Karlsruhe Institute of Technology

KASTEL

UC-sichere private Schnittmengenberechnung mit transparenten Enklaven

Masterarbeit

KIT – Karlsruher Institut für Technologie
ITI – Institut für Theoretische Informatik
Forschungsruppe Kryptographie und Sicherheit

Anastasia Zinkina

31. Januar 2019

Verantwortlicher Betreuer: Prof. Dr. Jörn Müller-Quade
Betreuender Mitarbeiter: Jeremias Mechler

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der gültigen Fassung beachtet habe.

Karlsruhe, den 31. Januar 2019

(Anastasia Zinkina)

Inhaltsverzeichnis

1 Einleitung

Ein gängiges Problem in der sicheren Mehrparteienberechnung ist das *Problem der privaten Schnittmengenberechnung* (Private Set Intersection, PSI). Bei diesem Problem geht es darum, dass zwei Parteien jeweils eine Eingabemenge haben und den Schnitt dieser Eingaben berechnen wollen. Dabei sollten die beiden Parteien außer dem Schnitt nichts weiteres über die Eingabe der jeweils anderen Partei lernen. In dieser Arbeit stellen wir Ansätze von um das Problem der privaten Schnittmengenberechnung effizienter zu lösen.

Für dieses Problem gibt es z B folgende Anwendungsfälle:

- In [DT10] ist eine der Parteien eine Steuerbehörde, die eine Menge der Steuerhinterziehung verdächtigter Personen hat und die andere eine Bank, die eine Menge ihrer Kunden hat. Gemeinsam wollen sie den Schnitt berechnen um Steuerhinterziehung zu bekämpfen, aber die Bank will nicht die Liste all ihrer Kunden offenlegen. Auch die Steuerbehörde will die Liste der Verdächtigten geheim halten.

- In der Praxis stellt sich das Problem der privaten Schnittmengenberechnung bei der Kontaktfindung in Messenger-Apps. Dabei ist einer der Parteien der Nutzer mit seinen Kontakten auf dem Smartphone. Die andere Partei ist der App-Anbieter, der eine Liste aller Handynummern führt, deren Besitzer diesen Messenger verwenden. Der App-Anbieter will diese Liste all seiner Nutzer nicht offenlegen. Ein Nutzer will zumindest die Handynummern aus seinen Kontakten geheim halten, die nicht diese Messenger-App verwenden.

In dieser Arbeit stellen wir Ansätze von um das Problem der privaten Schnittmengenberechnung durch den Einsatz von Enklaven effizienter zu lösen. Intel Software Guard Extensions (SGX, beschrieben in [CD16]) ist eine von Intel entwickelte Prozessorerweiterung, die in Microcode implementiert ist. Das Konzept dabei ist, dass in einem Prozessor ein Programm als *Enklave* installiert und ausgeführt werden kann. Die Ausgaben dieses Programms können gemeinsam mit dem Programmcode signiert werden. Dadurch kann man prüfen, dass diese Ausgabe aus einer Enklave mit bestimmten Programm auf einem Intel-Prozessor stammt. Dieses Prinzip wird *Attested Execution* genannt. Die Enklave kann dabei einen internen Zustand

haben, der vor anderen Prozessen, Hardwarekomponenten, dem Betriebssystem und dem Computerbenutzer verborgen ist. Auf diese Art und Weise kann ein Programm sogar dann sicher ausgeführt werden, wenn beispielsweise das Betriebssystem kompromittiert ist. Somit verspricht Intel SGX für kryptographische Protokolle, dass beliebige Programme effizient in einer unsicheren Umgebung korrekt und vertraulich ausgeführt werden können.

In der Praxis gibt es keine Garantie, dass die Vertraulichkeit oder Korrektheit der Ausführung gewährleistet ist. Man muss dem Hersteller des Prozessors vertrauen, dass der Prozessor korrekt hergestellt wurde. Aber auch im Fall eines ehrlichen Prozessorherstellers können unabsichtlich Fehler auftreten. Ein Beispiel für einen solchen Fehler stellt die Übertragung des Spectre-Angriffs von gewöhnlichen Intel-Prozessoren auf SGX-Enklaven in [Van+18] und [Wei+18] dar. Bei Spectre wird ausgenutzt, dass der Prozessor Instruktionen bereits spekulativ ausführt, bevor Zugriffsberechtigungen vollständig geprüft wurden. Damit kann man über Prozessgrenzen hinweg und sogar aus SGX-Enklaven unerlaubt Speicher auslesen. Daher ist es für Anwendungen, die sich auf Enklaven verlassen, wichtig, den Verlust an Sicherheit bei Versagen der Enklaventechnologie genau zu verstehen und in Modellen abzubilden.

