8*A*18*A*79*EED*
857*E*2*CAF*1*A*7*E*086*C*7*E*9*BDBD*46*DBDDB*7630*A*0*DBF*0*DC*08*D*0*F*6850*EBD*51*A*747*CA*1*C*8
1*D*80952201335*F*7*B*4328*AB*8*D*2*F*140*E*305885*E*8300*BCCDB*89*B*49*C*63*B*11*EE*86*CA*194*BF*
9*AB*0*B*2*E*87*E*1*E*9*AB*759*ED*13*DF*50527*DD*76135609*B*0337893*CDE*3*A*215*FE*8149*FBA*9*BF*
3*A*7775*B*9524*ECD*646*E*411*F*44*F*1586*C*2*F*044*C*0*D*91*CF*1*DA*5*CF*4929440*FE*7*FC*1*CDE*29

EC5CCADB57EFD0C48467D8041A74F7B6987F6C8B9D5ACA8

lambda=4BC2B6977D803A101E02455EBD82A69D89CF62DE50BDE786FEA21ACDFF
DA17C7A2AD9116122431BDCFD254090F679944E5F81424B51A07A66294581A3661FB8
3D7BE2ED062136E03A10776D3E6735A317CF75C4197A36F0C6F27148A78EE450C53C
F76C2BF16578D3F04363F4DEDEA36DEEDBB18506DF86E04687B42875EA8CF44B61
CDA484F994EF7A4F18A657B7841F34940B47EE0A159A499724E89C8B00D8C219A3EA
369AD71BD4B0112A43397214461AE5E8E88B10B16D857D57E5543E14AE277AE844B29
24FFCA139C2DEF4454F18EE601F99DF0022F7397CF839C8B064045457D6F75B4005
DB8F23A573A52CF63A3363E260306BAE5B392423E003E8

Signatur für RSA:

s=4DBD5B70EA180F5E5CD7A77D25C0EA66906825B349A393150A7B3EEDF5F8EA33
2DFB661FB1417DBEC318DD5FEB8DBFD5A1DBFF2F1E357F75369B90A13A0BE9055
99C830F9E7218846803CC648307D2DF6AAF2079050BB331B756F6C8939BC954336B1B
4ECF203A89F0B049C2523E8BF4E4E99A9ABBFBFFF4BC55C861EDA61B224114BC83
977C30961D07782B7E122A0DC2CB58F136C6250A5B3FF849FF9276FFAEFD0E20CC2
E42BFE2B43685DC6B4B3282A2FDE20B5803D76799CFCEC3B8E477B5D01A26157853
A88F616E7CCAFA8A1B12128F2EF726B09C66AB870BCAE4ED43AFC463EB9759721F8
3E514B371B8EF8D4EE165BC698972C6BBA84AEF9C58EDC

Signaturen für Paillier:

s1=25A05ECC6C814F20EC9B5FC563BD5AE94659AD4B336DEAF07E49536826E1991C
8075697F9EDB349272BA694E8E4E02505630C5E9DC40EDDDE03B78C953B99C875318
526EA90834FBD98A7255C7B0D17F3F459704C64E8C87ADBF2B1B7CD63B8ABF35AB7
2AA1D72DDA0ABFE9E17DECBC2C80F8D8C1BAF26D4F6EBF3A892BE1B0FB743A96
2B54478AA2463161C4D0616205868760E1C7CFC76C1AFADBDA682F5DA78F2D39791E
C531D5E587ACD7E8FB82E07B8DD499E88502589696109B134E77B767E74BEC2925F4
7CDBFDC1BBEF0D104B1F0EA04D4CAEA10B528000A504BA91FC9EF05A93C9C60D5
48C2B67778994A273299C9F035D9BA5D1882389D8C8483F

s2=5CA17F69B7F4512B9C5F782F5EF3603D465C972BAB8A314D7DAFC204D0F9850B
054C2660F442D066F58CEEBF91F1368561AC6C89AB1510E3D24C60E751A5AF30A416
78B3A84E6950E0284998CECCFC234F364519D334BA35C097BD87C2A7DB67E90B1577
029A22984E3D54F89FC4BBD6BD95410D07BECE1AC9470668A02E2F235B2D16051CA
7F2421EF02E1C10AEB7648748DDCB0FA0B3A86EBA6D4C3B4A59584E20CA1CEFBF9
301350A28244AA068FEDA013A74C53A14BF0CED78E2D0FFE591777F91D722B643577
1A1BACCD7C8AC869A8D16829671F2E8D738044CF6039C41EFFD2FB95D56CB99B3B1
2FF55385094EC99286409D2291019CD9AA96660B0AE

6 Ausblick und Zusammenfassung

6.1 Zusammenfassung

In Kapitel 1 „Einleitung“ wurden die Ziele der Arbeit definiert. Im darauffolgenden Grundlagenkapitel wurde das Verständnis für die zugrundeliegende Thematik geschaffen und die notwendigen Begriffe eingeführt. Das eigentliche Thema der Arbeit, die blinden Signaturen, wurde im dritten Kapitel ausgehend von der anfänglichen Idee von David Chaum erklärt. Damit wurden die in Unterkapitel 1.3 definierten Ziele 1 und 2 erfüllt.

