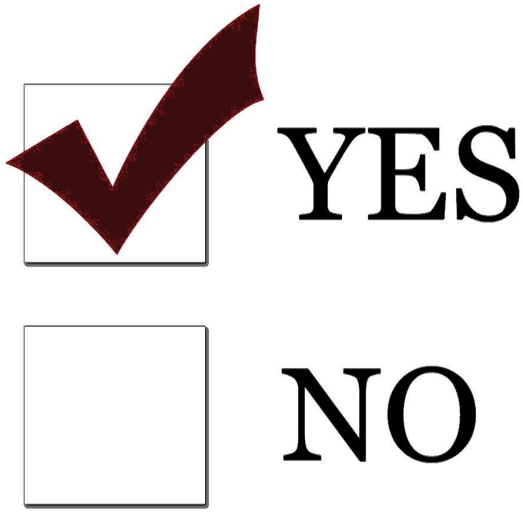


Projekt 2: E-Voting

Abschlussbericht Modul "BTI-7302 Projekt 2"

Dokumentation der Erkenntnisse und Vorarbeit zur Bachelorthesis

Studiengang:	BSc Informatik
Autoren:	Bürk Timo, Nellen Sebastian
Betreuer:	Dr. Haenni Rolf, Dr. Koenig Reto
Datum:	03.07.2016

Versionen

Version	Datum	Status	Bemerkungen
0.1	28.06.2016	Entwurf	Aufbau LaTeX-Dokument
0.2	30.06.2016	Entwurf	Einfügen der bestehenden Textblöcke in LaTeX
0.9	01.07.2016	Entwurf	Vorabversion erstellt
1.0	03.07.2016	Definitiv	Abgabeversion erstellt

Inhaltsverzeichnis

1. Einleitung

Dieses Dokument ist ein Bericht über die im BFH-Modul „7302 Projekt 2“ von Timo Bürk und Sebastian Nellen erarbeiteten Erkenntnisse. Im Rahmen des Projekts haben sich die Autoren mit dem Thema „E-Voting with everlasting privacy“ auseinandergesetzt. Als Ausgangslage dienten das Paper „Verifiable Internet Elections with Everlasting Privacy and Minimal Trust“ von Philipp Locher und Dr. Rolf Haenni aus dem Jahr 2015 [1], sowie weitere wissenschaftliche Arbeiten ([2] und [3]), welche an den entsprechenden Stellen vermerkt werden.

Das Dokument betrachtet zuerst im Kapitel 2 die theoretischen Aspekte rund um das Thema E-Voting. In diesem Kapitel wird einerseits das Protokoll von Locher und Haenni besprochen, mögliche Gegner veranschaulicht und die nötige Infrastruktur aufgezeigt. Andererseits werden die mathematischen Konstrukte, wie Commitments und Zero-Knowledge Proof of Knowledge, erörtert und veranschaulicht.

Danach werden im Kapitel 3 die drei Beweise π_1, π_2 und π_3, auf welchen das Protokoll aufbaut, betrachtet und analysiert.

Die beiden folgenden Kapitel 4 und 5 beschäftigen sich mit der praktischen Implementation des Protokolls. Zuerst wird die Theorie anhand der Implementation mit der Unicrypt-Bibliothek veranschaulicht und anschliessend wird mit GWT ein mögliches Framework für die Bachelor-Thesis evaluiert.

Dieser Bericht schliesst mit einem Fazit und gibt einen Ausblick auf die kommende Bachelor-Thesis in diesem Themengebiet.

Im Anhang des Dokuments wurden der Vollständigkeit halber Projektablauf, sowie wichtige Bilder, Notizen und Präsentationen festgehalten.

2. Theoretische Aspekte

2.1. E-Voting

Viele der heute verwendeten E-Voting-Protokolle basieren auf den folgenden Annahmen:

- Es sind vertrauenswürdige Behörden (so genannte *trusted authorities*) im Wahlvorgang beteiligt. Sie unterstützen oder übernehmen zentrale Tätigkeiten wie beispielsweise das Vermischen von Stimmen zur Sicherstellung der Privatsphäre oder das Entschlüsseln von Stimmen.
- Der oder die Gegner haben nur beschränkte Ressourcen. Dies schiebt zwar die Verletzung der Privatsphäre (*privacy breach*) in die Zukunft, verhindert sie aber nicht. Es ist somit bei jenen Protokollen möglich, dass ein zukünftiger Gegner in 30 Jahren genug Rechenleistung hat, um eine heutige Stimme zu identifizieren oder eine Stimme mit einer Person in Verbindung zu bringen.

In den nachfolgenden Kapiteln folgen weitere Details zu den Themen Gegner und Vertrauensannahmen in Bezug auf E-Voting.

Das E-Voting Protokoll von Locher und Haenni basiert auf weniger starken Annahmen, da es nicht von vertrauenswürdigen Behörden oder Rechenleistung eines zukünftigen Gegners abhängt. In einem informations-theoretischen Sinn bietet das Protokoll sogar dauerhafte Privatsphäre der getätigten Stimmen.

Es gibt aber auch bei diesem neuen Ansatz Voraussetzungen und Annahmen, welche in gewissen Fällen nötig sind:

- Wenn die Wahl *fair* sein muss - das heisst, wenn eine Stimme unabhängig abgegeben werden soll ohne zu wissen, wie das aktuelle Wahl-Resultat ist - wird eine vertrauenswürdige Drittpartei benötigt, welche das Resultat bis zum Ende der Wahl geheim hält.
- Damit keine falschen Stimmen erstellt und abgegeben werden können, muss davon ausgegangen werden, dass ein allfälliger Gegner zur Zeit der Abstimmung in seiner Rechenleistung eingeschränkt ist.

Ergänzend zu den Voraussetzungen und Annahmen sollte an dieser Stelle noch eine Einschränkung der Rechenleistung erwähnt werden: Die Generierung der elektronischen Stimmzettel und die abschliessende Kontrolle einer Wahl ist relativ kostspielig in Sachen Rechenkraft. Allerdings spielt dies mit der Leistung heutiger Technologien nur noch bei sehr grossen Wahlen eine Rolle.

2.2. Informeller Ablauf des Protokolls

Um einen Überblick zum Ablauf des vorgestellten Protokolls zu geben, werden anschliessend kurz die verschiedenen Phasen aufgezeigt und beschrieben.
Zu Beginn generiert sich jeder Wähler ein Paar aus privater und öffentlicher Wähleridentität (auch *private* und *public credentials* genannt). Daraufhin sendet der Wähler die öffentliche Wähleridentität über einen authentischen Kanal zur Wahladministration.
Die Wahladministration publiziert daraufhin die Liste der öffentlichen Wähleridentitäten auf der öffentlichen (virtuellen) Wahlanzeige (so genanntes *public bulletin board*). Pro Wähler wird eine öffentliche Wähleridentität auf dem bulletin board publiziert.
Anschliessend generiert jeder Wähler einen elektronischen Stimmzettel und sendet ihn über einen anonymen Kanal zum öffentlichen bulletin board. Ein elektronischer Stimmzettel besteht aus der Stimme selber, ein Commitment (siehe 2.6) zur öffentlichen Wähleridentität (des Wählers) und einer Zusammenstellung von Zero-Knowledge Proofs (siehe 2.8).
Am Ende einer Abstimmungsperiode kann jeder das Wahlresultat aus den publizierten Daten auf dem bulletin board ableiten und überprüfen. Ob das Resultat korrekt ist, kann jeder anhand der Zero-Knowledge Proofs der einzelnen elektronischen Stimmzettel berechnen.

2.3. Gegner

Beim Thema Gegner muss zwischen einem heutigen Gegner (*present day adversary*) und einem möglichen Gegner in der Zukunft (*future adversary*) unterschieden werden. Nachfolgend werden die beiden Begriffe erklärt.

Heutiger Gegner Ein heutiger Gegner handelt vor oder während einer Abstimmung. Seine Ziele sind auf der einen Seite das Brechen der Integrität, in dem der Gegner eine valide Stimme im Namen eines Anderen abgibt. Andererseits versucht der Gegner die Geheimhaltung einer Wahl zu brechen, in dem er versucht, Stimmen mit Wählern in Verbindung zu bringen.
Der heutige Gegner hat nur eine beschränkte Rechenleistung. Er ist *polynomially bounded*[1] und kann weder diskrete Logarithmen noch aktuelle kryptographische Hash-Funktionen brechen.

Zukünftiger Gegner Ein zukünftiger Gegner wird irgendwann in der Zukunft nach einer Wahl aktiv werden und versuchen eine Stimme mit einem Wähler in Verbindung zu bringen.
Im Gegensatz zum heutigen Gegner ist er an keine Rechenleistung gebunden, da man heute nicht mit Gewissheit sagen kann, welche Technologien in der Zukunft möglich sein werden.
Sofern mehrere Abstimmungen mit Hilfe dieses Protokolls und mit derselben Wähleridentität durchgeführt werden, kann ein zukünftiger Gegner die verschiedenen Stimmen auf denselben Wähler zurückführen - auch wenn er nicht weiss, welche Person hinter dieser Identität steckt. Es besteht aber trotzdem ein Risiko, dass ein Wählerprofil erstellt und auf einen Wähler (oder eine Gruppe von Wählern) geschlossen werden kann.

2.4. Infrastruktur

Damit das Protokoll realisiert werden kann, müssen einige Annahmen getroffen werden und eine gewisse Infrastruktur vorhanden sein.
Zwischen dem Wähler und der Wahladministration ist ein authentischer Kanal für die Registration notwendig. Nur so kann gewährleistet werden, dass die öffentliche Wahlidentität des Wählers korrekt und unverfälscht bei der Wahladministration eingeht.
Es braucht zusätzlich noch eine (virtuelle) Wahlanzeige (*bulletin board*), welche die verschiedenen Stimmzettel annimmt und publiziert. Damit die Stimmen nicht verfälscht werden können, sollte das bulletin board nur einmalig schreibendende Zugriffe (*write only once*) zulassen.

Zusätzlich wird ein *anonymer Kanal* zwischen Wähler und Wahlanzeige zur Stimmabgabe benötigt. Nur mit einem anonymen Kanal kann sichergestellt werden, dass keine Rückschlüsse zwischen Wähler und Stimmzettel gemacht werden können.

2.5. Protokoll Ablauf

Nachdem das Protokoll und die Rahmenbedingungen eingangs informell beschrieben wurden, ist es nun nötig, dass das Thema auch noch formell aufgezeigt wird. Nachfolgend werden nun die verschiedenen Phasen des Protokolls erklärt.

Registration Während der Registrationsphase generiert jeder **Wähler** V folgende zwei zufällige Teile seiner Wähleridentität:

- **Geheime Wähleridentität**: Die geheime Wähleridentität besteht aus zwei Werten α und β, welche aus der Restklassenmenge $\mathbb{Z}$ mit Basis q gewählt werden.
  $(\alpha, \beta) \in_r \mathbb{Z}_q^2$

[1] Der heutige Gegner kann nur jene Algorithmen und Funktionen effizient berechnen, welche maximal ein polynominaler Aufwand $O(f(x))$ besitzen.

- **Öffentliche Wähleridentität** u: Die öffentliche Wähleridentität u wird aus den beiden Werten α und β und den beiden Werten h_1 und h_2 berechnet. Letztere sind öffentlich bekannte Werte und Teil der aktuellen Abstimmung.
  $u = h_1^{\alpha} h_2^{\beta} \in G_q$

Nachdem der Wähler diese beiden Teile seiner Wähleridentität generiert hat, sendet er den öffentlichen Teil u über einen authentischen Kanal an die Wahladministration.

Abstimmungsvorbereitung Die Wahladministration definiert und publiziert die Liste aller zu dieser Wahl zugelassenen Wähler $U = ((V_1, u_1), ..., ((V_M, u_M))$

Zudem berechnet sie den Koeffizienten $A = (a_1, ..., a_M)$ mit dem Polynom $P(X) = \prod_{i=1}^{M}(X - ui) \in \mathbb{Z}_p[X]$. Der Koeffizient A wird von den Wählern benötigt, um zu beweisen, dass sie zur Wahl berechtigt sind (ein so genannter *Membership Proof*). Mit Hilfe der Wählerliste U könnte theoretisch jeder den Koeffizienten A nachrechnen. Da diese Berechnung aufwendig ist, wird sie nur einmalig von der Wahladministration berechnet.

Abschliessend wählt die Wahladministration einen unabhängigen Wahlgenerator $\hat{h} \in G_q$ und publiziert diesen gemeinsam mit U und A als Tripel $(U, A, \hat{h})$ auf das öffentliche bulletin board.

Stimmabgabe Der Wähler V trifft seine Entscheidung - er stimmt ab - und erzeugt seine Stimme e. Danach wird die Stimme encodiert. Das Protokoll schreibt allerdings nicht vor, wie die Stimme e encodiert sein muss.
Der Wähler V berechnet nun verschiedene Werte, welche nachfolgend beschrieben werden.

- Wahlidentität $\hat{u} = \hat{h}^{\beta} \in G_q$
- Commitment zur öffentlichen Identität $c = com_p(u, r)$ mit $r \in_R \mathbb{Z}_p$
- Commitment zur privaten Identität $d = com_q(\alpha, \beta, s)$ mit $s \in_R \mathbb{Z}_q$
- Set membership proof π_1
  $\pi_1 = NIZKP_e[(u, r) : c = com_p(u, r) \wedge P(u) = 0]$
  π_1 beweist, dass das Commitment zur öffentlichen Identität c zu einer gültigen öffentlichen Wähleridentität u aus der Wahlliste U gehört.
- Proof of known representation of a commited value π_2
  $\pi_2 = NIZKP_e[(u, r, \alpha, \beta, s) : c = com_p(u, r) \wedge d = com_q(\alpha, \beta, s) \wedge u = h_1^{\alpha} h_2^{\beta}]$
  π_2 beweist, dass derselbe Wähler die beiden Commitments c und d berechnet hat.
- General Preimage Equality Proof π_3
  $\pi_3 = NIZKP_e[((\alpha, \beta, s) : d = com_q(\alpha, \beta, s) \wedge \hat{u} = \hat{h}^{\beta}]$
  π_3 zeigt, dass das selbe β zur Berechnung der öffentlichen Identität d und der Wahlidentität $\hat{u}$ verwendet wurde. Zudem wird mit π_3 gezeigt, dass die Stimme e zu dieser Wahl gehört, und dass es sich nicht um ein Duplikat handelt.

Die obengenannten Beweise π_1, π_2 und π_3 werden jeweils in Verbindung mit der Stimme e gebracht.

Abschliessend schickt der Wähler den Stimmzettel (*Ballot*) B über einen anonymen Kanal zum bulletin board. Der elektronische Stimmzettel B enthält somit die Commitments zur öffentlichen und privaten Identität c und d, die Stimme e und die Wahlidentität $\hat{u}$, als auch die drei Beweise (proofs) π_1, π_2 und π_3.

Öffentliche Auszählung Am Ende der Wahl müssen sämtliche eingegangenen digitalen Stimmzettel (Ballots) überprüft werden, analog einer traditionellen Wahl in der Schweiz. Jeder und jede kann die Überprüfung als auch die Auszählung selbst durchführen und so verifizieren. Einerseits müssen ungültige Ballots als solche gekennzeichnet werden. Sie dürfen nicht gelöscht werden, damit jeder selbständig nochmals die Korrektheit resp. die Ungültigkeit aller Ballots überprüfen kann. Andererseits müssen doppelte Stimmabgaben über die Wahlidentität $\hat{u}$ identifiziert werden. Sofern nicht alle Stimmabgaben identisch sind, werden Konflikte anhand vordefinierter Prozesse gelöst. Hier wird unterschieden, zwischen erneut gesendeten Stimmabgaben und neu generierten Stimmen.

Everlasting Privacy Vorangehend wurden bereits grob die zwei Arten von Gegnern besprochen - dem heutigen und dem zukünftigen Gegner. Nach den theoretischen Ausführungen rund um das Protokoll ist es nun an der Zeit sie ein weniger präziser zu beschreiben.

Der heutige Gegner kann diskrete Logarithmen nicht effizient berechnen. Er kann somit kein α' und β' berechnen, welches dieselbe öffentliche Wähleridentität u ergeben würde. Der Gegner kann zudem keine valide Stimme erzeugen, so lange er kein solches Paar α' und β' kennt, weil sonst die Zuverlässigkeit (*Soundness* genannt) des *Zero-Knowledge Proofs* gebrochen wäre.

Zugleich kann ein heutiger Gegner mit seinen Mitteln nur eine Replay-Attacke ausführen. Diese Attacke lässt sich allerdings einfach feststellen, da zwei identische Stimmzettel mit derselben Wahlidentität $\hat{u} = \hat{h}^{\beta}$ eingegangen sind. Abschliessend kann festgehalten werden, dass ein heutiger Gegner keine Bedrohung für die *everlasting privacy* darstellt.

Der zukünftige Gegner stellt ebenfalls keine Gefahr für die *everlasting privacy* dar. Auch wenn dieser Gegner diskrete Logarithmen effizient berechnen und damit β aus $\hat{u} = \hat{h}^{\beta}$ berechnen kann, gibt es mehrere valide (mögliche) Lösungen. Denn u in $u = h_1^{\alpha} h_2^{\beta}$ besitzt jeweils eine Lösung für jedes Paar (u_i, β_i). Dazu kommt, dass alle Beweise (Proofs) in den Stimmzetteln (Ballots) *perfectly hiding* sind.

2.6. Commitments

Bei einem Commitment geht es darum, sich auf einen ausgewählten Wert festzulegen und diesen danach für eine gewisse Zeit zu verstecken. Ein weiterer wichtiger Anspruch an das Commitment ist, dass der ausgewählte Wert nicht verändert werden kann, solange er versteckt ist. Zur Veranschaulichung, kann man sich dies an Hand einer Auktion vorstellen, bei der die Bieter alle einen Betrag wählen und diesen z.B. in der Hand verstecken. Anschliessend zeigen alle Bieter gleichzeitig ihren gewählten Betrag und der Bieter mit dem höchsten Betrag gewinnt.

Genau wie beim Beispiel der Auktion besteht auch ein mathematisches Commitment aus zwei Phasen: *Commit* und *Reveal*

Der Commit beschreibt die erste Phase des Commitments. Der Sender wählt eine Nachricht, schliesst diese elektronisch in eine Box und übergibt diese dem Empfänger.

Die zweite Phase besteht aus dem Reveal. Hier muss der Sender dem Empfänger beweisen, dass sich seine gewählte Nachricht in der Box befindet.