Ein Prozessor mit Intel SGX kann durch ein idealisiertes theoretisches Modell eines *Attested Execution Prozessors* abgebildet werden, wie es in [PST17] im Rahmen des *Universal Composition Frameworks*, vorgestellt in [Can01], beschrieben wird. Ein solches Modell erlaubt die Sicherheit eines Protokolls, welches Attested Execution Prozessors benutzt, formal beweisen zu können.

Um die Realität möglichst gut abzubilden, werden die Konzepte einer nicht-transparenten und einer transparenten Enklave verwendet. Eine nicht-transparente Enklave hält ihren internen Zustand geheim. Das modelliert die Situation, in der der Benutzer dem Hersteller vollständig vertraut. Die Annahme einer nicht-transparenten Enklave ist zu stark, wenn dem Hersteller absichtlich oder unabsichtlich bei der Implementation Fehler unterlaufen.

In [PST17] wird als schwächere Annahme das Konzept der *transparenten Enklave* vorgestellt, die sich dadurch auszeichnet, dass sie ihren Zustand nicht geheim halten kann. Diese Variation der Enklave ist interessant, wenn man annimmt, dass die Enklave zwar nicht aktiv korrumpiert ist, aber mit Absicht oder durch Implementierungsfehler ihren Zustand an den Hersteller oder Angreifer verrät.

In dieser Arbeit betrachten wir sowohl transparente als auch nicht-transparente Enklaven. Wir gehen davon aus, dass im Normalfall die Enklave ihren Zustand geheim halten kann. Auf der anderen Seite wollen wir, dass das Protokoll noch Sicherheit bietet, wenn die Enklave transparent ist. Die vorgestellten Protokolle unterscheiden sich in Effizienz, Anforderungen und Sicherheitsziel. Diese Protokolle betrachten den Fall, dass ein Dienstanbieter Enklaven zur Verfügung stellt. Der Dienstanbieter nimmt die Eingaben der anderen Protokollteilnehmer entgegen und führt die Schnittmengenberechnung durch. Für eines der Protokolle wird ein

Random Oracle angenommen. Das zweite Protokoll schwächt diese Annahme ab und ersetzt das Random Oracle durch eine kollisionsresistente Hashfunktion. Alle Teilnehmer des zweiten Protokolls besitzen eigene Enklaven.

1.1 Motivation

In der Praxis gehen verschiedene Messenger-Apps unterschiedlich mit dem Problem der privaten Schnittmengenberechnung um. Beispielsweise sendet ein Smartphone mit WhatsApp an den WhatsApp Server gekürzte MD5-Hashwerte, wobei von der Landesvorwahl keine Hashwerte gebildet werden, sondern diese direkt an den Hashwert angehängt wird. Der Server vergleicht diese Werte mit einer Liste der entsprechend bearbeiteten Telefonnummern aller WhatsApp Nutzer und sendet alle Treffer zurück. Dies ist in einem Bericht zum Datenschutz bei WhatsApp [Aft14] beschrieben:

"28. According to WhatsApp, in-network numbers are stored as original values (i.e., in clear text) on their servers. Out-of-network numbers are stored as one-way, irreversibly hashed values. WhatsApp uses a multi-step treatment of the numbers, with the key step being an "MD5" hash function. The phone number and a fixed salt value serve as input to the hash function, and the output is truncated to 53 bits and combined with the country code for the number. The result is a 64-bit value which is stored in data tables on WhatsApp's servers. According to WhatsApp, this procedure is designed to render out-of-network numbers (i.e., the mobile numbers of non-users) anonymous."