Ziel 3, die Beschreibung blinder Signaturen anhand des RSA- und des Paillier-Verfahrens, wurde in Kapitel 3.2 erreicht. Vorteile und Nachteile sowie Lösungsvorschläge für die aufgetretenen Probleme bei blinden Signaturen wurden aufgezeigt und die bekannten Arten zur Unterteilung dargestellt.

Kapitel 4 erläuterte die Umsetzung von blinden Signaturen in CrypTool 2 (Ziel 4 der Arbeit). Hierzu wurde eine Komponente in das Programm hinzugefügt, welche für zwei verschiedene Verschlüsselungsverfahren eine blinde Signatur korrekt erzeugt. Eine weitere Komponente zur Verifizierung der erzeugten Signaturen wurde ebenfalls erstellt. Auch eine Vorlage zur beispielhaften Nutzung der neuen Komponenten wurde erstellt und getestet. Der Mehrwert der Arbeit besteht in den erstellten Komponenten, die bisher in CT2 fehlende Funktionen umsetzen. Nach Durchlauf der Qualitätssicherung (automatische Tests, umfangreichere Mehrsprachigkeit) werden diese Komponenten im Herbst 2019 auch in die kommende Releaseversion von CT2 aufgenommen werden. Die Funktionalität einer blinden Signatur hätte man mit den bisherigen Komponenten nur in einem sehr komplexen Template (ca. 30 Komponenten) und ohne die Flexibilität der Parameter realisieren können. Mit den beiden neuen Komponenten ist dieselbe Funktionalität weit übersichtlicher realisiert worden. Im Kapitel 5 „Anwendungen blinder Signaturen und Abwehrmöglichkeiten gegen Angriffe“ wurden die Anwendungsgebiete dieser Art digitaler Signaturen untersucht und hinsichtlich ihrer Umsetzbarkeit beschrieben. Ebenso wurden Angriffe auf blinde Signaturen erläutert, sowie gegen einen Angriff eine Abwehr implementiert.

Im Rahmen der Evaluation in Kapitel 5.3 konnte zuletzt gezeigt werden, dass – bei alltäglicher Verwendung und ohne zusätzlich hohen Sicherheitsbedarf – blinde Signaturen in ihrer Berechnungslaufzeit keine signifikaten Verzögerung mit sich bringen.

6.2 Ausblick

Mit der im Rahmen dieser Arbeit entstandenen Komponenten für blinde Signaturen wurde ein grundlegendes Modul für blindes Signieren innerhalb von CT2 geschaffen. Diese spezielle Art von Signaturverfahren kann jedoch nicht nur mit RSA und Paillier zum Einsatz kommen, sondern auch mit anderen kryptographischen Verfahren. Als weitere Signatur-Verfahren für blinde Signaturen bieten sich zukünftig homomorphe Verschlüsselungsverfahren wie das Goldwasser-Micali-Kryptosystem, das Benaloh-Kryptosystem und das Okamoto-Uchiyama-Kryptosystem an. Außerdem bieten sich auch reine Signaturverfahren wie das Nyberg-Rueppel-Verfahren oder DSA an.

Erweitern könnte man die Blinde-Signatur-Generator Komponente in CT2, damit sie auch gegen Double Spending resistent ist. Zusätzlich könnte man die Komponente zukünftig so erweitern, dass ein Nutzer zwischen den in Kapitel 3.5 erklärten Arten von blinden Signaturen wählen kann.

7 Abbildungsverzeichnis

8 Tabellenverzeichnis

9 Literatur

[1] CrypTool 2. *IPlugin call flow*. 2011. URL: https://www.cryptool.org/trac/CrypTool2/wiki/IPluginHints (besucht am 19. 07. 2019).

[2] CrypTool 2. *Plugin Template*. URL: https://www.cryptool.org/trac/CrypTool2/browser/trunk/Documentation/CrypPluginTemplate/CrypTool%202%20Plugin.zip (besucht am 04. 09. 2019).

[3] Ashutosh Ahelleya. *Blinding Attack on RSA Digital Signatures*. 2017. URL: https://masterpessimistaa.wordpress.com/2017/07/10/blinding-attack-on-rsa-digital-signatures/ (besucht am 10. 07. 2019).

[4] Tueno Anselme. „Zwei Anwendungen des Paillier-Kryptosystems: Blinde Signatur und Three-Pass-Protocol“. In: *arXiv preprint arXiv:1206.1078* (2012).