2.6.1. Eigenschaften

Zwei Eigenschaften sind bei einem Commitment entscheidend: *hiding* und *binding*. Die Eigenschaft *hiding* sagt aus, wie gut die ausgewählte Nachricht nach der ersten Phase versteckt ist. Ziel ist es, dass kein bösartiger Empfänger etwas über den versteckten Wert herausfinden kann, nachdem dieser versteckt wurde.
Man unterscheidet zudem, auf welche Art versteckte Werte gegenüber den verschiedenen Arten von Empfängern geschützt werden können. Kann der Wert gegenüber einem Empfänger geschützt werden, welcher limitiert ist auf Berechnungen in polynominaler Zeit, so spricht man von *computationally hiding*. Schützt das Commitment den versteckten Wert sogar gegen einen Empfänger mit unbegrenzten Mitteln und Rechenleistung, so spricht man von *perfectly hiding*.

Ein Commitment ist dann *binding*, wenn ein bösartiger Sender am Ende der zweiten Phase nur genau für einen Wert beweisen kann, dass es sich dabei um den von ihm versteckten Wert handelt. Analog zur Eigenschaft hiding unterscheidet man auch hier aufgrund der Rechenleistung des Senders zwischen *computationally binding* und *perfectly binding*.

Wählt man zum Beispiel als Commitment-Funktion einen symmetrischen Verschlüsselungsalgorithmus wie AES, so ist AES nach den heutigen Möglichkeiten hiding. Mit unlimitierter Rechenleistung könnte AES aber mit *brute force* überwunden werden. Daher ist AES in diesem Fall nur computationally hiding. Gleichzeitig kann mit einem festgelegten Key und einem festgelegten Ciphertext nur genau ein Plaintext hervorgeholt werden, unabhängig von der Rechenleistung. AES ist daher perfectly binding.

Es stellt sich die Frage, ob sich eine Commitment-Funktion finden lässt, welche sowohl perfectly hiding als auch perfectly binding ist. Dazu müssen die beiden Eigenschaften in Beziehung zu einander betrachtet werden. Nimmt man ein beliebiges Commitment, welches die Eigenschaft perfectly hiding besitzt, kann ein Gegenspieler mit unlimitierten Rechenleistung aus dem versteckten Wert keine Informationen gewinnen. Da dieser Gegenspieler jede beliebige mathematische Funktion umkehren kann, kann eine Commitment Funktion $f(x) = y$ nur dann perfectly hiding sein, wenn zu jedem y mehrere x existieren, welche die Gleichung lösen. Somit kann selbst nach dem Finden jeder möglichen Lösung der vom Sender ausgewählte Wert nicht eindeutig bestimmt werden. Somit ist die Eigenschaft perfectly hiding erfüllt. Die Tatsache, dass zu einem beliebigen y aber mehrere x die Gleichung $f(x) = y$ lösen, wird nun aber für die Erfüllung der Eigenschaft perfectly binding zum Problem. Wie vorhin gezeigt, kann ein Gegenspieler mit unlimitierter Rechenleistung zu einem gegebenen y alle möglichen x berechnen, welche die Gleichung lösen. Somit kann ein bösartiger Sender nun jede dieser Lösungen beim Reveal als sein ursprünglich ausgewählten Wert angeben. Damit kann gezeigt werden, dass eine Commitment Funktion nicht gleichzeitig perfectly hiding und perfectly binding sein kann.

2.7. Pedersen Commitment

Im E-Voting Protokoll wird an verschiedenen Stellen das Pedersen Commitment verwendet. Es wird sowohl beim Erstellen der Public Credentials als auch bei der Generierung der drei Beweise π_1, π_2 und π_3 verwendet. Das Pedersen Commitment ist wie folgt definiert.

- Ability to open a Pedersen commitment:

$$ZKP\{(m, r) : c = commit(m, r)\}$$

- $X = \mathbb{Z}_q \times \mathbb{Z}_q$
- $Y = G_q$
- $f(x_1, x_2) = commit(x_1, x_2) = g^{x_1} h^{x_2}$
- $f[X] = G_q$ of order q
- $x = (m, r)$, $w = (w_1, w_2)$, $s = (s_1, s_2)$

Abbildung 2.1.: Beschreibung eines Pedersen Commitments, aus [4, p. 22]

Als Input verwendet das Pedersen Commitment zwei Werte aus der Restklassenmenge $\mathbb{Z}$ mit Basis q. Dabei ist q eine Primzahl. Der Parameter m ist die Message, welche versteckt werden soll, und der Parameter r wird zufällig aus der Menge $\mathbb{Z}_q$ ausgewählt. Der Output des Pedersen Commitments ist ein Wert aus G_q, einer zyklischen Untergruppe von $\mathbb{Z}_q$. Dabei sind g und h Generatoren von G_q.

Das Pedersen Commitment ist perfectly hiding und computationally binding, da bei der Gleichung $c = g^m h^r$ mit einem bestimmten c mehrere Lösungspaare existieren.
Da aus dieser Menge von Lösungen nicht das Original-Paar (m, r) bestimmt werden kann, ist das Pedersen Commitment perfectly hiding. Zudem ist eine effiziente Berechnung einer Lösung bei gegebenem c nur möglich, wenn ein diskreter Logarithmus in polynomialer Zeit gerechnet werden kann. Dies ist aber nach dem heutigen Stand noch nicht möglich, daher ist das Pedersen Commitment computationally binding.

Das Pedersen Commitment kann zusätzlich erweitert werden, so dass mehrere Message-Werte gemeinsam versteckt werden können. Anstelle von einem Generator g wird in diesem Fall für jede Message ein eigener Generator bereitgestellt. Unabhängig von der Anzahl Message Inputs wird zum Schluss immer noch ein Random Element übergeben. Die Eigenschaften perfectly hiding und computationally binding bleiben auch bei dieser Erweiterung bestehen.

2.8. Zero-Knowledge Proof of Knowledge

Ziel eines Zero-Knowledge Proofs ist es jemandem zu beweisen, dass man ein bestimmtes Geheimnis kennt, ohne dabei das Geheimnis selbst preis zu geben. Als Basis für ein Zero-Knowledge Proof dient das Schnorr Protokoll. Im Weiteren wird das Schnorr Protokoll anhand des *Proof of Knowledge of a Discrete Logarithm* erläutert.

In diesem Fall muss der *Prover* einer zweiten Person, dem *Verifier*, beweisen, dass er den diskreten Logarithmus x zu einem bestimmten Wert $y = g^x$ kennt. Die Ausgangssituation ist also, dass der Prover sowohl x als auch y kennt. Der Verifier hingegen kennt nur den Wert y. Der Ablauf ist in der Abbildung unten illustriert. In einem ersten Schritt wählt der Prover zufällig einen Wert w aus der Menge $\mathbb{Z}_q$ aus. Aus diesem Zufallswert wird anschliessend das Commitment t berechnet, indem der Wert w als Input für die gleiche Funktion verwendet wird, wie sie für die Berechnung von y verwendet wurde. In diesem Fall wäre dies $t = g^w$. Das Commitment t wird anschliessend an den Verifier übermittelt. Nach Erhalt des Commitment t wählt der Verifier zufällig einen Wert c aus der Menge $\mathbb{Z}_q$. Der Wert c wird als Challenge an den Prover geschickt. Der Prover berechnet nun aus den geheimen Werten w und x und dem erhaltenen Challenge c die Response $s = w + cx$ und schickt diese an den Verifier zurück. Der Verifier kann nun mit Hilfe der Gleichung $g^s = ty^c$ verifizieren, ob der Prover den Wert x wirklich kennt. Dass die beiden Terme gleichwertig sind, sofern die richtigen Werte übergeben wurden, lässt sich wie folgt zeigen: $g^s = g^{w+cx} = g^w g^{cx} = ty^c$

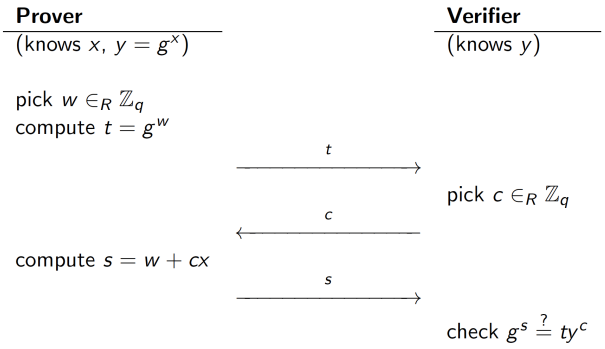


Abbildung 2.2.: Interaktiver Zero-Knowledge Proof, aus [4, p. 9]

Auch bei einem Zero-Knowledge Proof of Knowledge gibt es wichtige Eigenschaften: die *Completeness* und die *Soundness*. Die Completeness, oder auch Vollständigkeit, beschreibt die Eigenschaft, dass ein ehrlicher Prover den Verifier erfolgreich davon überzeugen kann, dass er den geheimen Wert x kennt. Die Soundness beschreibt die Eigenschaft, dass ein bösartiger Prover den Verifier nicht überzeugen kann, sofern er den geheimen Wert x nicht kennt.

In Bezug auf das oben genannte Beispiel ist die Completeness gegeben, da durch Kenntnis des Wertes x die Berechnung der Response s durchgeführt werden kann. Zusätzlich wurde bereits gezeigt, dass die so berechnete Response mit der Gleichung $g^s = ty^c$ vom Verifier kontrolliert werden kann.

Die Soundness ist in diesem Beispiel auch gegeben. Wenn ein bösartiger Prover den Wert x nicht kennt, so kann er nur durch lösen der Gleichung $g^s = ty^c$ einen korrekten Wert für die Response s berechnen. In diesem Fall entspricht dies dem Lösen eines diskreten Logarithmus und ist deshalb nach den heutigen Möglichkeiten nicht in polynominaler Zeit machbar. Somit ist die Soundness auch gegeben.

Es bleibt noch zu zeigen, dass beim Beweis tatsächlich keine Informationen über den geheimen Wert x preisgegeben werden - also ob der Beweis tatsächlich *zero-knowledge* ist. Dies ist gegeben, da zu jedem $x' \neq x$ ein w' existiert, so dass x' und w' die Gleichung $s = w' + cx'$ für ein gegebenes s und c lösen. Daher kann durch die Kenntnis von s und c nicht auf das x geschlossen werden.

2.8.1. Non-Interactive Proof of Knowledge

Ein Problem mit der zuvor gezeigten interaktiven Variante des Proof of Knowledge ist, dass der Beweis selbst nicht übertragbar ist. Das heisst, dass wenn ein Prover ein Proof of Knowledge mit mehreren Verifiern durchführen will, muss er den interaktiven Beweis mit jedem Verifier einzeln durchführen. Ein Transkript eines erfolgreichen interaktiven Proof of Knowledge reicht nicht aus, um einen zusätzlichen Verifier zu überzeugen. Ein bösartiger Prover könnte hier ein Transkript simulieren, indem er zuerst ein s und ein c bestimmt und erst anschliessend das passende t dazu berechnet. Deshalb muss ein interaktiver Proof of Knowledge für jeden Verifier einzeln durchgeführt werden.

Um dieses Problem zu lösen, wurde von Fiat und Shamir ein allgemeines Verfahren entwickelt, wie ein interaktiver Proof of Knowledge in eine nicht-interaktive Variante überführt werden kann. Die Idee besteht darin, das vom Verifier gewählte Challange c mit einem Hashwert aus allen allgemein bekannten Variablen und dem Commitment t zu ersetzen. Dadurch ist nun das Challenge c abhängig von Commitment t. Der Verifier muss in der non-interactive Variante zusätzlich das Challenge c überprüfen, indem er den Hash selbst berechnet. Dies hat zur Folge, dass es nun nicht mehr möglich ist, zuerst die Werte s und c zu bestimmen und anschliessend den Wert t zu berechnen. Geht man nach dieser Methode vor, wird die Kontrolle des Challenges c fehlschlagen, weil der ausgewählte Wert für c nicht dem Hash entsprechen wird, welcher mit dem neuberechneten Wert t erstellt wird.

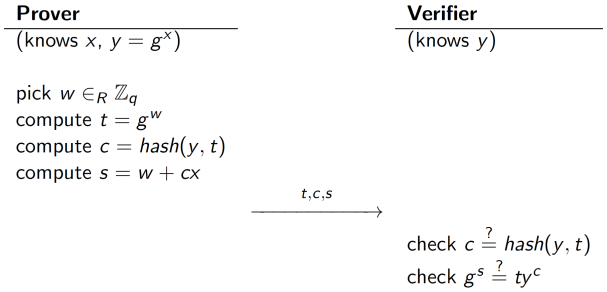


Abbildung 2.3.: Nicht-Interaktiver Zero-Knowledge Proof, aus [4, p. 15]

2.8.2. Generic Preimage Proof

Das oben behandelte Schnorr Protokoll ist eine Instanz des sogenannten Generic Preimage Proof $ZKP(x) : y = f(x)$. Der Generic Preimage Proof liefert für eine Funktion, welche gewisse Eigenschaften erfüllt, eine Vorlage für die Konstruktion eines Zero-Knowledge Proofs of Knowledge. Die Ansprüche des Generic Preimage Proof an eine Funktion f sind folgende:

- Seien $(X; \oplus; -; 0)$ und $(Y; \otimes; ^{-1}; 1)$ zwei Gruppen und $f : X -> Y$
- So muss für alle x_1,x_2 gelten: $f(x_1 \oplus x_2) = f(x_1) \otimes f(x_2)$
- Ausserdem muss es sich bei der Funktion f um eine sogenannte „one way function“ handeln. Dies bedeutet, dass die Berechnung von $y = f(x)$ effizient durchgeführt werden kann, die Berechnung der Umkehrung $x = f^{-1}(y)$ jedoch nicht in effizienter Zeit berechnet werden kann.

Sind diese Eigenschaften gegeben, so kann die Funktion f für einen Preimage Proof verwendet werden. Dabei dient $x \in X$ als Geheimnis und $y = f(x) \in Y$ als öffentlicher Wert. Nach den beiden folgenden Protokoll-Schemen von Maurer kann nun mit der Funktion f ein interaktiver bzw. nicht-interaktiver Zero-Knowledge Proof erstellt werden.

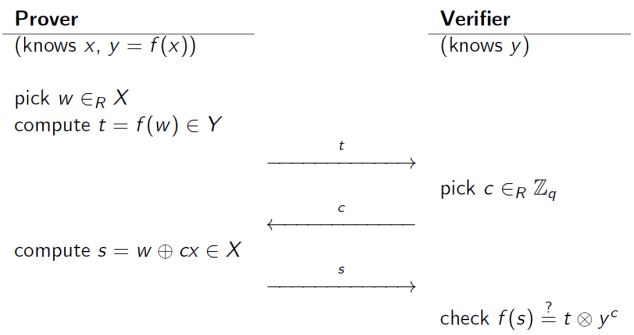


Abbildung 2.4.: Interaktiver Preimage Proof nach Maurer, aus [4, p. 19]

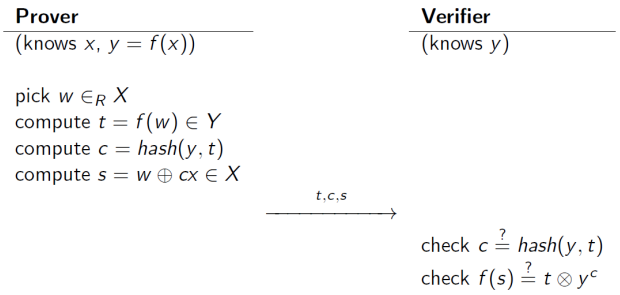


Abbildung 2.5.: Nicht-Interaktiver Preimage Proof nach Maurer, aus [4, p. 20]

2.8.3. AND-Composition

Mit der AND-Composition können mehrere Preimage Proofs in einem Beweis kombiniert werden. Dabei müssen alle beteiligten Funktionen f_i der Voraussetzungen eines Preimage Proofs genügen. Ist dies der Fall, so können die einzelnen Preimage Proofs wie folgt zu einem einzigen, zusammengesetzten Preimage Proof kombiniert werden:

$ZKP(x_1, ..., x_n) : y_1 = f_1(x_1) \wedge ... \wedge y_n = f_n(x_n) = ZKP(x_1, ..., x_n) : (y_1, ..., y_n) = f(x_1, ..., x_n)$ für $f(x_1, ..., x_n) = (f_1(x_1), ..., f_n(x_n))$

Dieser zusammengesetzte Beweis ist besonders für die non-interactive Variante des Zero-Knowledge Proofs interessant. Im Vergleich zu n separaten non-interactive Beweisen macht der kombinierte Beweis eine stärkere Aussage, indem er zeigt, dass die geheimen Werte $(x_1, ..., x_n)$ alle gleichzeitig der gleichen Partei bekannt sind.

2.8.4. General Preimage Equality Proof

Der General Preimage Equality Proof ist ein Spezialfall einer AND-Composition, in der für die Funktionen f_i jeweils derselbe geheime Wert als Input verwendet wird.

$ZKP(x) : y_1 = f_1(x) \wedge ... \wedge y_n = f_n(x) = ZKP(x) : (y_1, ..., y_n) = f(x)$ für $f(x) = (f_1(x), ..., f_n(x))$

Somit kann bewiesen werden, dass der geheime Input x das Urbild oder Preimage aller Funktionen f_i ist.

3. Beweise

3.1. π_3

Mit Hilfe der erarbeiteten Theorie zu den Commitments und den Zero-Knowledge Proofs lässt sich der Beweis π_3 herleiten. In einem ersten Schritt analysieren wir die einzelnen Bausteine, welche im Beweis verwendet werden. Wir erkennen, dass es sich um einen non-interactive Zero-Knowledge Proof handelt. Dieser besteht aus einem Pedersen Commitment und einem *Knowledge of discrete Logarithm*-Beweis. Die beiden separaten Beweise sind durch eine And Composition verbunden. Bei genauerer Analyse kann man erkennen, dass die beiden Beweise die gleichen geheimen Werte als Input verwenden. Somit handelt es sich bei π_3 um ein General Preimage Equality Proof.