Dieser Ansatz hat einige Probleme:

- MD5 sollte nicht mehr benutzt werden, da effiziente Angriffe auf ihre Kollisionsresistenz bekannt sind. Für diese Anwendung leitet sich daraus aber nicht direkt ein praktischer Angriff ab.
- Es ist offensichtlich, dass der Dienstanbieter die Landesvorwahl aller Kontakte eines Nutzers lernt, denn diese werden an den mit MD5 erzeugten Hashwert angehängt.
- Weil Handynummern niedrige Entropie haben, kann WhatsApp alle möglichen Handynummern ausprobieren und damit die zugehörige Telefonnummer zu einem Hashwert finden.
- WhatsApp lernt die Anzahl der Kontakte im Adressbuch des Benutzers.
- Ein Netzwerkangreifer lernt die Größe des Schnitts.

- Der Vollständigkeit halber merken wir noch an, dass WhatsApp den Schnitt beliebig manipulieren kann. Beispielsweise könnte WhatsApp immer jeden zweiten Kontakt aus dem Adressbuch des Benutzers im Schnitt zurückgeben. Dieser Angriff bietet in diesem Szenario aber keinen praktischen Nutzen.

Signal führt die Kontaktfindung bereits in einer SGX Enklave durch. In [Mar17] ist dieses Vorgehen beschrieben. Es löst manche Probleme des Ansatzes von WhatsApp.

Beide Verfahren sind schlecht dokumentiert. Außerdem wird nicht auf ihre Sicherheitsgarantien eingegangen. Grundsätzlich fehlt ihnen eine formale Beschreibung und ein Sicherheitsbeweis. Die Protokolle sind aber dennoch sehr effizient. Deshalb wollen wir sie als Ausgangspunkt verwenden um ähnlich effiziente aber sichere Protokolle zu bauen.

1.2 Verwandte Arbeiten

Das Problem der privaten Schnittmengenberechnung ist eine oft betrachtete Problemstellung der sicheren Mehrparteienberechnung. Daher gibt es verschiedene Arbeiten die sich auf klassische Weise (ohne Einsatz von Enklaven) mit dem Problem beschäftigen. Wir wollen uns hier auf eine Auswahl beschränken.

In [DT10] wurden verschiedene Ideen vorgestellt, wie die Protokolle zur sicheren Schnittmengenberechnung schneller und somit in realen Szenarien nutzbarer gestaltet werden können. In [HEK12] werden Yao's generic garbled-circuit für sichere Mehrparteienberechnung angewendet. Die vorgestellten Protokolle können sich mit den speziell für die sichere Schnittmengenberechnung entwickelten, wie [DT10], messen. Damit war möglich private Schnittmengenberechnung auf Eingabemengen mit Millionen von Elementen auf herkömmlichen Desktop-Rechnern durchzuführen. Die Messungen in [HEK12] zeigen, dass das Protokoll für höhere Sicherheitsniveaus besser abschneidet als das in [DT10]. [Kam+14] spezialisiert sich darauf, auch mit Eingaben von Milliarden von Elementen umgehen zu können.

Auf der andern Seite gibt vielfältige Überlegungen, Enklaven für Mehrparteienberechnungen einzusetzen. Beispielsweise wird in [Bah+17] basierend auf der Formalisierung aus [Bar+16] ein Protokoll vorgestellt, welches auch zur sichere Schnittmengenberechnung verwendet werden kann. Die Arbeit basiert auf der Annahme, dass Die Enklave eine sichere Primitive ist und nicht korrupt sein kann. Das heißt es wird nicht auf den Verlust an Sicherheit eingegangen, der auftritt, wenn Intel SGX die Vertraulichkeitsgarantie nicht hält, die es verspricht.

In [YY08] wird ein Ansatz vorgestellt eine spezielle Variante von Identity Based Encryption (IBE) zu verwenden um GUC-sichere ([Can+07]) Schnittmengenberechnung zu realisieren. Als globale ideale Funktionalität wird $\mathcal{G}_{acrs}$ (wie in Programm 8) angenommen. Hier wird

eine einseitige Schnittmengenberechnung betrachtet, bei der nur eine der beiden Parteien den Schnitt lernt. Für in [YY08] vorgestellte Instantiierung dieses IBE Schema werden Pairings benutzt, sodass für jedes Element in der Eingabe einer der Parteien mehrere Pairing Auswertungen nötig sind. Das in unserer Arbeit vorgestellte Protokoll verzichtet auf eine solch rechenintensive Primitive.