[5] Bundesnetzagentur. *Bekanntmachung zur elektronischen Signatur nach dem Signaturgesetz und der Signaturverordnung*. 2013. URL: https://www.bundesnetzagentur.de/SharedDocs/Downloads/DE/Sachgebiete/QES/Veroeffentlichungen/Algorithmen/2013Algorithmenkatalog.pdf?__blob=publicationFile&v=2 (besucht am 10. 07. 2019).

[6] David Chaum. „Blind signatures for untraceable payments“. In: *Advances in cryptology*. Springer. 1983, S. 199–203.

[7] David Chaum. „Zero-knowledge undeniable signatures“. In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1990, S. 458–464.

[8] David Chaum, Amos Fiat und Moni Naor. „Untraceable electronic cash“. In: *Conference on the Theory and Application of Cryptography*. Springer. 1988, S. 319–327.

[9] David Chaum und Eugène Van Heyst. „Group signatures“. In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1991, S. 257–265.

[10] Matthew Franklin und Moti Yung. „The blinding of weak signatures“. In: *Workshop on the Theory and Application of of Cryptographic Techniques*. Springer. 1994, S. 67–76.

[11] Shafi Goldwasser und Mihir Bellare. „Lecture notes on cryptography“. In: *Summer course “Cryptography and computer security” at MIT* (1996).

[12] Alexander Hirsch. *Implementierung und Visualisierung von Format-erhaltenden Verschlüsselungsverfahren in CrypTool 2*. Universität Kassel. 2018.

[13] Patrick Horster und Holger Petersen. *Classification of Blind Signature Schemes and Examples of Hidden and Weak Blind Signatures*. 1994.

[14] Thomas Claudius Huber. *Windows Presentation Foundation: Das umfassende Handbuch*. Rheinwerk Computing. 2017.

[15] Subariah Ibrahim, Maznah Kamat, Mazleena Salleh und Shah Rizan Abdul Aziz. „Secure E-voting with blind signature“. In: *4th National Conference of Telecommunication Technology, 2003. NCTT 2003 Proceedings.* IEEE. 2003, S. 193–197.

[16] GEMPLUS Card International. *Subscriber digital signatures require the use of smart cards*. 2002. URL: https://www.3gpp.org/ftp/TSG_SA/WG3_Security/TSGS3_26_Oxford/Docs/PDF/S3-020625.pdf (besucht am 10. 07. 2019).

[17] Ari Juels, Michael Luby und Rafail Ostrovsky. „Security of blind digital signatures“. In: *Annual International Cryptology Conference*. Springer. 1997, S. 150–164.

[18] Aggelos Kiayias, Yiannis Tsiounis und Moti Yung. „Traceable signatures“. In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2004, S. 571–589.

[19] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen Lenstra, Emmanuel Thomé u. a. *Factorization of a 768-bit RSA modulus*. Cryptology ePrint Archive, Report 2010/006. https://eprint.iacr.org/2010/006. 2010.

[20] Alfred J Menezes, Paul C Van Oorschot und Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[21] Rolf Oppliger. *Contemporary Cryptography*. Artech House, 2005.

[22] Pascal Paillier. „Public-key cryptosystems based on composite degree residuosity classes“. In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 1999, S. 223–238.

[23] Andreas Pfitzmann und Marit Hansen. „A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management“. In: (2010).

[24] Europäisches Parlament und Rat. *VERORDNUNG (EU) Nr. 910/2014 DES EUROPÄISCHEN PARLAMENTS UND DES RATES*. 2014. URL: https://eur-lex.europa.eu/legal-content/DE/TXT/PDF/?uri=CELEX:32014R0910&from=DE (besucht am 10. 07. 2019).

[25] Bundesamt für Sicherheit in der Informationstechnik. *BSI-TR-02102-1*. 2019. URL: https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR02102/BSI-TR-02102.html (besucht am 02. 09. 2019).

[26] Thomas Theis. *Einstieg in C# mit Visual Studio 2017*. Rheinwerk Computing. 2017.

[27] Olaf Versteeg. *Visualisierung und Implementierung von Betriebsmodi von Blockchiffren für CrypTool 2*. Universität Kassel. 2018.

[28] Oliver Vornberger. *ecash: Das Geld auf der Festplatte 5. Blinde Signaturen*. 2001. URL: http://www-lehre.inf.uos.de/vsin/vsin04/ecash/5_Blinde_Signaturen.html (besucht am 10. 07. 2019).

[29] Wikipedia. *DigiCash*. 2019. URL: https://en.wikipedia.org/wiki/DigiCash (besucht am 10. 07. 2019).

[30] Jianying Zhou und Robert Deng. „On the validity of digital signatures“. In: *ACM SIGCOMM Computer Communication Review* 30.2 (2000), S. 29–34.