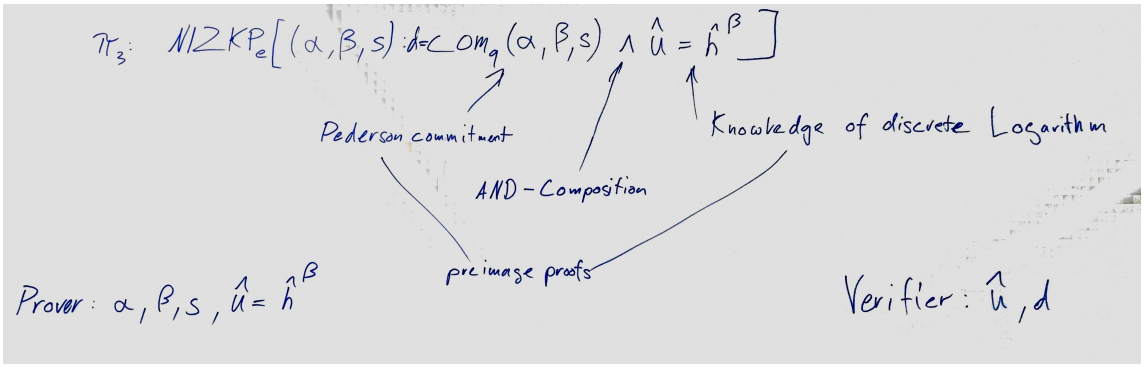


Abbildung 3.1.: Beweis π_3 mit Beschreibungen der einzelnen Teile, Wandtafel-Photo (Eigenarbeit)

3.1.1. Herleitung

Wir betrachten zuerst die einzelnen Bausteine losgelöst voneinander, um die Funktionsweise der einzelnen Bausteine zuerst zu verstehen und anschliessend zum gesamten π_3-Beweis zusammenzusetzen. Als erstes betrachten wir den Knowledge of discrete Logarithm Proof. Dieser Bestandteil wurde bereits während der Theorie behandelt. In der Abbildung wird der Knowledge of discrete Logarithm Proof nochmals aufgeführt, dabei werden für die Variablen die Bezeichnungen aus dem π_3-Beweis verwendet.

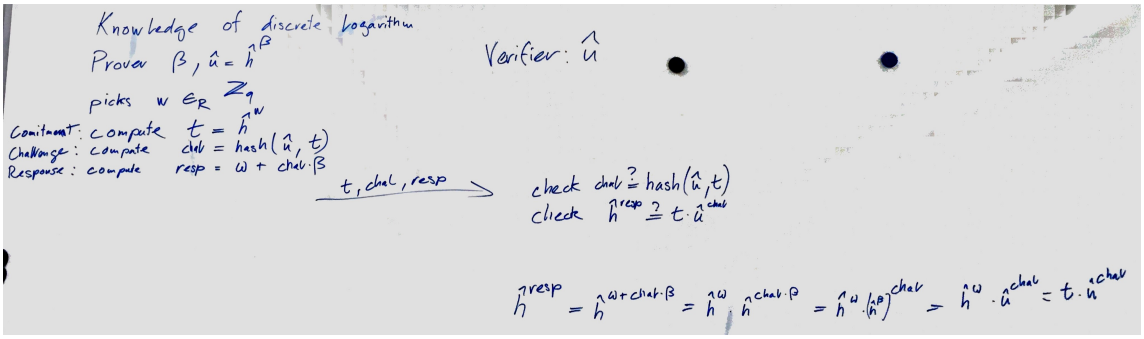


Abbildung 3.2.: Knowledge of Discrete Logarithm, Wandtafel-Photo (Eigenarbeit)

Die Herleitung des Zero-Knowledge Proofs mit dem Pedersen Commitment wird zuerst im allgemeinen Fall mit einem Message Input m betrachtet. Da die Commitment Funktion des Pedersen Commitments eine one way

Funktion ist und auch alle übrigen Eigenschaften erfüllt sind, lässt sich für den Zero-Knowledge Proof das Schema des Generic Preimage Proofs verwenden. In der Abbildung ist somit die Zero-Knowledge Proof für das Pedersen Commitment mit Hilfe des Generic Preimage Proofs Schemas hergeleitet.

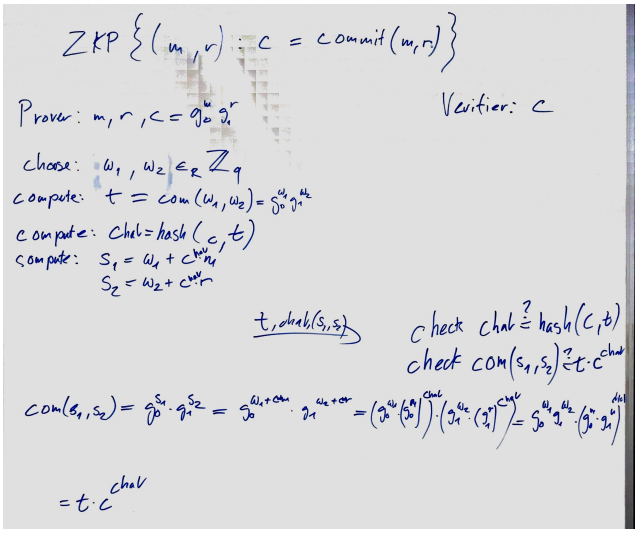


Abbildung 3.3.: Pedersen Commitment mit einem Message Input m, Wandtafel-Photo (Eigenarbeit)

In einem weiteren Schritt wird das Pedersen Commitment auf zwei Messages erweitert. Zudem wird die Benennung der Variablen auf die Notation des π_3-Beweises angepasst. Auch nach der Erweiterung des Pedersen Commitments sind die Voraussetzungen für das Generic Preimage Proof Schema weiterhin erfüllt.

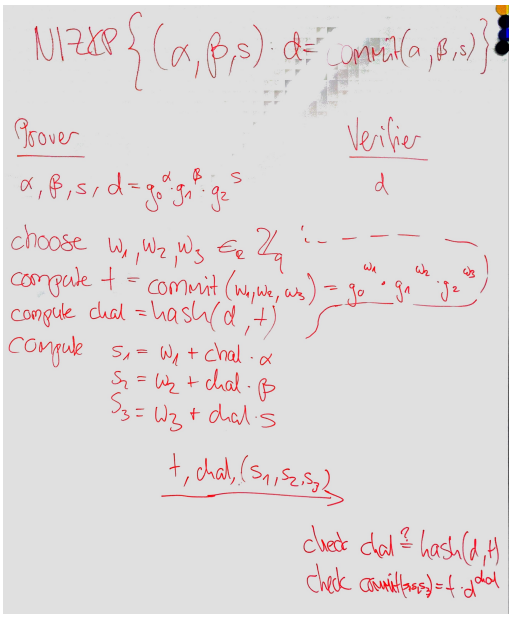


Abbildung 3.4.: Pedersen Commitment mit zwei Message Inputs α, β, Wandtafel-Photo (Eigenarbeit)

Somit sind die einzelnen Bestandteile des π_3-Beweises hergeleitet. Nun müssen noch die beiden separaten Zero-Knowledge Proofs zu den Funktionen $com(\alpha, \beta, s)$ und $\hat{h}^{\beta}$ mit dem General Preimage Equality Proof Schema zu verbinden. Dieses Schema kann in diesem Fall verwendet werden, da die beiden Funktionen dieselben Werte als Input nehmen. Dazu wird nach dem Schema aus den beiden Funktionen eine kombinierte Funktion $f(\alpha, \beta, s) = (com(\alpha, \beta, s), \hat{h}^{\beta})$ hergestellt. Anschliessend wird diese Funktion f verwendet, um den π_3-Beweis nach dem Preimage Proof Schema herzuleiten.

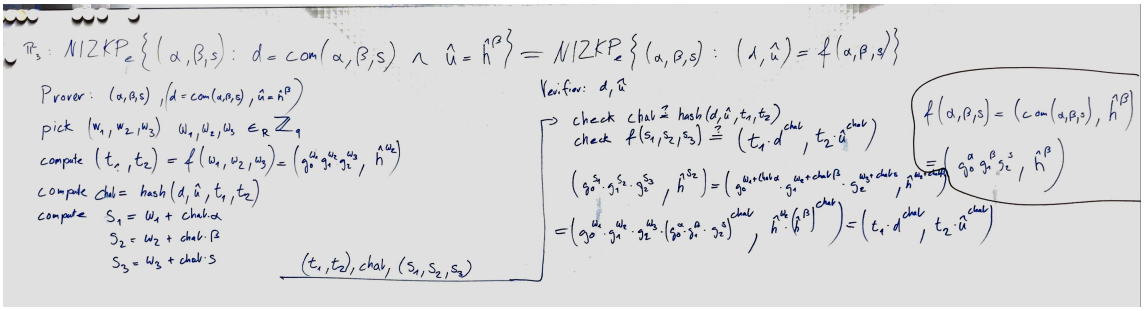


Abbildung 3.5.: Beweis π_3, Wandtafel-Photo (Eigenarbeit)

In der Abbildung des π_3-Beweises fehlt die Abhängigkeit des Beweises von der Stimme e. Diese Abhängigkeit kann erreicht werden, indem die Stimme e in den Hash für das Challenge *chal* integriert wird. Somit ist der Beweis von der Stimme e abhängig.

3.2. Non Interactive Zero-Knowledge Proofs $\pi_1 + \pi_2$

Die beiden übrigen Beweise π_1 und π_2 wurden während des Projekts nur analysiert und im Gegensatz zum Beweis π_3 nicht selbst hergeleitet. Im Fokus standen bei der Analyse vor allem zwei Punkte: Erstens die einzelnen Elemente zu untersuchen und ihren Zweck nachzuvollziehen. Zweitens festzustellen, welche der Elemente zur Komplexität der Beweise beitragen und somit am meisten zur Erhöhung des Rechenaufwands führen.

3.2.1. π_1

Der π_1-Beweis ist ein *Proof of Membership*. Dafür wählt der Prover seine public Credentials u aus und versteckt diese in einem Pedersen Commitment. Verbunden in einem Preimage Equality Proof wird zudem gezeigt, dass der gewählte Wert u aus der Menge der für diese Wahl zugelassenen Wähler U stammt. Die genaue Generierung des Beweises ist in der folgenden Abbildung aufgeführt.

Public Input: $c = \mathrm{com}_p(u, r) \in \mathcal{G}_p$, $P(X) = \sum_{i=0}^{M} a_i X^i \in \mathbb{Z}_p[X]$
Secret Input: $u, r \in \mathbb{Z}_p$
Generation:

1. For $j = 1, \ldots, m$, pick $r_j \in_R \mathbb{Z}_p$ and compute $c_j = \mathrm{com}_p(u^{2^j}, r_j)$.
2. For $j = 0, \ldots, m$, pick $\bar{a}_j, \bar{r}_j \in_R \mathbb{Z}_p$ and compute $\bar{c}_j = \mathrm{com}_p(\bar{a}_j, \bar{r}_j)$.
3. Compute new polynomial

$$\tilde{P}(X) = \sum_{j=0}^{m} \tilde{a}_j X^j = \sum_{i=0}^{M} a_i \prod_{j=0}^{m} (u^{2^j} X + \bar{a}_j)^{i[j]} X^{1-i[j]} \in \mathbb{Z}_p[X]$$

   of degree m. For $j = 0, \ldots, m$, pick $\tilde{r}_j \in_R \mathbb{Z}_p$ and compute $\tilde{c}_j = \mathrm{com}_p(\tilde{a}_j, \tilde{r}_j)$.
4. For $j = 0, \ldots, m-1$, compute $\hat{a}_j = u^{2^j} \bar{a}_j$, pick $\hat{r}_j \in_R \mathbb{Z}_p$, and compute $\hat{c}_j = \mathrm{com}_p(\hat{a}_j, \hat{r}_j)$.
5. Compute $x = h(c, a_0, \ldots, a_M, c_1, \ldots, c_m, \bar{c}_0, \ldots, \bar{c}_m, \tilde{c}_0, \ldots, \tilde{c}_m, \hat{c}_0, \ldots, \hat{c}_{m-1})$.
6. For $j = 0, \ldots, m$, compute $\bar{a}'_j = \bar{a}_j + x u^{2^j}$.
7. For $j = 0, \ldots, m$, compute $\bar{r}'_j = \bar{r}_j + x r_j$.
8. For $j = 0, \ldots, m-1$, compute $\hat{r}'_j = \hat{r}_j + x r_{j+1} - b_j r_j$.
9. Compute $\tilde{r}' = \sum_{j=0}^{m} \tilde{r}_j x^j$.

Transcript:
$(c_1, \ldots, c_m, \bar{c}_0, \ldots, \bar{c}_m, \tilde{c}_0, \ldots, \tilde{c}_m, \hat{c}_0, \ldots, \hat{c}_{m-1}, \bar{a}'_0, \ldots, \bar{a}'_m, \bar{r}'_0, \ldots, \bar{r}'_m, \hat{r}'_0, \ldots, \hat{r}'_{m-1}, \tilde{r}')$
Verification:

1. Compute $x = h(c, a_0, \ldots, a_M, c_1, \ldots, c_m, \bar{c}_0, \ldots, \bar{c}_m, \tilde{c}_0, \ldots, \tilde{c}_m, \hat{c}_0, \ldots, \hat{c}_{m-1})$.
2. For $j = 0, \ldots, m$, check $c_j^x \bar{c}_j = \mathrm{com}_p(\bar{a}'_j, \bar{r}'_j)$.
3. For $j = 0, \ldots, m-1$, check $c_{j+1}^x \hat{c}_j = c_j^{\bar{a}'_j} \cdot \mathrm{com}_p(0, \hat{r}'_j)$.
4. Check

$$\prod_{j=0}^{m} \tilde{c}_j^{x^j} = \mathrm{com}_p\left(\sum_{i=0}^{M} a_i \prod_{j=0}^{m} \bar{a}_j'^{\,i[j]} x^{1-i[j]}, \tilde{r}' \right).$$

Abbildung 3.6.: Vorgehensweise Beweis π_1, aus [1, p. 6, (Fig.1)]

Es ist zu erkennen, dass sowohl die Generierung als auch die Verifizierung des Beweises stark von der Anzahl der zugelassenen Wähler M abhängen. Zum Teil besteht auch eine Abhängigkeit von der Bitlänge m von M: $m = \lfloor log(M) \rfloor = |M| - 1$. Hier ist vor allem der Aufwand für die Generierung des Beweises entscheidend, da dieser Aufwand für jeden Wähler anfällt und mit einer Erhöhung der Wähleranzahl wächst. Dieser Rechenaufwand muss zwingend vom Rechner bzw. Device des einzelnen Wählers bewältigt werden, da der Server der Wahladministration die geheimen Werte nicht kennen darf.

Des Weiteren wurden für das bessere Verständnis die Punkte 2 bis 4 der Verifizierung nachgerechnet, um die Completeness des Beweises nachzuvollziehen. Die handschriftlichen Berechnungen sind im Anhang F hinterlegt.

3.2.2. π_2

Der π_2-Beweis ist ein *Proof of Known Representation of a Committed Value*. Hier geht es darum zu beweisen, dass man eine Repräsentation des im π_1-Beweis erstellten Commitments $c = com(u, r)$ kennt. In diesem Fall sind die Private Credentials (α, β), welche zu dem Public Credentials u gehören, eine solche Repräsentation. Somit beweist der Prover, dass er die Private Credentials zum gewählten Public Credential kennt. Die Generierung und die Verifizierung des Beweises sind in der Abbildung aufgeführt.

Public Input: $c = \mathrm{com}_p(u, r) \in \mathcal{G}_p$, $d = \mathrm{com}_q(v_1, \ldots, v_N, s) \in \mathbb{G}_q$
Secret Input: $u, r \in \mathbb{Z}_p$, $v_1, \ldots, v_N, s \in \mathbb{Z}_q$
Generation:

1. Pick $\bar{u}, \bar{r} \in_R \mathbb{Z}_p$ and compute $\bar{c} = \mathrm{com}_p(\bar{u}, \bar{r})$.
2. For $j = 1, \ldots, K$,
   (a) pick $\bar{v}_{1,j}, \ldots, \bar{v}_{N,j} \in_R \mathbb{Z}_q$ and compute $\bar{u}_j = h_1^{\bar{v}_{1,j}} \cdots h_N^{\bar{v}_{N,j}}$,
   (b) pick $\bar{r}_j \in_R \mathbb{Z}_p$ and compute $\bar{c}_j = \mathrm{com}_p(\bar{u}_j, \bar{r}_j)$,
   (c) pick $\bar{s}_j \in_R \mathbb{Z}_q$ and compute $\bar{d}_j = \mathrm{com}_q(\bar{v}_{1,j}, \ldots, \bar{v}_{N,j}, \bar{s}_j)$.
3. Compute $x = h(c, d, \bar{c}, \bar{c}_1, \ldots, \bar{c}_k, \bar{d}_1, \ldots, \bar{d}_k)$.
4. Compute $\bar{u}' = \bar{u} - xu$ and $\bar{r}' = \bar{r} - xr$.
5. For $j = 1, \ldots, K$,
   (a) for $i = 1, \ldots, N$, compute $\bar{v}'_{i,j} = \bar{v}_{i,j} - x[j]v_i$,
   (b) compute $\bar{r}'_j = \bar{r}_j - x[j] \cdot \mathrm{com}_q(\bar{v}'_{1,j}, \ldots, \bar{v}'_{N,j}, r)$,
   (c) compute $\bar{s}'_j = \bar{s}_j - x[j]s$.

Transcript:
$(\bar{c}, \bar{c}_1, \ldots, \bar{c}_k, \bar{d}_1, \ldots, \bar{d}_k, \bar{u}', \bar{r}', \bar{v}'_{1,1}, \ldots, \bar{v}'_{N,K}, \bar{r}'_1, \ldots, \bar{r}'_k, \bar{s}'_1, \ldots, \bar{s}'_k)$
Verification:

1. Compute $x = h(c, d, \bar{c}, \bar{c}_1, \ldots, \bar{c}_k, \bar{d}_1, \ldots, \bar{d}_k)$.
2. Check $\bar{c} = c^x \cdot \mathrm{com}_p(\bar{u}', \bar{r}')$.
3. For $j = 1, \ldots, K$,
   (a) check $\bar{d}_j = d^{x[j]} \cdot \mathrm{com}_q(\bar{v}'_{1,j}, \ldots, \bar{v}'_{N,j}, \bar{s}'_j)$,
   (b) compute $\bar{u}'_j = h_1^{\bar{v}'_{1,j}} \cdots h_N^{\bar{v}'_{N,j}}$, and check

$$\bar{c}_j = \begin{cases} \mathrm{com}_p(\bar{u}'_j, \bar{r}'_j), & \text{if } x[j] = 0, \\ c^{\bar{u}'_j} \cdot \mathrm{com}_p(0, \bar{r}'_j), & \text{if } x[j] = 1. \end{cases}$$

Abbildung 3.7.: Vorgehensweise Beweis π_2, aus [1, p. 7, (Fig.2)]

Die Komplexität des π_2-Beweises und somit auch der Rechenaufwand entsteht durch die K-fache Wiederholung des eigentlichen Beweises. Die Wiederholung des Beweises wird zwingend benötigt, da ein bösartiger Prover mit nur einer Durchführung des Beweises eine 50-prozentige Chance hat, den Verifier zu überzeugen, obwohl er keine passende Repräsentation kennt. Um die Soundness des Beweises zu erhöhen, wird dieser wiederholt. Der Parameter K, welcher die Anzahl Wiederholungen definiert, wird in diesem Zusammenhang Security Parameter genannt.