In [Pin+18] wird ein Protokoll vorgestellt, das in einem Standalone-Modell als sicher bei einem semi-honest Angreifer bewiesen wurde. Es verwendet $\binom{N}{1}$-OTs und Cuckoo-Hashing um das Protokoll effizienter als den in [Pin+18] vorgestellten ersten Entwurf zu machen.

In [GN17] wird ein UC-sicheres Protokoll zur sicheren Schnittmengenberechnung vorgestellt, ohne Einsatz von Enklaven. Es ist interessant, weil es eines der wenigen UC-sicheren Protokolle zur privaten Schnittmengenberechnung ist. Dieses Protokoll basiert auf einem selbst definierten Primitiv, welches *obliviously polynomial addition* (OPA) genannt wird. Dieses wird aus *oblivious linear evaluation* (OLE) konstruiert, welches eine Generalisierung von OT ist. Die grundlegende Idee der Arbeit ist es für die Schnittberechnung die Eingabemengen als Nullstellen in Polynome einzubetten und diese Polynome zu addieren. Das vorgestellte Protokoll gibt den Schnitt an beide Parteien aus und ist auch gegen aktive Angreifer sicher. Interessant ist hierbei, dass [GN17] genau wie unsere Arbeit von einer vorher bekannten maximalen Eingabegröße ausgeht. Auch geht [GN17] von einer maximalen Größe der Elemente in den Eingabemengen aus um sie als Elemente in einem endlichen Körper darstellen zu können. Das tut unsere Arbeit nicht, denn sie modelliert die Elemente als Bitstrings beliebiger Länge. Des weiteren ist die Definition der UC-realisierten Funktionalität leicht anders. Im Gegensatz zu unserer Arbeit wird in [GN17] kein Wert darauf gelegt, den Schnitt (und auch nicht dessen Größe) vor einem Angreifer geheim zu halten.

In [PST17] wird kein Protokoll zur Schnittmengenberechnung vorgestellt. Hier konzentriert man sich auf die Modellierung von Attested Execution Processors und schneidet auch kurz den Fall an, in dem die Enklave nicht mehr Vertraulichkeit gewährleisten kann.

1.3 Unser Beitrag

Diese Arbeit stellt zwei Protokolle mit konstanten Rundenkomplexität zur privaten Schnittberechnung mit Enklaven vor. Ihre Kommunikationskomplexität ist dabei unabhängig von der Größe der Elemente in den Eingabemengen und vergleichbar mit bereits existierenden Protokollen. Ihre Sicherheit wird dabei im UC-Framework modelliert und bewiesen. Sie sind einfach auf mehr als zwei Parteien erweiterbar.

In dieser Arbeit betrachten wir transparente und nicht-transparente Enklaven. Das erlaubt die Sicherheit des Protokolls sowohl für den Fall, dass die Enklaventechnologie sicher ist, als

auch für den Fall, dass die Enklave die ihr anvertrauten Geheimnisse nicht schützen kann, zu modellieren. Dadurch bilden wir exakt ab, in welchem Ausmaß das Protokoll sich auf die Ehrlichkeit des Prozessorherstellers verlässt.

1.4 Gliederung der Arbeit

Zunächst werden in dieser Arbeit verschiedene Sicherheitsdefinitionen angegeben und Annahmen definiert. Des weiteren wird ein grober Überblick über Modellierung im UC-Framework gegeben und ideale Funktionalitäten vorgestellt, die für die Konstruktion der Protokolle benötigt werden. Danach wird das Protokoll *protoRO* zur sicheren Schnittmengenberechnung mit Zuhilfenahme einer dritten Partei und der Verwendung von einem Random Oracle vorgestellt. Dieses Protokoll wird auf seine Sicherheitseigenschaften untersucht. Danach wird das Protokoll *protoHASH* betrachtet, welches eine Hashfunktion anstelle von einem Random Oracle verwendet und seine Sicherheit im UC-Framework bewiesen.