Zudem wurden für das bessere Verständnis die Punkte 2 und 3 der Verifizierung nachgerechnet, um die Completeness des Beweises nachzuvollziehen. Die Berechnungen sind im Anhang G hinterlegt.

4. Implementation mit Unicrypt

4.1. Allgemein

Das Unicrypt-Framework wird von der E-Voting Group der Berner Fachhochschule entwickelt und ist ein auf Java basierendes, kryptographisches Framework für die Implementation von kryptographischen Protokollen. Insbesondere für die Implementation der im E-Voting Protokoll vorkommenden Beweise π_1, π_2 und π_3 bietet das Unicrypt Framework die nötigen Bausteine. Mit Hilfe der sogenannten Proof Systems lassen sich die für das Protokoll verwendeten Beweise konstruieren.

4.2. Implementation π_3

Um mit Unicrypt vertraut zu werden, wurde der erarbeitete π_3-Beweis mit Hilfe des Frameworks implementiert. Dabei wurde ähnlich wie bei der mathematischen Herleitung des Beweises auch hier schrittweise vorgegangen. Das heisst, es wurden zuerst die einzelnen Komponenten losgelöst voneinander implementiert, um mit den einzelnen Bestandteilen an sich vertraut zu werden. Mit dem Wissen aus diesen Implementationen wurde anschliessend der vollständige Beweis umgesetzt.

```
//DiscreteLogarithm part of Pi3 NIZKP

// Create cyclic group G_q (modulo 167) and wth generator=98
GStarModSafePrime cyclicGroup = GStarModSafePrime.getInstance(167);
// generator = h-dach
Element generator = cyclicGroup.getElement(98);

// Create DiscreteLogarithScheme with h-dach as generator
DiscreteLogarithmCommitmentScheme discreteLogCommitmentScheme = DiscreteLogarithmCommitmentScheme.getInstance(generator);

//choose beta as 42 (normaly beta is chosen at random)
Element beta = discreteLogCommitmentScheme.getMessageSpace().getElement(42);

// Create commitment to beta = û
Element commitment = discreteLogCommitmentScheme.commit(beta);

// Decommit = Reveal
BooleanElement result = discreteLogCommitmentScheme.decommit(beta, commitment);
```

Abbildung 4.1.: Implementation Discrete Logarithm aus π_3, Code-Fragment aus Anhang H

Als erstes wurde der Teil $\hat{u} = \hat{h}^{\beta}$ umgesetzt. Dabei wurde die von Unicrypt bereitgestellte Klasse *DiscreteLogarithmCommitmentScheme* verwendet. Während dieser ersten Implementation fiel auf, dass in Unicrypt keine losgelösten Werte bzw. Elemente existieren. Ein Element ist stets an jene Gruppe gebunden, aus der das Element stammt.

```
// PedersenCommitment with 2 messageGenerators for Pi3 NIZKP

// Create cyclic group G_q (modulo 167) with 3 generators( 2 message generators an 1 random generator) generators are chosen at random
CyclicGroup cyclicGroup = GStarModSafePrime.getInstance(167);
Element generatorAlpha = cyclicGroup.getRandomGenerator();
Element generatorBeta = cyclicGroup.getRandomGenerator();
Element generatorR = cyclicGroup.getRandomGenerator();

// Create a PairElement containing the 2 Messeage generators
Pair messageGenerators = Pair.getInstance(generatorAlpha,generatorBeta);

// Create PedersenCommitmentScheme for two message and one random Element
GeneralizedPedersenCommitmentScheme pedersenCommitmentScheme = GeneralizedPedersenCommitmentScheme.getInstance(generatorR,messageGenerators);


// Create a Pair of messageElements = (alpha, beta) (normaly alpha and beta are chosen at random)
Tuple messageElements = pedersenCommitmentScheme.getMessageSpace().getElementFrom(16,42);
// Create a randomElement for the commitment
Element rElement = pedersenCommitmentScheme.getRandomizationSpace().getRandomElement();

// Create commitment for alpha and beta
Element commitment = pedersenCommitmentScheme.commit(messageElements, rElement);

// Decommit = Reveal
BooleanElement result = pedersenCommitmentScheme.decommit(messageElements, rElement, commitment);
```

Abbildung 4.2.: Implementation Pedersen Commitment aus π_3, Code-Fragment aus Anhang H

In einem zweiten Schritt wurde das Pedersen Commitment $d = com(\alpha, \beta, s)$ in Unicrypt umgesetzt. Hierzu wurde die Klasse *GeneralizedPedersenCommitmentScheme* verwendet. Mithilfe dieser Klasse lassen sich Pedersen Commitments mit beliebig vielen Message-Werten umsetzten. Bei der Konstruktion wird genau ein Generator für das Random-Element und beliebig viele Message-Generatoren übergeben. Die Anzahl übergebener Message Generatoren bestimmt wie viele Message-Inputs das Pedersen Commitment hat. Wird anschliessend die Commit Funktion des Schemas verwendet, so wird als Input das Random-Element als alleinstehendes Element übergeben und die Message-Elemente werden alle gemeinsam in einem Element „verpackt“ übergeben.

Im letzten Schritt wurden die beiden erarbeiteten Einzelteile zu einem einzigen Beweis zusammengefügt. Die Erstellung eines Beweises wird in Unicrypt mit Hilfe der ProofSystem-Klassen erreicht. Wie bereits in der Herleitung des π_3-Beweises erwähnt handelt sich hierbei um ein Equality Preimage Proof, da die beiden Bestandteile des Beweises durch ein logisches Und verknüpft sind und beide mit denselben Input-Werten arbeiten. Aus diesem Grund wird für die Programmierung des Beweises auch hier die *EqualityPreimageProofSystem*-Klasse verwendet. Zur Konstruktion des Beweises können dieser Klasse die Funktionen der jeweiligen Bestandteile übergeben werden. Diese Funktionen können aus den in Schritt eins und zwei erarbeiteten Commitment-Schemes extrahiert werden.

```
// Create PedersenCommitmentScheme for two message and one random Element
GeneralizedPedersenCommitmentScheme pedersenCommitmentScheme = GeneralizedPedersenCommitmentScheme.getInstance(generatorR,messageGenerators);

//Extract the the CommitmentFunction from the pedersenCommitmentScheme
Function pedersenFunction = pedersenCommitmentScheme.getCommitmentFunction();

//in order to put the two functions in the same ProofSystem they need to have the same Domains.

//define a selection Function with the same Domain as the PedersenCommitmentFunction that simply selects the Beta Element and returns it
ProductSet space = (ProductSet) pedersenFunction.getDomain();
Function adapterFunction = SelectionFunction.getInstance(space, 0,1); //0,1 : Pfad für selektiertes Element 0 für MessageElements -> 1 für Beta
// We chain the selection Function and the discreteLogFunction together. So the output of the selection function acts as input for the discreteLogFunction
Function adaptedDiscreteLogFunction = CompositeFunction.getInstance(adapterFunction,discreteLogFunction);

//Create a EqualityPreimageProofSystem with the PedersenCommitmentFunction and the adapted DicreteLogFunction

EqualityPreimageProofSystem proofSystem = EqualityPreimageProofSystem.getInstance(pedersenFunction, adaptedDiscreteLogFunction);
```

Abbildung 4.3.: Implementation Commitment Scheme aus π_3, Code-Fragment aus Anhang H

Allerdings stellt sich in diesem Zusammenhang das Problem, dass die Funktionen aus den beiden Schemen nicht die gleiche Menge an Inputs benötigen. Während die Funktion für den diskreten Logarithmus nur ein Input β benötigt, nimmt die Funktion für das Pedersen Commitment drei Inputs α, β, s. Zusätzlich ist beim Input auch die Strukturierung entscheidend. Wie bereits erwähnt wird im Falle der Pedersen Commitment Funktion das Random Element s einzeln übergeben und die beiden Werte α und β „verpackt“ in einem Element. Genauer betrachtet wird der Input also in der Form $((\alpha, \beta), s)$ übergeben. Das *EqualityPreimageProofSystem* kann nur mit Funktionen arbeiten, deren Input genau gleich strukturiert ist. Um dies zu erreichen wird eine Art Adapterfunktion erstellt, welche als Input $((\alpha, \beta), s)$ nimmt und daraus das β selektiert und ausgibt. Anschliessend wird die Adapterfunktion mit der diskreten Logarithmus Funktion verkettet. Die daraus resultierende Funktion ist nun kompatibel mit der Pedersen Commitment Funktion.

Somit können die Pedersen Commitment Funktion und die adaptierte diskrete Logarithmus Funktion dem *EqualityPreimageProofSystem* übergeben werden. Die vollständige Implementation des π_3-Beweises mit Unicrypt ist im Anhang H dieses Dokuments enthalten.

4.3. Analyse π_2 Implementation in Unicrypt

Um das Verständnis des Unicrypt Frameworks zu vertiefen, wurde zusätzlich die Implementation des π_2-Beweises analysiert und nachvollzogen. Der Beweis ist in der Klasse *DoubleDiscreteLogProofSystem* implementiert.

Durch Vergleichen der Implementation und der Theorie wurden in einem ersten Schritt die relevanten Variablen im Code identifiziert. Gleichzeitig eine Übersetzungstabelle zwischen den Variablennamen der Implementation und den Variablenbezeichnungen aus der Theorie erstellt. In einem zweiten Schritt wurden die einzelnen Codesegmente analysiert und den jeweiligen Schritten der Theorie zugeordnet. Die Notizen befinden sich im Anhang G.

5. Implementation mit GWT / Vaadin

5.1. Überblick

Im Hinblick auf die Bachelor-Thesis wurde untersucht, ob es möglich ist, Teile der bestehenden Unicrypt-Bibliothek plattform-unabhängig via Webbrowser zur Verfügung zu stellen. Im Rahmen des BFH-Moduls „*7081 Software Engineering and Design*“ wurde für diesen Zweck das Webanwendungs-Framework Vaadin verwendet. Vaadin ist (vereinfacht gesagt) eine server-seitige Architektur, welche in Java geschriebene Befehle beim Client als JavaScript ausgibt. Für den Client-seitigen Code greift Vaadin auf das Google Web Toolkit (GWT) zurück, welches Java-Code in JavaScript-Code kompiliert.

5.2. Proof of Concept

Im Rahmen dieser Projektarbeit sollte ein Proof of Concept (PoC) realisiert werden, der Teile von Unicrypt mit Hilfe von Vaadin publiziert. Erwähnenswert ist hier sicherlich die Anforderung, dass die Wahlidentitäten (*private* und *public key*) client-seitig berechnet werden müssen, da sonst die Privacy nicht gewährleistet werden kann.
Im Verlaufe des Projekts musste allerdings festgestellt werden, dass Vaadin für diesen PoC nicht das richtige Tool ist. Einerseits erzeugt Vaadin einen gewissen Overhead, weil neben Vaadin noch GWT verwendet wird, und andererseits war es schwieriger zu evaluieren, ob die Berechnungen auf Seite Client oder Server gemacht wurden.
Nach diversen Internet-Recherchen wurden der PoC und die nachfolgende Performance-Analyse mit GWT realisiert. GWT bietet zudem den Vorteil, dass es vorgefertigte Plugins für die Entwicklungsumgebung (IDE) Eclipse gibt.

5.3. Performance Analyse

Die Performance-Analyse wurde anhand des GWT-Beispielprojekts (Google "Web Application Project") in Eclipse realisiert, da es von Haus aus eine Client-Server-Architektur aufweist. Diese Architektur vereinfachte es, sicher zu stellen, dass die Werte auf Seite Client berechnet werden.
Es wurde ein zufällig gewählter 1024-Bit-langer BigInteger-Wert als Basis gewählt und anschliessend mit einen zufälligen 1024-Bit-langen BigInteger exponenziert. Diese benötigte Zeitdauer gab einen ungefähren Anhaltspunkt, wie lange es dauern würde, die Wahlidentitäten zu berechnen.

Der nachfolgende Code-Ausschnitt (aus Anhang I) zeigt exemplarisch, wie die Performance gemessen wurde:

```
// create 3 BigInteger objects
BigInteger bi1, bi2, bi3;

// create a BigInteger exponent
BigInteger exponent = new BigInteger(1024, new Random());

bi1 = new BigInteger(1024, new Random());
bi2 = new BigInteger(1024, new Random());

// perform modPow operation on bi1 using bi2 and exp
bi3 = bi1.modPow(exponent, bi2);

String str = bi1 + "^" +exponent+ " mod " + bi2 + " is " +bi3;
```

Abbildung 5.1.: Code-Ausschnitt der 1024-Bit-Exponentiation

Die Berechnung der Exponentiation dauerte zwischen wenigen Sekunden und einer Minute. Dies war merklich länger, als seitens Auftraggebern erhofft.

Die Tests haben zudem gezeigt, dass der Java-Code tatsächlich in JavaScript übersetzt wird und an den Webbrowser in Form einer .js-Datei übertragen wird. Die nachfolgende Abbildung zeigt die Warnung, welche Firefox angezeigt hat, falls die Berechnung nach 10 Sekunden noch nicht abgeschlossen war.

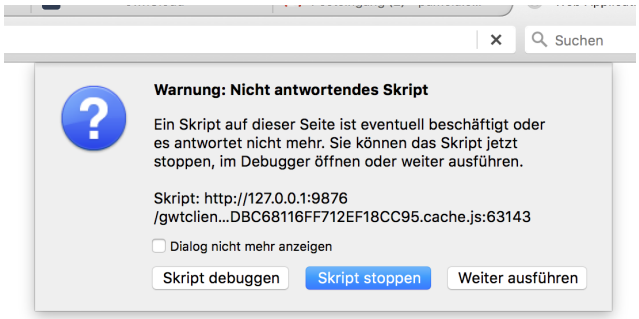


Abbildung 5.2.: JavaScript Warnung im Firefox

Bei der nachfolgenden Analyse der Warnung wurde erneut bestätigt, dass die Berechnungen im JavaScript und somit Client-seitig geschehen:

```
63128    res_0_g$ = initDim_0_g$(I_classLit_0_g$, {1401:1, 1407:1, 1429:1, 1:1}, 0, (modulusLen_0_g$ << 1) + 1, 7, 1);
63129    multArraysPAP_0_g$(a_0_g$.digits_1_g$, min_3_g$(modulusLen_0_g$, a_0_g$.numberLength_1_g$), b_0_g$.digits_1_g$,
63130    monReduction_0_g$(res_0_g$, modulus_0_g$, n2_0_g$);
63131    return finalSubtraction_0_g$(res_0_g$, modulus_0_g$);
63132  }
63133
63134  function monReduction_0_g$(res_0_g$, modulus_0_g$, n2_0_g$){
63135    $clinit_Division_0_g$();
63136    var i_0_g$, innnerCarry_0_g$, j_0_g$, j0_0_g$, m_0_g$, modulusDigits_0_g$, modulusLen_0_g$, outerCarry_0_g$;
63137    modulusDigits_0_g$ = modulus_0_g$.digits_1_g$;
63138    modulusLen_0_g$ = modulus_0_g$.numberLength_1_g$;
63139    outerCarry_0_g$ = {l:0, m:0, h:0};
63140    for (i_0_g$ = 0; i_0_g$ < modulusLen_0_g$; i_0_g$++) {
63141      innnerCarry_0_g$ = {l:0, m:0, h:0};
63142      m_0_g$ = toInt_0_g$(unsignedMultAddAdd_0_g$(res_0_g$[i_0_g$], n2_0_g$, 0, 0));
63143      for (j0_0_g$ = 0; j0_0_g$ < modulusLen_0_g$; j0_0_g$++) {
63144        innnerCarry_0_g$ = unsignedMultAddAdd_0_g$(m_0_g$, modulusDigits_0_g$[j0_0_g$], res_0_g$[i_0_g$ + j0_0_g$],
63145        res_0_g$[i_0_g$ + j0_0_g$] = toInt_0_g$(innnerCarry_0_g$);
63146        innnerCarry_0_g$ = shru_0_g$(innnerCarry_0_g$, 32);
63147      }
63148      outerCarry_0_g$ = add_3_g$(outerCarry_0_g$, add_3_g$(and_0_g$(fromInt_0_g$(res_0_g$[i_0_g$ + modulusLen_0_g$])
63149      res_0_g$[i_0_g$ + modulusLen_0_g$] = toInt_0_g$(outerCarry_0_g$);
63150      outerCarry_0_g$ = shru_0_g$(outerCarry_0_g$, 32);
63151    }
63152    res_0_g$[modulusLen_0_g$ << 1] = toInt_0_g$(outerCarry_0_g$);
63153    for (j_0_g$ = 0; j_0_g$ < modulusLen_0_g$ + 1; j_0_g$++) {
63154      res_0_g$[j_0_g$] = res_0_g$[j_0_g$ + modulusLen_0_g$];
63155    }
63156  }
63157
63158  function multiplyAndSubtract_0_g$(a_0_g$, start_0_g$, b_0_g$, bLen_0_g$, c_0_g$){
63159    $clinit_Division_0_g$();
63160    var carry0_0_g$, carry1_0_g$, i_0_g$;
63161    carry0_0_g$ = {l:0, m:0, h:0};
63162    carry1_0_g$ = {l:0, m:0, h:0};
63163    for (i_0_g$ = 0; i_0_g$ < bLen_0_g$; i_0_g$++) {
63164      carry0_0_g$ = unsignedMultAddAdd_0_g$(b_0_g$[i_0_g$], c_0_g$, toInt_0_g$(carry0_0_g$), 0);
63165      carry1_0_g$ = add_3_g$(sub_0_g$(and_0_g$(fromInt_0_g$(a_0_g$[start_0_g$ + i_0_g$]), {l:4194303, m:1023, h:0}),
```

Abbildung 5.3.: JavaScript Code, welcher GWT aus der Java Applikation generiert hat

Dazu kam, dass die Website von einem Smartphone (im Test iPhone 6S mit iOS 9.3.2 und Safari Webbrowser) mit einem Timeout abbrach und folgenden Fehler anzeigte: „Safari konnte die Seite nicht öffnen, da der Server nicht mehr antwortet."

Abschliessend wurden diese Berechnungen noch mit 2048-Bit-langen BigInteger-Werten durchgeführt. Entweder stürzte hierbei der Webbrowser ab oder der Entwicklungsserver war sehr stark ausgelastet - ein Resultat erschien auch nach mehreren Minuten Wartezeit nicht.

5.4. Schlussfolgerung

Das GWT-Framework hätte die Implementation einer Client-Applikation in JavaScript vereinfacht, weil die bestehende Unicrypt-Java-Bibliothek hätte genutzt werden können. Zudem wären Änderungen in Unicrypt einfacher und schneller in die neue Client-Applikation hineingeflossen. Auf Grund der mässigen bis schlechten Performance bei BigInteger-Berechnungen kann GWT allerdings zum heutigen Zeitpunkt nicht als Basis einer plattformunabhängigen Client-Applikation genutzt werden.

6. Fazit

Die im Rahmen dieses Projekts erarbeiteten Themen sind eine ideale Vorbereitung auf die Bachelor-Thesis im Gebiet E-Voting. Das Coaching der Betreuer lenkte geschickt durch die anfänglich sehr kompliziert wirkende Materie. Der Projektstart mit dem Paper [1] war wie „ein Wurf ins kalte Wasser“ und zwang die beiden Projektarbeiter sich mit dem Thema von Anfang an auseinander zu setzen.

Auch wenn das Thema und die mathematische Theorie hinter dem Protokoll nach wie vor komplex ist, konnte eine Basis und ein Grundverständnis aufgebaut werden. Beispielsweise wurden mathematische Konstrukte wie *Commitments*, *Zero-Knowledge Proofs* und *Non-Interactive Proofs* theoretisch als auch praktisch erarbeitet. Die Herleitung der Beweise π_1, π_2 und π_3 war interessant und lehrreich, und diente nicht zuletzt auch als "Mathe-Auffrischer".

Neben der Mathematik kamen auch informations-theoretische Themengebiete, wie *Completeness* und *Soundness*, zur Sprache. Zudem wurden (E-)Voting-spezifische Themen wie Fairness und *bulletin board* und deren Einfluss betrachtet. Schliesslich konnte LaTex als Sprache und deren Verwendung kennengelernt werden.

Die Implementation von π_3 mithilfe von Unicrypt und die Implementationsversuche sowie Performance-Analyse von GWT trugen dazu bei, dass nicht nur theoretisches Wissen vorhanden ist, sondern auch ein Gespühr für die technische Machbarkeit und die Ressourcen-Anforderungen.

Rückblickend kann festgehalten werden, dass es keinen Sinn macht eine GWT-Unicrypt-Implementation im Rahmen der Thesis zu verfolgen. Auch wenn GWT und der Gedanke der Übersetzung von Java in JavaScript nach wie vor sehr interessant ist, fehlt (jedenfalls zurzeit) die nötige Performance im Bereich der BigInteger-Exponentiation.

Eine graphische Implementation mithilfe von JavaFX oder die Programmierung einer Android-App wurde bereits als Alternative mit dem Betreuer-Team besprochen. Der Grundgedanke der Betreuer sollte auch damit erreicht werden. Denn es würde eine Protokoll-Implementation für Clients entstehen, welche die nötigen Berechnungen selbst durchführen kann und nicht all zu viele Voraussetzungen (z.B. Plugins oder spezifische Betriebssyteme) hat. Dazu kommt, dass JavaFX als auch Android Java als Basis hat und Unicrypt in letzterem realisiert wurde.

Selbständigkeitserklärung

Wir bestätigen, dass wir die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der im Literaturverzeichnis angegebenen Quellen und Hilfsmittel angefertigt haben. Sämtliche Textstellen, die nicht von uns stammen, sind als Zitate gekennzeichnet und mit dem genauen Hinweis auf ihre Herkunft versehen.

Ort, Datum:	Bern, 03.07.2016	
Namen Vornamen:	Bürk Timo	Nellen Sebastian
Unterschriften:	.....	.....

Literaturverzeichnis

[1] P. Locher and R. Haenni, "Verifiable internet elections with everlasting privacy and minimal trust," 2015.

[2] U. Maurer, "Unifying zero-knowledge proofs of knowledge," 2009.

[3] S. Bayer and J. Groth, "Zero-knowledge argument for polynomial evaluation with application to blacklists," 2013.

[4] R. Haenni, "Advanced cryptography for security and privacy," 2015.

Abbildungsverzeichnis

A. Projektablauf

Im nachfolgenden Abschnitt wurde stichwortartig festgehalten, welche Aufgaben und Themen während der Projektarbeit angegangen wurden. Die wichtigsten Projekt-Meilensteine wurden fett markiert.

- **Projektstart, 22. Februar 2016**
- Einlesen und einarbeiten in das Paper „Verifiable Internet Elections with Everlasting Privacy and Minimal Trust“ von Philipp Locher und Dr. Rolf Haenni aus dem Jahr 2015 [1]
- **Präsentation & Besprechung 1, 03. März 2016**
  - Aufbau des Protokolls
  - Mögliche Gegner
  - Vertrauensannahmen
  - Infrastruktur
- Erarbeiten der Theorie
  - Commitments generell
  - Pedersen Commitment
  - Schnorr-Protokoll (Proof of Knowledge of Discrete Logarithm)
  - Begriffe Completeness & Soundness
  - Zero-Knowledge
  - Interaktive und nicht-interaktive (mathematische) Beweise
  - Herleitung des Beweises π_3
- **Präsentation & Besprechung 2, 18. März 2016**
  - Pedersen Commitment
  - Schnorr-Protokoll (Proof of Knowledge of Discrete Logarithm)
  - Vergleich interaktiver und nicht-interaktiver Beweise
  - Beweis π_3
- Einarbeiten in LaTeX
- Implementation von π_3 mithilfe von Unicrypt
- **Präsentation & Besprechung 3, 07. April 2016**
  - Präsentation in LaTex erstellt, Retrospektive und Diskussion
  - Implementation von π_3 mithilfe von Unicrypt
  - Besprechung der Implementation und des Codes
- Implementation/Integration von Unicrypt in Vaadin und GWT
- Analyse von Philipp Locher's π_2 Implementation
- Theoretische Analyse der Beweise π_1 und π_2

- **Präsentation & Besprechung 4, 12. Mai 2016**
  - Implementation/Integration von Unicrypt in Vaadin und GWT
  - Analyse von Philipp Locher's π_2 Implementation
  - Theoretische Analyse der Beweise π_1 und π_2 Beweise
  - Komplexität von π_1 und π_2
- Perfomance-Analyse der BigInteger-Exponentiation mit GWT
- **Abgabe Projektbericht, 04. Juli 2016**
- **Schlusspräsentation, 11. Juli 2016**

B. Präsentation 1, 03.03.2016

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Everlasting privacy in E-Voting

Summary / presentation of paper

- Modul 7302 «Projekt 2»

Introduction

Introduction

- new **cryptographic Internet voting protocol** based
  - on a **set membership proof**
  - on a **proof of knowledge of the representation of a committed value**
- **Not depending** on the usual trust assumptions:
  - Presence of **trusted authorities**
    - e.g. mixing votes, decrypting them, etc.
  - Limit of **adversary's computational capabilities**
    - Postpones privacy breach but doesn't prevent it

Contribution of this protocol

- new cryptographic voting protocol for remote electronic elections
- guarantees the **secrecy of the vote** without relying on
  - trusted authorities
  - computational intractability assumptions
- offers **everlasting privacy** in an information-theoretical sense
- **Trusted parties** are only needed **for fairness**
- **Computational intractability assumptions** are **only necessary** to prevent the creation of invalid votes **during** the **voting period**

Informal procedure (1)

- **Registration**
  - Each voter creates **a pair of private and public voter credentials**
  - sends the **public voter credentials** over an authentic channel to the **election administration**
- **Election Preparation**
  - The election administration publishes the **list of public voter credentials** - one for every registered voter - on the **public bulletin board**.

Informal procedure (2)

- **Vote Casting**
  - The voter creates an **electronic ballot** and sends it over an **anonymous channel** to the public bulletin board.
  - The **ballot** consists of the **vote**, a **commitment to the voter's public credential**, and the above-mentioned **composition of zero-knowledge proofs**.
- **Public Tallying**
  - At the end of the election period, anyone can derive the final election result from the data published on the public bulletin board. The **correctness of the result** follows from **verifying the zero-knowledge proofs** included in the ballots.

Limitations

- relatively **high computational costs** for **generating** and **verifying** the **ballots**
  - with today's technology, this is only a drawback for very large electorates.

Adversary Models & Trust assumptions

Adversary Models

- **1. Present day adversary**
  - Acts **before and during the voting period**
  - Goals:
    - **Breaking integrity**: Forge a valid vote in somebody elses name
    - **Breaking secrecy**: being able to link a vote to a voter
  - Capabilities:
    - Bound to todays restrictions:
      - Polynomially bounded
      - Not able to calculate discrete logarithms or breaking contemporary hash functions

Adversary Models

- **2. future adversary**
  - Acts **after the voting period** at some point in the future
  - Goals:
    - **Breaking secrecy**: being able to link a vote to a voter
  - Capabilities:
    - Bound to no restrictions:
      since there is no way to tell how future technologies affect the adversary's capabilities

Trust/Infrastructure

- **Authentic Channel** between voter and election administration for registration
- Public **bulletin board** for collecting election data
- **Anonymous channel** between voter and bulletin board for vote casting

Protocol description

Protocol 1: Registration

- Each Voter V **generates private credentials** by choosing $(\alpha, \beta) \in_R \mathbb{Z}^2_q$ at random
- Voter **computes** the **public credential u**

  $u = h_1^{\alpha} h_2^{\beta} \in G_q$
- Voter **sends public credential u to the election administration** over authentic channel

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

Protocol 2: Preparation

- election administration defines **list of all Voters who are permitted** to participate in the election
  $U = ((V_1, u_1), \ldots, (V_M, u_M))$
- election administration **computes the coefficients A** = $(a_1, \ldots, a_M)$ of the polynomial
  $P(X) = \prod_{i=1}^{M} (X - ui) \in \mathbb{Z}_p[X]$
- Coefficients A are used by the Voters to construct their membership proof during vote casting
- Using the list of allowed voters U everyone could calculate A but since it is expensive this is done only once by the election administration
- Finally the election administration defines an independent election **generator** $\hat{h} = G_q$
- **$(U, A, \hat{h})$ gets posted to the public bulletin board**

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

Protocol 3: Vote Casting 1

- Voter chooses his election option and create the corresponding vote e
  - How the vote e is encoded is not defined, but the protocol doesn't impose any restrictions
- Voter computes **election credential** $\hat{u} = \hat{h}^{\beta} \in Gq$
- Commitment to the **public credential c** = $com_p(u,r)$ r ∈ $_R \mathbb{Z}_p$
- Commitment to the **private credentials d** = $com_q(\alpha,\beta,s)$ s ∈ $_R \mathbb{Z}_q$

Protocol 3: Vote Casting 2

- Computes **set membership proof** $\pi_1 = NIZKPe[(u,r) : c = com_p(u,r) \wedge P(u) = 0]$ to proof that the commitment c indeed belongs to a valid u from the voter list U
- Computes the **proof of known representation** of a committed **value** $\pi_2 = NIZKPe\left[(u, r, \alpha, \beta, s): c = com_p(u,r) \wedge d = com_q(\alpha,\beta,s) \wedge u = h_1^{\alpha} h_2^{\beta}\right]$ this links commitment c to private credentials
- The third prove $\pi_3 = NIZKPe\left[(\alpha, \beta, s): d = com_q(\alpha,\beta,s) \wedge \hat{u} = \hat{h}^{\beta}\right]$ to show that the used β to calulate d and $\hat{u}$ was the same. This shows that the the **vote is fresh and belongs to the current election**
- All of these **proofs are linked to e**
- Lastly the ballot B = (c, d, e, $\hat{u}$, π_1, π_2, π_3) is sent to the bulletin board over an anonymous channel

Protocol 4: Tallying

- All of the **comitted Ballots have to be verified**
  - **Invalid ballots are marked not deleted** so everyone could still verify them by themselves
  - **Duplicates can by identified** by having the same $\hat{u}$ value. Conflicts are resolved by a predetermed policy
- The **verify and tallying process can by verified by anyone**

Everlasting Privacy (1)

- **Present day adversary**:
  - **Can't compute** $(\alpha', \beta') \in_R \mathbb{Z}^2_q$ such that u = $h_1^{\alpha'} h_2^{\beta'}$ since that would be equivalaent to solve the **discret logarithm problem**
  - **Can't forge a valid vote without knowing such a pair** (α', β') because of the soundness of the zero knowledge proofs
  - **Can't create duplicates** because that would result in two votes having the same election credential $\hat{u} = \hat{h}^{\beta}$

Everlasting Privacy (2)

- **Future adversary**:
  - While a future adversary can solve the discrete logarithm and therefor calculate β from $\hat{u}$.
  - However since u = $h_1{}^{\alpha} h_2{}^{\beta}$ has a solution for every pairing (u_i, β_i) and all proofs in the Ballot are perfectly hiding the protocol provides **everlasting privacy**

Extensions

- **Achieving Fairness**:
  - Fairness: Fairness means that nobody can know partial results during the voting period
  - To achieve fairness votes are encrypted before posting the Ballot to the bulletin board
- **Multiple Elections**:
  - If protocol is used for multiple elections, but without requiring voters to renew their credentials, then a future adversary will be able to link the votes from the same voter by uncovering the same value from different election credentials.

Open Topics

Open Topics

- Mathematical **theory about "NIZKP"** (non-interactive zero-knowledge proof)
- **Further planning**
  - Deliverables
  - Assessment

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

C. Präsentation 2, 18.03.2016

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Everlasting privacy in E-Voting

Pedersen Commitment / Schnorr Protocol / NIZKP / π_3

- Modul 7302 «Projekt 2»

Agenda

- Pedersen Commitment
- Proof of knowledge of discrete logarithm (Schnorr)
- Comparing Interactive and Non-Interactive Proof of Knowledge
- Proof π_3

Screenshots based on theory of R. Haenni's Advanced Crypto course and http://www.cs.utexas.edu/~shmat/courses/cs380s_fall09/16zk.ppt

Pedersen Commitment

Commitment

- Temporarily hide a value, but ensure that it cannot be changed later
  - Example: sealed bid at an auction
- 1st stage: commit
  - Sender electronically "locks" a message in a box and sends the box to the Receiver
- 2nd stage: reveal
  - Sender proves to the Receiver that a certain message is contained in the box

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

Properties of Commitment Schemes

- Commitment must be hiding
  - At the end of the 1st stage, no adversarial receiver learns information about the committed value
  - If receiver is probabilistic polynomial-time, then computationally hiding; if receiver has unlimited computational power, then perfectly hiding
- Commitment must be binding
  - At the end of the 2nd stage, there is only one value that an adversarial sender can successfully "reveal"
  - Perfectly binding vs. computationally binding

- Ability to open a Pedersen commitment:

$$ZKP\{(m,r) : c = commit(m,r)\}$$

- → $X = \mathbb{Z}_q \times \mathbb{Z}_q$
- → $Y = G_q$
- → $f(x_1, x_2) = commit(x_1, x_2) = g^{x_1} h^{x_2}$
- → $f[X] = G_q$ of order q
- → $x = (m, r)$, $w = (w_1, w_2)$, $s = (s_1, s_2)$

Proof of knowledge of DL

Schnorr Protocol (1991)

Schnorr Protocol (1991)

Prover		**Verifier**
(knows x, $y = g^x$)		(knows y)
pick $w \in_R \mathbb{Z}_q$		
compute $t = g^w$		
	$t \longrightarrow$	
		pick $c \in_R \mathbb{Z}_q$
	$\longleftarrow c$	
compute $s = w + cx$		
	$s \longrightarrow$	
		check $g^s \stackrel{?}{=} ty^c$

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

Completeness and Soundness

- Completeness
  - means that an honest prover succeeds in convincing an honest verifier
- Soundness
  - means that a dishonest prover does not succeed in convincing the verifier of a false statement

Schnorr Protocol (1991)

- The Schnorr protocol generates a proof transcript

$$(t, c, s) = ZKP\{(x) : y = g^x\}$$

  - → Commitment $t \in G_q$
  - → Challenge $c \in \mathbb{Z}_q$
  - → Response $s \in \mathbb{Z}_q$
- Completeness: Knowing x allows computing $s = w + cx$, which implies $g^s = g^{w+cx} = g^w g^{cx} = g^w (g^x)^c = ty^c$
- Soundness: Without knowing x, finding a suitable value s that satisfies $g^s = ty^c$ is equivalent to solving DL (the probability of guessing the right s is $\frac{1}{q}$)
- Zero-Knowledge: for every $x' \neq x$, the is a value w' satisfying $s = w' + cx'$

Comparing Interactive and Non-Interactive Proof of Knowledge

(Fiat-Shamir)

Non-Interactive Proof of Knowledge

Non-Interactive Proof of Knowledge

- An interactive proof of knowledge is non-transferable
  - → Only the verifier knows that c has been picked *after* receiving t
  - → The verifier could simulate a proof transcript by first picking arbitrary values c and s, and use them to derive a matching t
  - → Therefore, the proof is only convincing to the verifier
- The goal of a non-interactive proof of knowledge is to produce a transferable proof that is convincing to everyone
- A general way of constructing non-interactive proofs from interactive proofs is due to Fiat and Shamir (1986)
  - → The challenge c is computed using a hash function
  - → . . . with the public values and the commitments as inputs
- Non-interactive proofs are generally easier to implement

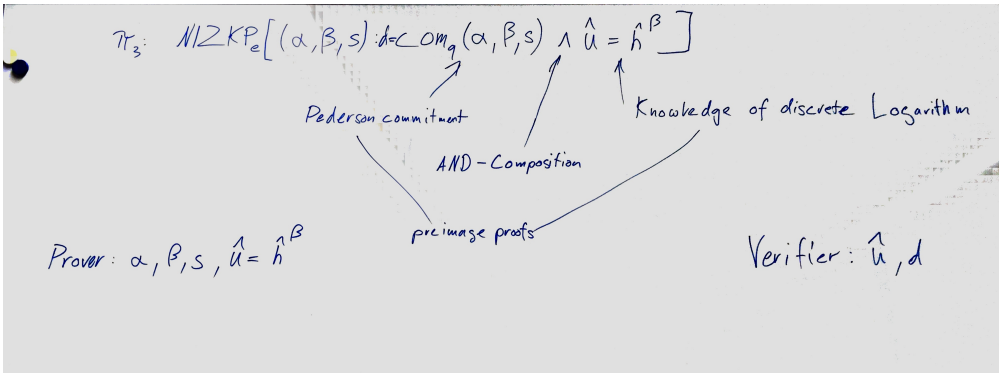
Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