2 Grundlagen

In diesem Abschnitt werden die notwendigen Grundlagen für das Verständnis des in dieser Arbeit vorgestellten Protokolls dargelegt. Diese umfassen Private- und Public-Key Verschlüsselung, Signaturverfahren, die DDH-Annahme und das UC-Framework zur Modellierung von Sicherheitseigenschaften. Zusätzlich wird eine Reihe von Annahmen innerhalb des UC-Frameworks vorgestellt. Für die Sicherheitsdefinitionen wird von einem probabilistischen Polynomialzeit-Angreifer mit polynomiellem Hinweis (nicht-uniformer PPT-Angreifer) ausgegangen.

2.1 Allgemeine Definitionen

Als erstes werden einige klassische kryptographische Primitive zusammen mit ihren Eigenschaften und Sicherheitsdefinitionen vorgestellt. Auf diese Sicherheitsbegriffe wird später die Sicherheit des Protokolls reduziert. Dabei wird die Sicherheit immer asymptotisch in Abhängigkeit von einem Sicherheitsparameter λ betrachtet. Die Wahrscheinlichkeit, dass die Sicherheit nicht gewährleistet werden kann, muss dann vernachlässigbar in λ sein. Die Laufzeit aller Algorithmen muss auch polynomiell in diesem Sicherheitsparameter sein. Die Laufzeit von Algorithmen wird typischerweise in Abhängigkeit der Eingabelänge betrachtet. Deshalb wird in dieser Arbeit davon ausgegangen, dass Algorithmen zusätzlich den Sicherheitsparameter in unärer Kodierung als Eingabe erhalten.

Vernachlässigbare Funktion Sicherheitsdefinitionen können selten absolute Sicherheit garantieren, denn mit einer gewissen Wahrscheinlichkeit kann der Angreifer beispielsweise den zufällig gezogenen Schlüssel raten. Daher wird der Begriff der *vernachlässigbaren Funktion* definiert. Wir definieren ein Verfahren als sicher, wenn die Wahrscheinlichkeit, mit der ein Angriff auf das Verfahren gelingt, vernachlässigbar ist. Dabei ist wichtig, dass diese Restunsicherheit für große Werte des Sicherheitsparameters λ schnell sehr klein wird. Die nachfolgende Definition einer vernachlässigbaren Funktion ist von [KL14, Definition 3.5] übernommen.

Definition 2.1 (Vernachlässigbare Funktion)**.** *Eine Funktion f ist vernachlässigbar, wenn für jedes Polynom $p(.)$ ein N existiert, sodass für alle $n \in \mathbb{Z}$ mit $n > N$ gilt:*

$$f(n) < \frac{1}{p(n)}$$

Private-Key Verschlüsselung und Public-Key Verschlüsselung Um die Vertraulichkeit einer über ein unsicheres Netzwerk gesendeten Nachricht vor einem Angreifer zu schützen, können zwei Parteien ein Verschlüsselungsverfahren einsetzen. Man unterscheidet dabei zwischen symmetrischen und asymmetrischen Verfahren. Die symmetrischen werden auch Private-Key Verschlüsselung, die asymmetrischen auch Public-Key Verschlüsselung genannt. Bei symmetrischen Verfahren haben die Parteien einen gemeinsamen geheimen Schlüssel, mit dem sie ver- und entschlüsseln können. Bei asymmetrischen Verfahren wird ein Schlüsselpaar generiert. Der öffentliche Schlüssel wird zur Verschlüsselung und der private Schlüssel zur Entschlüsselung benutzt.

Definition 2.2 (Symmetrische Private-Key Verschlüsselung)**.** *Ein symmetrisches Private-Key Verschlüsselung Verfahren (SKE) wird durch die polynomialzeit Algorithmen SKE.Gen, SKE.Enc, SKE.Dec beschrieben, die in [KL14, Definition 3.8] wie folgt definiert sind:*

- *Der probabilistische* Algorithmus zur Schlüsselgenerierung $k \leftarrow SKE.Gen(1^\lambda)$ *nimmt als Eingabe die unäre Kodierung des Sicherheitsparameters und gibt den Schlüssel k aus.*
- *Der probabilistische* Algorithmus zur Verschlüsselung $c \leftarrow SKE.Enc_k(m)$ *nimmt als Eingabe den Klartext $m \in \{0,1\}^*$ und den Schlüssel k und gibt das Chiffrat c aus.*
- *Der deterministische* Algorithmus zur Entschlüsselung $m \leftarrow SKE.Dec_k(c)$ *entschlüsselt das Chiffrat c unter Verwendung des privaten Schlüssels k und gibt die Nachricht m aus.*

Die Korrektheit ist gegeben, wenn folgendes gilt: $\forall \lambda \in \mathbb{N}, \forall m \in \{0,1\}^*$ *:*

$$Pr[k \leftarrow SKE.Gen(1^\lambda) : SKE.Dec_k(SKE.Enc_k(m)) = m] = 1$$

Die Wahrscheinlichkeit wird über den von den probabilistischen Algorithmen SKE.Gen und SKE.Enc verwendeten Zufall gebildet.