Non-Interactive Schnorr Protocol

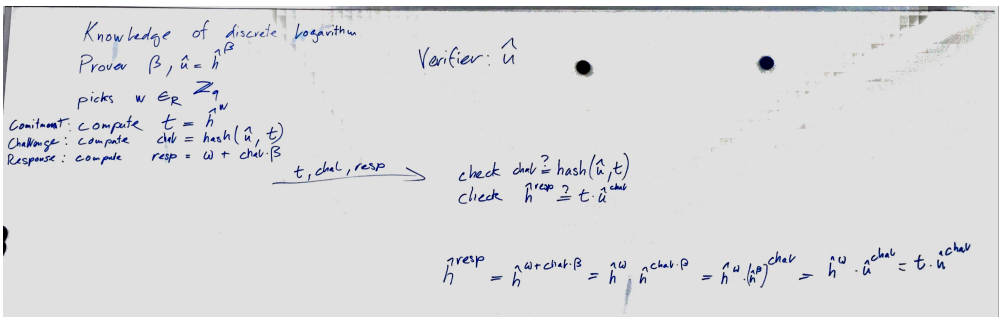
Prover		**Verifier**
(knows x, $y = g^x$)		(knows y)
pick $w \in_R \mathbb{Z}_q$		
compute $t = g^w$		
compute $c = hash(y, t)$		
compute $s = w + cx$		
	$\xrightarrow{t,c,s}$	
		check $c \stackrel{?}{=} hash(y, t)$
		check $g^s \stackrel{?}{=} ty^c$

Proof π_3

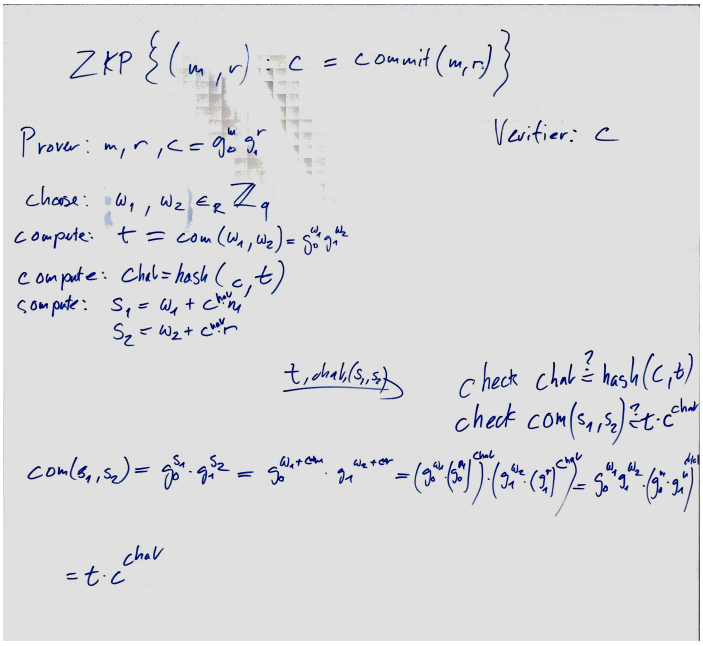
π_3


π_3: $NIZKP_e[(\alpha,\beta,s): d=com_q(\alpha,\beta,s) \wedge \hat{u}=\hat{h}^{\beta}]$
Pederson commitment
AND-Composition
Knowledge of discrete Logarithm
preimage proofs
Prover: $\alpha,\beta,s,\hat{u}=\hat{h}^{\beta}$
Verifier: $\hat{u},d$

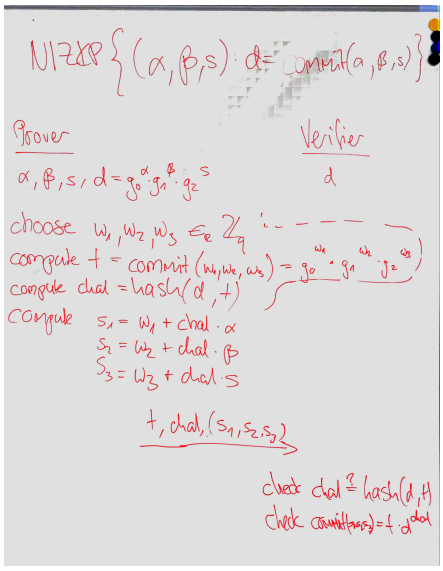
π_3


Knowledge of discrete Logarithm
Prover $\beta, \hat{u}=\hat{h}^{\beta}$
picks $w \in_R \mathbb{Z}_q$
Commitment: compute $t=\hat{h}^{w}$
Challenge: compute $chal=hash(\hat{u},t)$
Response: compute $resp=\omega+chal\cdot\beta$
Verifier: $\hat{u}$
$t, chal, resp$
check $chal \stackrel{?}{=} hash(\hat{u},t)$
check $\hat{h}^{resp} \stackrel{?}{=} t\cdot\hat{u}^{chal}$
$\hat{h}^{resp}=\hat{h}^{\omega+chal\cdot\beta}=\hat{h}^{\omega}\cdot\hat{h}^{chal\cdot\beta}=\hat{h}^{\omega}\cdot(\hat{h}^{\beta})^{chal}=\hat{h}^{\omega}\cdot\hat{u}^{chal}=t\cdot\hat{u}^{chal}$

π_3 > Commitment with 2 Variables



π_3 > Commitment with 3 Variables



AND-Composition of multiple Proofs

AND-Composition of Multiple Proofs

- For $1 \leq i \leq n$, let
  - → $f_i : X_i \to Y_i$ be a one-way homomorphism
  - → $x_i \in X_i$ be a secret values known to the prover
  - → $y_i = f_i(x_i) \in Y_i$ be publicly known
- Proving knowledge of $x_1, \ldots, x_n$ can be done by a single composed preimage proof

$$ZKP\{(x_1, \ldots, x_n) : y_1 = f_1(x_1) \wedge \cdots \wedge y_n = f_n(x_n)\}$$
$$= ZKP\{(x_1, \ldots, x_n) : (y_1, \ldots, y_n) = f(x_1, \ldots, x_n)\}$$

  for $f(x_1, \ldots, x_n) = (f_1(x_1), \ldots, f_n(x_n))$
- A non-interactive composed proof is different from n single proofs $ZKP\{(x_1) : y_1 = f_1(x_1)\}$, …, $ZKP\{(x_n) : y_n = f_n(x_n)\}$, because it implies knowledge by the same party

General Preimage Equality Proofs

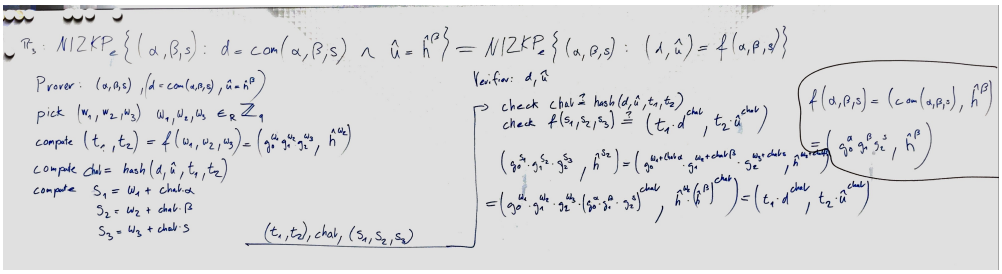
General Preimage Equality Proofs

- Let $x \in X$ be a secret values known to the prover
- For $1 \leq i \leq n$, let
  - → $f_i : X \to Y_i$ be a one-way homomorphism
  - → $y_i = f_i(x) \in Y_i$ be publicly known
- Proving that $y_1, \ldots, y_n$ have the same preimage x is a special case of an AND-composition

$$ZKP\{(x) : y_1 = f_1(x) \wedge \cdots \wedge y_n = f_n(x)\}$$
$$= ZKP\{x) : (y_1, \ldots, y_n) = f(x)\}$$

  for $f(x) = (f_1(x), \ldots, f_n(x))$

π_3



Open Topics

Open Topics

- **Further planning**
  - Deliverables
  - Assessment

D. Präsentation 3, 07.04.2016

Everlasting privacy in E-Voting > Presentation 3

Timo Buerk, Nellen Sebastian

Bern University of Applied Sciences

burkt4@bfh.ch, nells1@bfh.ch

7. April 2016

Overview

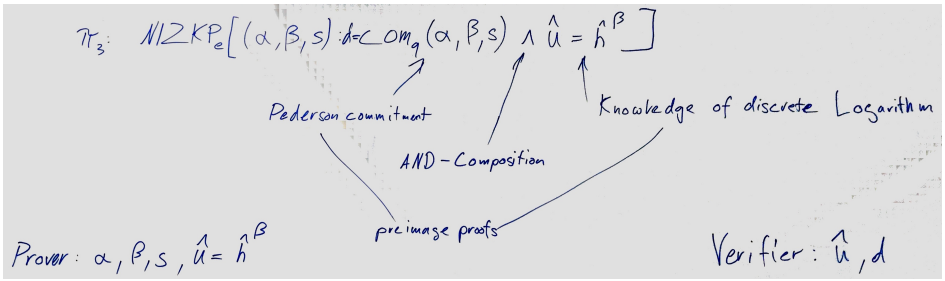
π_3
- blackboard photos
- written in LaTeX

Preimage Equality Proofs
- blackboard photos
- written in LaTeX

Code-Demo: Implementing π_3 with UniCrypt

What's next?

π_3 > blackboard photos



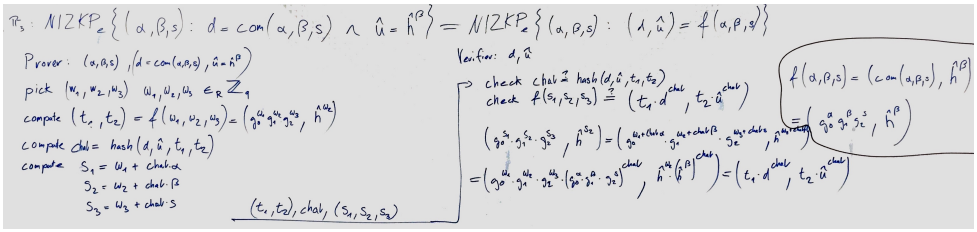
π_3 > written in LaTeX

$$\pi_3 : NIZKP_e[(\alpha, \beta, s)] : d = com_q(\alpha, \beta, s) \wedge \hat{u} = \hat{h}^{\beta}] \qquad (1)$$

Prover:
α, β, s, $\hat{u} = \hat{h}^{\beta}$

Verifier
$\hat{u}$, d

π_3 : AND-composition > blackboard photos



π_3 : AND-composition > written in LaTeX– with colums

$$\begin{aligned} \pi_3 : NIZKP_e[(\alpha, \beta, s)] : d = com_q(\alpha, \beta, s) \wedge \hat{u} = \hat{h}^{\beta}] \\ = NIZKP_e[(\alpha, \beta, s) : (d, \hat{u}) = f(\alpha, \beta, s)] \end{aligned} \tag{2}$$

Prover: α, β, s, $\hat{u} = \hat{h}^{\beta}$

pick $(w1, w2, w3) \in_R \mathbb{Z}_q$
compute **(t1, t2)** =
$f(w1, w2, w3) = (g_0^{w1} g_1^{w2} g_2^{w3}, \hat{h}^{w2})$
compute **chal** = hash (d, $\hat{u}$, t1, t2)
compute

- **s1** $= w1 + chal * \alpha$
- **s2** $= w2 + chal * \beta$
- **s3** $= w3 + chal * s$

Verifier: $\hat{u}$, d

check chal ?= hash (d, $\hat{u}$, t1, t2)
check f(s1, s2, s3) ?=
$(t1 * d^{chal}, t2 * \hat{u}^{s2})$

$$(t_1, t_2), chal, (s1, s2, s3) \longrightarrow$$

π_3 : AND-composition > written in LaTeX−*try*#2

$$\begin{aligned}\pi_3 : NIZKP_e[(\alpha,\beta,s)] : d = com_q(\alpha,\beta,s) \wedge \hat{u} = \hat{h}^{\beta}] \\ = NIZKP_e[(\alpha,\beta,s) : (d,\hat{u}) = f(\alpha,\beta,s)]\end{aligned} \tag{3}$$

Prover: α, β, s, $\hat{u} = \hat{h}^{\beta}$

pick $(w1, w2, w3) \in_R \mathbb{Z}_q$
compute **(t1, t2)** $= f(w1, w2, w3) = (g_0^{w1} g_1^{w2} g_2^{w3}, \hat{h}^{w2})$
compute **chal** = hash (d, $\hat{u}$, t1, t2)
compute

- **s1** $= w1 + chal * \alpha$
- **s2** $= w2 + chal * \beta$
- **s3** $= w3 + chal * s$

$$(t_1, t_2), chal, (s1, s2, s3) \longrightarrow$$

Verifier: $\hat{u}$, d

check chal ?= hash (d, $\hat{u}$, t1, t2)
check f(s1, s2, s3) ?= $(t1 * d^{chal}, t2 * \hat{u}^{s_2})$

π_3 : AND-composition > written in LaTeX− with minipage

$$\begin{aligned}\pi_3 : NIZKP_e[(\alpha,\beta,s)] : d = com_q(\alpha,\beta,s) \wedge \hat{u} = \hat{h}^{\beta}] \\ = NIZKP_e[(\alpha,\beta,s) : (d,\hat{u}) = f(\alpha,\beta,s)]\end{aligned} \tag{4}$$

Prover: α, β, s, $\hat{u} = \hat{h}^{\beta}$

pick $(w1, w2, w3) \in_R \mathbb{Z}_q$
compute **(t1, t2)** $= f(w1, w2, w3) = (g_0^{w1} g_1^{w2} g_2^{w3}, \hat{h}^{w2})$
compute **chal** = hash (d, $\hat{u}$, t1, t2)
compute

- **s1** $= w1 + chal * \alpha$
- **s2** $= w2 + chal * \beta$
- **s3** $= w3 + chal * s$

Verifier: $\hat{u}$, d

check chal ?= hash (d, $\hat{u}$, t1, t2)
check f(s1, s2, s3) ?= $(t1 * d^{chal}, t2 * \hat{u}^{s_2})$

$$(t_1, t_2), chal, (s1, s2, s3) \longrightarrow$$

Code-Demo: Implementing π_3 with UniCrypt

```
131
132     //in order to put the two functions in the same ProofSystem they need to have the same Domains.
133
134     //define a selection Function with the same Domain as the PedersenCommitmentFunction that simply selects the Beta Element and r
135     ProductSet space = (ProductSet) pedersenFunction.getDomain();
136     Function adapterFunction = SelectionFunction.getInstance(space, 0,1); //0,1 : Pfad für selektiertes Element 0 für MessageElemen
137     // We chain the selection Function and the discreteLogFunction together. So the output of the selection function acts as input
138     Function adaptedDiscreteLogFunction = CompositeFunction.getInstance(adapterFunction,discreteLogFunction);
139
140     //Create a EqualityPreimageProofSystem with the PedersenCommitmentFunction and the adapted DicreteLogFunction
141
142     EqualityPreimageProofSystem proofSystem = EqualityPreimageProofSystem.getInstance(pedersenFunction, adaptedDiscreteLogFunction);
143
144     //choose alpha as 16 and beta as 42 (normaly alpha and beta are chosen at random)
145     Element alpha = discreteLogCommitmentScheme.getMessageSpace().getElement(16);
146     Element beta = discreteLogCommitmentScheme.getMessageSpace().getElement(42);
147     // Create a randomElement for the commitment
148     Element rElement = pedersenCommitmentScheme.getRandomizationSpace().getRandomElement();
149
150     // Create the Elements containing the private and public Inputs for the Pi3 NIZKP
151     Pair privateInput = Pair.getInstance(Pair.getInstance(alpha, beta), rElement);
152     Element result0 = adapterFunction.apply(privateInput);
153     Element result1 = pedersenFunction.apply(privateInput);
154     Element result2 = adaptedDiscreteLogFunction.apply(privateInput);
155     Pair publicInput = Pair.getInstance(result1, result2);
156
157     // Generate the Pi3 NIZKP for the (alpha, beta, r)
158     Triple proof = proofSystem.generate(privateInput, publicInput);
159
160     // Verifiy the proof with the publicinputs
161     Boolean result = proofSystem.verify(proof, publicInput);
```

What's next?

- Implement π_3 on our own
- Integrate our own π_3 in Vaadin Framework

The End

E. Präsentation 4, 12.05.2016

Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Everlasting privacy in E-Voting

Theory and Implementation aspects of π_1 & π_2 | Documentation

- Modul 7302 «Projekt 2»

Agenda

- A try on using unicrypt with Vaadin
- Analysis of P. Locher's implementation of π_2
- Analysis of theory π_1
- Analysis of theory π_2
- Complexity of π_1 & π_2
- Open Topics

Screenshots based on theory of P. Locher's and R. Haenni's Paper «Veriable Internet Elections with Everlasting Privacy and Minimal Trust» 2015

A try on using unicrypt withVaadin

Findings

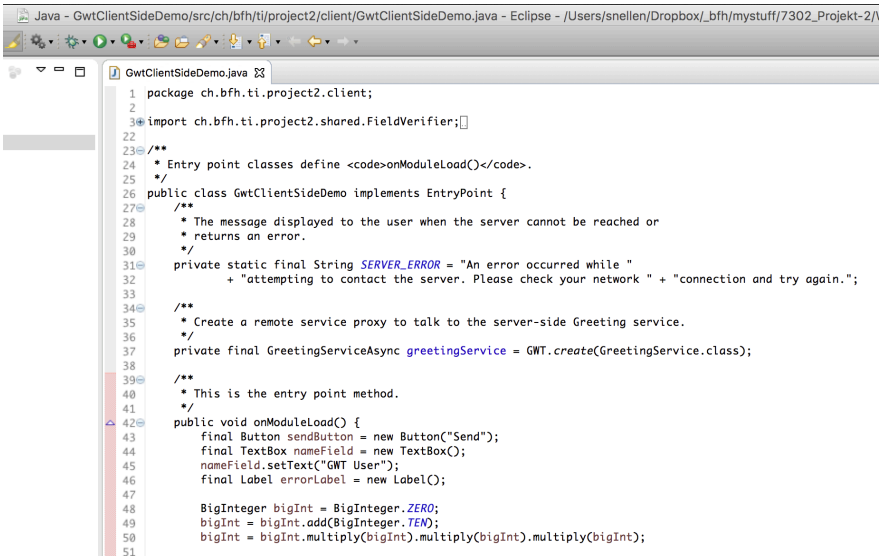
- Vaadin uses GWT for client-side applications
- Decided to use GWT for client-side application without Vaadin overhead
- GWT-only client-side application can be built in Eclipse thanks to Google Plugin, it compiles even with BigInteger variables
- GWT-only client-side application in Netbeans can't be realized that easy as there's no Google Plugin or similar, gwt4NB doesn't help much

Approaches

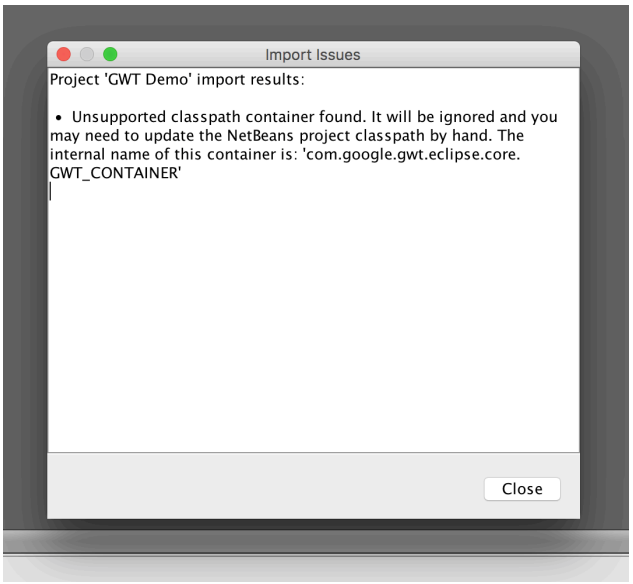
- Approach 1
  - Create a GWT ClientSide Demo in Eclipse
  - Import GWT Eclipse Demo into NetBeans,
    as unicrypt functions “out-of-the-box” in NetBeans
  - Integrate Timo’s π_3 Implementation in demo application
- Approach 2
  - Create a Create a GWT ClientSide Demo in NetBeans
  - Integrate Timo’s π_3 Implementation in demo application
- Approach 3
  - Try the same thing as above but with implementing in Vaadin
    from scratch > not enough time…

Approach 1: GWT ClientSide Eclipse Demo

- http://127.0.0.1:8888/GwtClientSideDemo.html



Approach 1: Import of Eclipse project into NB



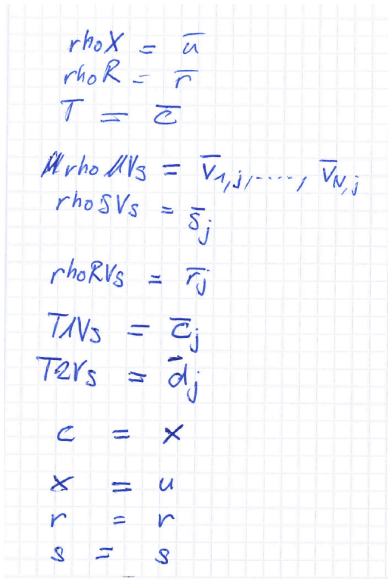
Approach 2: GWT ClientSide Application in NB

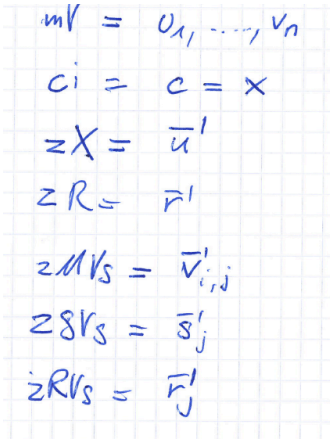
- Problems with dependencies > unresolvable
  - Manually changed dependencies
  - Faked version numbers
  - Installing dependencies locally

Analysis of π_2 implementation

Analysis of P. Locher's implementation of π_2

Variable-Translation





Analysis of π_1

Summary of π_1

Public Input: $c = \text{com}_p(u, r) \in \mathcal{G}_p$, $P(X) = \sum_{i=0}^{M} a_i X^i \in \mathbb{Z}_p[X]$
Secret Input: $u, r \in \mathbb{Z}_p$
Generation:

1. For $j = 1, \ldots, m$, pick $r_j \in_R \mathbb{Z}_p$ and compute $c_j = \text{com}_p(u^{2^j}, r_j)$.
2. For $j = 0, \ldots, m$, pick $\bar{a}_j, \bar{r}_j \in_R \mathbb{Z}_p$ and compute $\bar{c}_j = \text{com}_p(\bar{a}_j, \bar{r}_j)$.
3. Compute new polynomial

$$\tilde{P}(X) = \sum_{j=0}^{m} \tilde{a}_j X^j = \sum_{i=0}^{M} a_i \prod_{j=0}^{m} (u^{2^j} X + \bar{a}_j)^{i[j]} X^{1-i[j]} \in \mathbb{Z}_p[X]$$

   of degree m. For $j = 0, \ldots, m$, pick $\tilde{r}_j \in_R \mathbb{Z}_p$ and compute $\tilde{c}_j = \text{com}_p(\tilde{a}_j, \tilde{r}_j)$.
4. For $j = 0, \ldots, m-1$, compute $\hat{a}_j = u^{2^j} \bar{a}_j$, pick $\hat{r}_j \in_R \mathbb{Z}_p$, and compute $\hat{c}_j = \text{com}_p(\hat{a}_j, \hat{r}_j)$.
5. Compute $x = h(c, a_0, \ldots, a_M, c_1, \ldots, c_m, \bar{c}_0, \ldots, \bar{c}_m, \tilde{c}_0, \ldots, \tilde{c}_m, \hat{c}_0, \ldots, \hat{c}_{m-1})$.
6. For $j = 0, \ldots, m$, compute $\bar{a}'_j = \bar{a}_j + x u^{2^j}$.
7. For $j = 0, \ldots, m$, compute $\bar{r}'_j = \bar{r}_j + x r_j$.
8. For $j = 0, \ldots, m-1$, compute $\hat{r}'_j = \hat{r}_j + x r_{j+1} - b_j r_j$.
9. Compute $\tilde{r}' = \sum_{j=0}^{m} \tilde{r}_j x^j$.

Transcript:
$(c_1, \ldots, c_m, \bar{c}_0, \ldots, \bar{c}_m, \tilde{c}_0, \ldots, \tilde{c}_m, \hat{c}_0, \ldots, \hat{c}_{m-1}, \bar{a}'_0, \ldots, \bar{a}'_m, \bar{r}'_0, \ldots, \bar{r}'_m, \hat{r}'_0, \ldots, \hat{r}'_{m-1}, \tilde{r}')$
Verification:

1. Compute $x = h(c, a_0, \ldots, a_M, c_1, \ldots, c_m, \bar{c}_0, \ldots, \bar{c}_m, \tilde{c}_0, \ldots, \tilde{c}_m, \hat{c}_0, \ldots, \hat{c}_{m-1})$.
2. For $j = 0, \ldots, m$, check $c_j^x \bar{c}_j = \text{com}_p(\bar{a}'_j, \bar{r}'_j)$.
3. For $j = 0, \ldots, m-1$, check $c_{j+1}^x \hat{c}_j = c_j^{\bar{a}'_j} \cdot \text{com}_p(0, \hat{r}'_j)$.
4. Check

$$\prod_{j=0}^{m} \tilde{c}_j^{x^j} = \text{com}_p\left(\sum_{i=0}^{M} a_i \prod_{j=0}^{m} \bar{a}_j'^{\,i[j]} x^{1-i[j]}, \tilde{r}' \right).$$

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

Analysis of π_2

Summary of π_2

Public Input: $c = \mathrm{com}_p(u, r) \in \mathcal{G}_p$, $d = \mathrm{com}_q(v_1, \ldots, v_N, s) \in \mathbb{G}_q$
Secret Input: $u, r \in \mathbb{Z}_p$, $v_1, \ldots, v_N, s \in \mathbb{Z}_q$
Generation:

1. Pick $\bar{u}, \bar{r} \in_R \mathbb{Z}_p$ and compute $\bar{c} = \mathrm{com}_p(\bar{u}, \bar{r})$.
2. For $j = 1, \ldots, K$,
   (a) pick $\bar{v}_{1,j}, \ldots, \bar{v}_{N,j} \in_R \mathbb{Z}_q$ and compute $\bar{u}_j = h_1^{\bar{v}_{1,j}} \cdots h_N^{\bar{v}_{N,j}}$,
   (b) pick $\bar{r}_j \in_R \mathbb{Z}_p$ and compute $\bar{c}_j = \mathrm{com}_p(\bar{u}_j, \bar{r}_j)$,
   (c) pick $\bar{s}_j \in_R \mathbb{Z}_q$ and compute $\bar{d}_j = \mathrm{com}_q(\bar{v}_{1,j}, \ldots, \bar{v}_{N,j}, \bar{s}_j)$.
3. Compute $x = h(c, d, \bar{c}, \bar{c}_1, \ldots, \bar{c}_k, \bar{d}_1, \ldots, \bar{d}_k)$.
4. Compute $\bar{u}' = \bar{u} - xu$ and $\bar{r}' = \bar{r} - xr$.
5. For $j = 1, \ldots, K$,
   (a) for $i = 1, \ldots, N$, compute $\bar{v}'_{i,j} = \bar{v}_{i,j} - x[j]v_i$,
   (b) compute $\bar{r}'_j = \bar{r}_j - x[j] \cdot \mathrm{com}_q(\bar{v}'_{1,j}, \ldots, \bar{v}'_{N,j}, r)$,
   (c) compute $\bar{s}'_j = \bar{s}_j - x[j]s$.

Transcript:
$(\bar{c}, \bar{c}_1, \ldots, \bar{c}_k, \bar{d}_1, \ldots, \bar{d}_k, \bar{u}', \bar{r}', \bar{v}'_{1,1}, \ldots, \bar{v}'_{N,K}, \bar{r}'_1, \ldots, \bar{r}'_k, \bar{s}'_1, \ldots, \bar{s}'_k)$
Verification:

1. Compute $x = h(c, d, \bar{c}, \bar{c}_1, \ldots, \bar{c}_k, \bar{d}_1, \ldots, \bar{d}_k)$.
2. Check $\bar{c} = c^x \cdot \mathrm{com}_p(\bar{u}', \bar{r}')$.
3. For $j = 1, \ldots, K$,
   (a) check $\bar{d}_j = d^{x[j]} \cdot \mathrm{com}_q(\bar{v}'_{1,j}, \ldots, \bar{v}'_{N,j}, \bar{s}'_j)$,
   (b) compute $\bar{u}'_j = h_1^{\bar{v}'_{1,j}} \cdots h_N^{\bar{v}'_{N,j}}$, and check

$$\bar{c}_j = \begin{cases} \mathrm{com}_p(\bar{u}'_j, \bar{r}'_j), & \text{if } x[j] = 0, \\ c^{\bar{u}'_j} \cdot \mathrm{com}_p(0, \bar{r}'_j), & \text{if } x[j] = 1. \end{cases}$$

Complexity of π_1 & π_2

Complexity of π_1 & π_2

- π_1
  - Complexity depending on quantity of voters
- π_2
  - Not really complex – by itself
  - Becoming complex, as security comes with multiple computations → K

Open Topics

Open Topics

- **Further planning**
  - **Documentation** will be written in the next 3-4 weeks
    - Report-like (approach, findings)
    - Notes, Photographs and Presentations in appendix
    - Deadline?
  - Project **presentation** "Thesis-like"
    - Suggestion 1: Thursday, 16.06.2016 in Biel (in the evening, the Day before "Finaltag")
    - Suggestion 2: 2nd exam week (11th – 15th of July 2016)
    - Suggestion 3: Beginning fall semester, as a repetition and preparation for the Bachelor Thesis

F. Verifikation π_1

Nachfolgender Scan zeigt die Analyse des Beweises π_1

π_1: Verification:

2. $c_j^x \bar{c}_j = com_p(\bar{a}_j', \bar{r}_j')$

$$c_j^x \bar{c}_j = g_0^{u^{2^j} x} g_1^{r_j x} \cdot g_0^{\bar{a}_j} g_1^{\bar{r}_j} = g_0^{x u^{2^j}} \cdot g_1^{x r_j} g_0^{\bar{a}_j} g_1^{\bar{r}_j} = g_0^{\bar{a}_j + x u^{2^j}} \cdot g_1^{\bar{r}_j + x r_j}$$

$$= g_0^{\bar{a}_j'} g_1^{\bar{r}_j'} = com_p(\bar{a}_j, \bar{r}_j')$$

3. $c_{j+1}^x \cdot \hat{c}_j = c_j^{\bar{a}_j'} \cdot com_p(0, \hat{r}_j')$

$$c_{j+1}^x \cdot \hat{c}_j = \left(g_0^{u^{2^{j+1}}} \cdot g_0^{r_{j+1}} \right)^x \cdot g_0^{\hat{a}_j} g_1^{\hat{r}_j} = g_0^{x u^{2^{j+1}}} \cdot g_1^{x r_{j+1}} \cdot g_0^{u^{2^j} \hat{a}_j} \cdot g_1^{\hat{r}} =$$

$$= g_0^{\bar{a}_j u^{2^j} + x u^{2^{j+1}}} \cdot g_1^{x r_{j+1} + \hat{r}} = g_0^{\bar{a}_j u^{2^j} + x u^{2^j \cdot 2^1}} \cdot g_1^{x r_{j+1} + \hat{r}} = g_0^{\bar{a}_j u^{2^j} + x (u^{2^j})^2} \cdot$$

$$g_1^{x r_{j+1} + \hat{r}} = g_0^{(\bar{a}_j + x u^{2^j}) \cdot u^{2^j}} g_1^{\hat{r}_j + x r_{j+1}} = g_0^{\bar{a}_j' \cdot u^{2^j}} \cdot g_1^{\hat{r}_j + x r_{j+1}}$$

$$c_j^{\bar{a}_j'} \cdot com_p(0, \hat{r}_j') = g_0^{\bar{a}_j' u^{2^j}} \cdot g_1^{\bar{a}_j' r_j} \cdot g_0^0 \cdot g_1^{\hat{r}_j'} = g_0^{\bar{a}_j' u^{2^j}} \cdot g_1^{\bar{a}_j' r_j + \hat{r}_j'} =$$

$$g_0^{\bar{a}_j' u^{2^j}} \cdot g_1^{\hat{r}_j + x r_{j+1} - b_j r_j + \bar{a}_j' r_j} = g_0^{\bar{a}_j' u^{2^j}} \cdot g_1^{\hat{r}_j + x r_{j+1}}$$

$b_j = \bar{a}_j'$?

4. $\prod_{j=0}^{m} \tilde{c}_j^{x^j} = com_p\left(\sum_{i=0}^{M} a_i \prod_{j=0}^{m} \bar{a}_j^{i[j]} \cdot x^{1-i[j]}, \tilde{r}' \right)$

$$\prod_{j=0}^{m} \tilde{c}_j^{x^j} = \prod_{j=0}^{m} \left(g_0^{\tilde{a}_j} \cdot g_1^{\tilde{r}_j} \right)^{x^j} = g_0^{\sum_{j=0}^{m} \tilde{a}_j x^j} \cdot g_1^{\sum_{j=0}^{m} \tilde{r}_j x^j} \overset{\text{Formel } \tilde{r}'}{=} g_0^{\sum_{j=0}^{m} \tilde{a}_j x^j} \cdot g_1^{\tilde{r}'}$$

$$= g_0^{\sum_{i=0}^{M} a_i \prod_{j=0}^{m} (u^{2^j} x + \bar{a}_j)^{i[j]} \cdot x^{1-i[j]}} g_1^{\tilde{r}'} = com_p\left(\sum_{i=0}^{M} a_i \prod_{j=0}^{m} \bar{a}_j^{i[j]} \cdot x^{1-i[j]}, \tilde{r}' \right)$$

↑ Formel $\tilde{P}(x)$

G. Verifikation π_2

Nachfolgender Scan zeigt die Analyse des Beweises π_2. Auf der nachfolgenden Seite befindet sich eine Übersetzungstabelle zwischen dem Beweis und der Unicrypt Implementation von P. Locher.

π_2 Verification

2. $\bar{c} = c^x \cdot \mathrm{com}_p(\bar{u}', \bar{r}')$

$c^x \cdot \mathrm{com}_p(\bar{u}', \bar{r}') = g_0^{u^x} g_1^{r^x} \cdot g_0^{\bar{u}'} \cdot g_1^{\bar{r}'} = g_0^{ux} g_1^{rx} \cdot g_0^{\bar{u}-ux} g_1^{\bar{r}-rx}$

$= g_0^{\bar{u}} g_1^{\bar{r}} = \bar{c}$

3a) $\bar{d}_j = d^{x[j]} \cdot \mathrm{com}_q(\bar{v}'_{1,j}, \dots, \bar{v}'_{N,j}, \bar{s}'_j)$

Case $x[j] = 0$:

da $\bar{v}'_{i,j} = \bar{v}_{i,j}$ für $x[j]=0$

$\bar{s}'_j = \bar{s}_j$ für $x[j]=0$

$\bar{d}_j = \mathrm{com}_q(\bar{v}_{1,j}, \dots, \bar{v}_{N,j}, \bar{s}_j) = \mathrm{com}_q(\bar{v}'_{1,j}, \dots, \bar{v}'_{N,j}, \bar{s}'_j)$

$= d^0 \mathrm{com}_q(\bar{v}'_{1,j}, \dots, \bar{v}'_{N,j}, \bar{s}'_j) = d^{x[j]} \mathrm{com}_q(\bar{v}'_{1,j}, \dots, \bar{v}'_{N,j}, \bar{s}'_j)$

Case $x[j] = 1$:

$\bar{d}_j = \mathrm{com}_q(\bar{v}_{1,j}, \dots, \bar{v}_{N,j}, \bar{s}_j)$

$d \cdot \mathrm{com}_q(\bar{v}'_{1,j}, \dots, \bar{v}'_{N,j}, \bar{s}'_j) = h_1^{v_1} \cdots h_N^{v_N} \cdot h_s^{s} \cdot h_1^{\bar{v}_{1,j} - v_1} \cdots h_N^{\bar{v}_{N,j} - v_N} \cdot h_s^{\bar{s}_j - s}$

$= \mathrm{com}_q(\bar{v}_{1,j}, \dots, \bar{v}_{N,j}, \bar{s}_j)$

3b) $\bar{u}'_j = h_1^{\bar{v}'_{1,j}} \cdots h_N^{\bar{v}'_{N,j}}$

case $x[j] = 0$ $\quad \bar{c}_j = \mathrm{com}_p(\bar{u}'_j, \bar{r}'_j)$

$\bar{u}_j = \bar{u}'_j$ und $\bar{r}_j = \bar{r}'_j$ dafür $x[j]=0$

$\bar{c}_j = \mathrm{com}_p(\bar{u}_j, \bar{r}_j) = \mathrm{com}_p(\bar{u}'_j, \bar{r}'_j)$

case $x[j] = 1$ $\quad \bar{c}_j = c^{\bar{u}'_j} \cdot \mathrm{com}_p(0, \bar{r}'_j)$

$\bar{c}_j = \mathrm{com}_p(\bar{u}_j, \bar{r}_j) = g_0^{\bar{u}_j} g_1^{\bar{r}_j}$

$c^{\bar{u}'_j} \mathrm{com}_p(0, \bar{r}'_j) = (g_0^u g_1^r)^{\bar{u}'_j} \cdot g_0^0 g_1^{\bar{r}'_j} = g_0^{u \cdot \bar{u}'_j} \cdot g_1^{r \bar{u}'_j} \cdot g_1^{\bar{r}_j} = \mathrm{com}_p(\bar{v}'_1, \dots, \bar{v}'_N, r)$

$= g_0^{u \bar{u}'_j} \cdot g_1^{r \bar{u}'_j} \cdot g_1^{\bar{r}_j - \bar{u}'_j \cdot h_s^r} = g_0^{h_1^{v_1} \cdots h_N^{v_N} \cdot h_1^{\bar{v}_{1,j}} \cdots h_N^{\bar{v}_{N,j}}} \cdot g_1^{\bar{r}_j - \bar{u}'_j h_s^{r + r\bar{u}_j}} = g_0^{\bar{u}_j} \cdot g_1^{\bar{r}_j - \bar{u}'_j h_s^{r + r\bar{u}_j}}$

$h_s = r$

$= g_0^{\bar{u}_j} \cdot g_1^{\bar{r}_j} = \mathrm{com}_q(\bar{u}_j, \bar{r}_j) = \bar{c}_j$

rhoX = $\bar{u}$

rhoR = $\bar{r}$

T = $\bar{c}$

rhoMVs = $\bar{v}_{1,j}, \ldots, \bar{v}_{N,j}$

rhoSVs = $\bar{s}_j$

rhoRVs = $\bar{r}_j$

TMVs = $\bar{c}_j$

TRVs = $\bar{d}_j$

c = x

x = u

r = r

s = s

mV = $v_1, \ldots, v_n$

ci = c = x

zX = $\bar{u}'$

zR = $\bar{r}'$

zMVs = $\bar{v}'_{i,j}$

zSVs = $\bar{s}'_j$

zRVs = $\bar{r}'_j$

H. Implementation π_3 mittels Unicrypt

Zum besseren Verständnis des Beweises π_3 und zum Kennenlernen der Unicrypt-Library, wurde ersterer mit Hilfe von Unicrypt implementiert.

```java

package ch.bfh.unicrypt.crypto.proofsystem;

import ch.bfh.unicrypt.Example;
import ch.bfh.unicrypt.UniCryptException;
import ch.bfh.unicrypt.crypto.proofsystem.classes.EqualityPreimageProofSystem;

import
ch.bfh.unicrypt.crypto.schemes.commitment.classes.GeneralizedPedersenCommitmentScheme;
import
ch.bfh.unicrypt.crypto.schemes.commitment.classes.DiscreteLogarithmCommitmentScheme;
import ch.bfh.unicrypt.math.algebra.general.classes.BooleanElement;
import ch.bfh.unicrypt.math.algebra.general.classes.Pair;
import ch.bfh.unicrypt.math.algebra.general.classes.ProductSet;
import ch.bfh.unicrypt.math.algebra.general.classes.Triple;
import ch.bfh.unicrypt.math.algebra.general.classes.Tuple;
import ch.bfh.unicrypt.math.algebra.general.interfaces.CyclicGroup;
import ch.bfh.unicrypt.math.algebra.general.interfaces.Element;
import ch.bfh.unicrypt.math.algebra.multiplicative.classes.GStarModSafePrime;
import ch.bfh.unicrypt.math.function.classes.CompositeFunction;
import ch.bfh.unicrypt.math.function.classes.SelectionFunction;
import ch.bfh.unicrypt.math.function.interfaces.Function;

/**
 *
 * @author t.buerk
 */


public class Pi3Example {
        public static void example1() {

        //DiscreteLogarithm part of Pi3 NIZKP

        // Create cyclic group G_q (modulo 167) and wth generator=98
        GStarModSafePrime cyclicGroup = GStarModSafePrime.getInstance(167);
        // generator = h-dach
        Element generator = cyclicGroup.getElement(98);

        // Create DiscreteLogarithScheme with h-dach as generator
        DiscreteLogarithmCommitmentScheme discreteLogCommitmentScheme =
        DiscreteLogarithmCommitmentScheme.getInstance(generator);

        //choose beta as 42 (normaly beta is chosen at random)
        Element beta = discreteLogCommitmentScheme.getMessageSpace().getElement(42);

        // Create commitment to beta = û
        Element commitment = discreteLogCommitmentScheme.commit(beta);

        // Decommit = Reveal
        BooleanElement result = discreteLogCommitmentScheme.decommit(beta,
        commitment);

        Example.printLine("Cylic Group", cyclicGroup);
        Example.printLine("Message", beta);
        Example.printLine("Commitment", commitment);
        Example.printLine("Result", result);

    }

    public static void example2() throws UniCryptException {

        // PedersenCommitment with 2 messageGenerators for Pi3 NIZKP

        // Create cyclic group G_q (modulo 167) with 3 generators( 2 message
        generators an 1 random generator) generators are chosen at random
        CyclicGroup cyclicGroup = GStarModSafePrime.getInstance(167);
        Element generatorAlpha = cyclicGroup.getRandomGenerator();
        Element generatorBeta = cyclicGroup.getRandomGenerator();
        Element generatorR = cyclicGroup.getRandomGenerator();

```

```
67            // Create a PairElement containing the 2 Messeage generators
68            Pair messageGenerators = Pair.getInstance(generatorAlpha,generatorBeta);
69
70            // Create PedersenCommitmentScheme for two message and one random Element
71            GeneralizedPedersenCommitmentScheme pedersenCommitmentScheme =
              GeneralizedPedersenCommitmentScheme.getInstance(generatorR,messageGenerators);
72
73
74            // Create a Pair of messageElements = (alpha, beta) (normaly alpha and beta
              are chosen at random)
75            Tuple messageElements =
              pedersenCommitmentScheme.getMessageSpace().getElementFrom(16,42);
76            // Create a randomElement for the commitment
77            Element rElement =
              pedersenCommitmentScheme.getRandomizationSpace().getRandomElement();
78
79            // Create commitment for alpha and beta
80            Element commitment = pedersenCommitmentScheme.commit(messageElements,
              rElement);
81
82            // Decommit = Reveal
83            BooleanElement result = pedersenCommitmentScheme.decommit(messageElements,
              rElement, commitment);
84
85            Example.printLine("Cylic Group", cyclicGroup);
86            Example.printLine("GeneratorAlpha", generatorAlpha);
87            Example.printLine("GeneratorBeta", generatorBeta);
88            Example.printLine("GeneratorR", generatorR);
89            Example.printLine("Alpha Beta", messageElements);
90            Example.printLine("R", rElement);
91            Example.printLine("Commitment", commitment);
92            Example.printLine("Result", result);
93
94
95        }
96
97        public static void example3() {
98
99            // Full Pi3 NIZKP Example
100
101           //Difine the DiscreteLogarithm part for Pi3 NIZKP and exract
              commitmentfunction
102
103           // Create cyclic group G_q (modulo 167) and wth generator=98
104           GStarModSafePrime cyclicGroup = GStarModSafePrime.getInstance(167);
105           // generator = h-dach
106           Element generator = cyclicGroup.getElement(98);
107
108           // Create DiscreteLogarithScheme with h-dach as generator
109           DiscreteLogarithmCommitmentScheme discreteLogCommitmentScheme =
              DiscreteLogarithmCommitmentScheme.getInstance(generator);
110
111           //Extract the the CommitmentFunction from the commitmentScheme
112           Function discreteLogFunction =
              discreteLogCommitmentScheme.getCommitmentFunction();
113
114
115           //Define PedersenCommitment with 2 messageGenerators for Pi3 NIZKP
116
117           // Create cyclic group G_q (modulo 167) with 3 generators( 2 message
              generators an 1 random generator) generators are chosen at random
118
119           Element generatorAlpha = cyclicGroup.getRandomGenerator();
120           Element generatorBeta = cyclicGroup.getRandomGenerator();
121           Element generatorR = cyclicGroup.getRandomGenerator();
122
123           // Create a PairElement containing the 2 Messeage generators
124           Pair messageGenerators = Pair.getInstance(generatorAlpha,generatorBeta);
125
126           // Create PedersenCommitmentScheme for two message and one random Element
127           GeneralizedPedersenCommitmentScheme pedersenCommitmentScheme =
```

```
            GeneralizedPedersenCommitmentScheme.getInstance(generatorR,messageGenerators);
128
129         //Extract the the CommitmentFunction from the pedersenCommitmentScheme
130         Function pedersenFunction = pedersenCommitmentScheme.getCommitmentFunction();
131
132         //in order to put the two functions in the same ProofSystem they need to
            have the same Domains.
133
134         //define a selection Function with the same Domain as the
            PedersenCommitmentFunction that simply selects the Beta Element and returns it
135         ProductSet space = (ProductSet) pedersenFunction.getDomain();
136         Function adapterFunction = SelectionFunction.getInstance(space, 0,1); //0,1
            : Pfad für selektiertes Element 0 für MessageElements -> 1 für Beta
137         // We chain the selection Function and the discreteLogFunction together. So
            the output of the selection function acts as input for the discreteLogFunction
138         Function adaptedDiscreteLogFunction =
            CompositeFunction.getInstance(adapterFunction,discreteLogFunction);
139
140         //Create a EqualityPreimageProofSystem with the PedersenCommitmentFunction
            and the adapted DicreteLogFunction
141
142         EqualityPreimageProofSystem proofSystem =
            EqualityPreimageProofSystem.getInstance(pedersenFunction,
            adaptedDiscreteLogFunction);
143
144         //choose alpha as 16 and beta as 42 (normaly alpha and beta are chosen at
            random)
145         Element alpha = discreteLogCommitmentScheme.getMessageSpace().getElement(16);
146         Element beta = discreteLogCommitmentScheme.getMessageSpace().getElement(42);
147         // Create a randomElement for the commitment
148         Element rElement =
            pedersenCommitmentScheme.getRandomizationSpace().getRandomElement();
149
150         // Create the Elements containing the private and public Inputs for the Pi3
            NIZKP
151         Pair privateInput = Pair.getInstance(Pair.getInstance(alpha, beta), rElement);
152         Element result0 = adapterFunction.apply(privateInput);
153         Element result1 = pedersenFunction.apply(privateInput);
154         Element result2 = adaptedDiscreteLogFunction.apply(privateInput);
155         Pair publicInput = Pair.getInstance(result1, result2);
156
157         // Generate the Pi3 NIZKP for the (alpha, beta, r)
158         Triple proof = proofSystem.generate(privateInput, publicInput);
159
160         // Verifiy the proof with the publicinputs
161         Boolean result = proofSystem.verify(proof, publicInput);
162
163         Example.printLine("Cylic Group", cyclicGroup);
164         Example.printLine("h-dach", generator);
165         Example.printLine("GeneratorAlpha", generatorAlpha);
166         Example.printLine("GeneratorBeta", generatorBeta);
167         Example.printLine("GeneratorR", generatorR);
168         Example.printLine("Alpha", alpha);
169         Example.printLine("Beta", beta);
170         Example.printLine("R", rElement);
171         Example.printLine("adapter", result0);
172         Example.printLine("discreteLog", result2);
173         Example.printLine("Pedersen", result1);
174
175
176         Example.printLine("Proof", proof);
177         Example.printLine("Check", result);
178
179     }
180
181
182
183     public static void main(final String[] args) {
184         Example.runExamples();
185     }
186 }
```

I. Implementation BigInteger-Exponentation mit Hilfe von GWT

Um die Performance von GWT in Zusammenhang mit BigInteger-Exponentiationen zu evaluieren, wurde im GWT-Beispielprojekt (Google "Web Application Project") die Klasse *GwtClientSideDemo.java* wie folgt angepasst.

```java
package ch.bfh.ti.project2.client;

import ch.bfh.ti.project2.shared.FieldVerifier;

import java.math.BigInteger;
import java.util.Random;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.core.client.GWT;
import com.google.gwt.event.dom.client.ClickEvent;
import com.google.gwt.event.dom.client.ClickHandler;
import com.google.gwt.event.dom.client.KeyCodes;
import com.google.gwt.event.dom.client.KeyUpEvent;
import com.google.gwt.event.dom.client.KeyUpHandler;
import com.google.gwt.user.client.rpc.AsyncCallback;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.DialogBox;
import com.google.gwt.user.client.ui.HTML;
import com.google.gwt.user.client.ui.Label;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.TextBox;
import com.google.gwt.user.client.ui.VerticalPanel;

/**
 * Entry point classes define <code>onModuleLoad()</code>.
 */
public class GwtClientSideDemo implements EntryPoint {
    /**
     * The message displayed to the user when the server cannot be reached or
     * returns an error.
     */
    private static final String SERVER_ERROR = "An error occurred while "
            + "attempting to contact the server. Please check your network " +
            "connection and try again.";

    /**
     * Create a remote service proxy to talk to the server-side Greeting service.
     */
    private final GreetingServiceAsync greetingService =
    GWT.create(GreetingService.class);

    /**
     * This is the entry point method.
     */
    public void onModuleLoad() {
        final Button sendButton = new Button("Send");
        final TextBox nameField = new TextBox();
        nameField.setText("GWT User");
        final Label errorLabel = new Label();

        // create 3 BigInteger objects
        BigInteger bi1, bi2, bi3;

        // create a BigInteger exponent
        BigInteger exponent = new BigInteger(1024, new Random());

        bi1 = new BigInteger(1024, new Random());
        bi2 = new BigInteger(1024, new Random());

        // perform modPow operation on bi1 using bi2 and exp
        bi3 = bi1.modPow(exponent, bi2);

        String str = bi1 + "^" +exponent+ " mod " + bi2 + " is " +bi3;

        // print bi3 value
        System.out.println( str );

        nameField.setText(bi3.toString());

        // We can add style names to widgets
        sendButton.addStyleName("sendButton");
```

```
70
71          // Add the nameField and sendButton to the RootPanel
72          // Use RootPanel.get() to get the entire body element
73          RootPanel.get("nameFieldContainer").add(nameField);
74          RootPanel.get("sendButtonContainer").add(sendButton);
75          RootPanel.get("errorLabelContainer").add(errorLabel);
76
77          // Focus the cursor on the name field when the app loads
78          nameField.setFocus(true);
79          nameField.selectAll();
80
81          // Create the popup dialog box
82          final DialogBox dialogBox = new DialogBox();
83          dialogBox.setText("Remote Procedure Call");
84          dialogBox.setAnimationEnabled(true);
85          final Button closeButton = new Button("Close");
86          // We can set the id of a widget by accessing its Element
87          closeButton.getElement().setId("closeButton");
88          final Label textToServerLabel = new Label();
89          final HTML serverResponseLabel = new HTML();
90          VerticalPanel dialogVPanel = new VerticalPanel();
91          dialogVPanel.addStyleName("dialogVPanel");
92          dialogVPanel.add(new HTML("<b>Sending name to the server:</b>"));
93          dialogVPanel.add(textToServerLabel);
94          dialogVPanel.add(new HTML("<br><b>Server replies:</b>"));
95          dialogVPanel.add(serverResponseLabel);
96          dialogVPanel.setHorizontalAlignment(VerticalPanel.ALIGN_RIGHT);
97          dialogVPanel.add(closeButton);
98          dialogBox.setWidget(dialogVPanel);
99
100         // Add a handler to close the DialogBox
101         closeButton.addClickHandler(new ClickHandler() {
102             public void onClick(ClickEvent event) {
103                 dialogBox.hide();
104                 sendButton.setEnabled(true);
105                 sendButton.setFocus(true);
106             }
107         });
108
109         // Create a handler for the sendButton and nameField
110         class MyHandler implements ClickHandler, KeyUpHandler {
111             /**
112              * Fired when the user clicks on the sendButton.
113              */
114             public void onClick(ClickEvent event) {
115                 sendNameToServer();
116             }
117
118             /**
119              * Fired when the user types in the nameField.
120              */
121             public void onKeyUp(KeyUpEvent event) {
122                 if (event.getNativeKeyCode() == KeyCodes.KEY_ENTER) {
123                     sendNameToServer();
124                 }
125             }
126
127             /**
128              * Send the name from the nameField to the server and wait for a response.
129              */
130             private void sendNameToServer() {
131                 // First, we validate the input.
132                 errorLabel.setText("");
133                 String textToServer = nameField.getText();
134                 if (!FieldVerifier.isValidName(textToServer)) {
135                     errorLabel.setText("Please enter at least four characters");
136                     return;
137                 }
138
139                 // Then, we send the input to the server.
140                 sendButton.setEnabled(false);
```

```
141                     textToServerLabel.setText(textToServer);
142                     serverResponseLabel.setText("");
143                     greetingService.greetServer(textToServer, new
                        AsyncCallback<String>() {
144                         public void onFailure(Throwable caught) {
145                             // Show the RPC error message to the user
146                             dialogBox.setText("Remote Procedure Call - Failure");
147                             serverResponseLabel.addStyleName("serverResponseLabelError");
148                             serverResponseLabel.setHTML(SERVER_ERROR);
149                             dialogBox.center();
150                             closeButton.setFocus(true);
151                         }
152
153                         public void onSuccess(String result) {
154                             dialogBox.setText("Remote Procedure Call");
155
                                serverResponseLabel.removeStyleName("serverResponseLabelError"
                                );
156                             serverResponseLabel.setHTML(result);
157                             dialogBox.center();
158                             closeButton.setFocus(true);
159                         }
160                     });
161                 }
162             }
163
164             // Add a handler to send the name to the server
165             MyHandler handler = new MyHandler();
166             sendButton.addClickHandler(handler);
167             nameField.addKeyUpHandler(handler);
168         }
169     }
170
```