Ein symmetrisches SKE Schema ist IND-CCA sicher, wenn jeder nicht-uniforme PPT-Angreifer im IND-CCA Sicherheitsspiel nur vernachlässigbaren Vorteil hat.

Definition 2.3 (IND-CCA Sicherheit)**.** *Ein symmetrisches SKE Verfahren $\Pi \coloneqq (SKE.Gen, SKE.Enc, SKE.Dec)$ wird IND-CCA sicher genannt (Ununterscheidbarkeit bei adaptivem Chosen-Chiphertext-Angriff), wenn für alle $\lambda \in \mathbb{N}$ in dem in Experiment 1 vorgestelltem Spiel der Vorteil*

$\mathrm{Adv}_{\Pi}^{\text{IND-CCA}}(\lambda)$ *für alle nicht-uniformen PPT-Angreifer* $\mathcal{A}$*, die polynomiell viele Anfragen an das Verschlüsselungsorakel SKE.OEncrypt und polynomiell viele Anfragen an das Entschlüsselungsorakel SKE.ODecrypt stellen, welche in Experiment 2 und Experiment 3 beschrieben sind, durch die vernachlässigbare Funktion* $\varepsilon(\lambda)$ *beschränkt ist:*

$$\mathrm{Adv}_{\Pi}^{\text{IND-CCA}}(\lambda) := \max_{\mathcal{A}}\{\Pr[\mathbf{Exp}_{\Pi,\mathcal{A}}^{\text{IND-CCA}-1}(\lambda)= 1] - \Pr[\mathbf{Exp}_{\Pi,\mathcal{A}}^{\text{IND-CCA}-0}(\lambda)= 1]\}.$$

Die Wahrscheinlichkeit wird über den im Sicherheitsexperiment von dem Angreifer $\mathcal{A}$ *und den probabilistischen Algorithmen SKE.Gen und SKE.Enc verwendeten Zufall gebildet.*

Experiment 1 : $\mathbf{Exp}_{\Pi,\mathcal{A}}^{\text{IND-CCA}-b}(\lambda)$ Sicherheitsexperiment für Private-Key Verschlüsselung

$k \leftarrow SKE.Gen(1^{\lambda})$
$Q_d := \emptyset$
$(\text{state}, m_0, m_1) \leftarrow \mathcal{A}^{SKE.OEncrypt(\cdot),\, SKE.ODecrypt(\cdot)}$ (FIND)
$c^* \leftarrow SKE.Enc_k(m_b)$
$b' \leftarrow \mathcal{A}^{SKE.OEncrypt(\cdot),\, SKE.ODecrypt(\cdot)}$ (GUESS, state, c^*)
if $c^* \in Q_d \vee |m_0| \neq |m_1|$ **then**
  return 0
else
  return b'
end

Experiment 2 : $SKE.OEncrypt(m)$ Verschlüsselungsorakel

$c \leftarrow SKE.Enc_k(m)$
return c

Experiment 3 : $SKE.ODecrypt(c)$ Entschlüsselungsorakel

$Q_d := Q_d \cup \{\text{c}\}$
$m := SKE.Dec_k(c)$
return m

In diesem Spiel darf der Angreifer beliebig Nachrichten unter einem vom Herausforderer zufällig gewählten Schlüssel verschlüsseln und entschlüsseln lassen. Danach muss er sich für zwei Nachrichten m_0 und m_1 entscheiden und diese dem Herausforderer geben. Der Herausforderer zieht ein Bit $b \leftarrow \{0, 1\}$, verschlüsselt $c^* \leftarrow Enc_k(m_b)$ und übergibt c^* dem Angreifer. Der Angreifer darf weiterhin beliebig Anfragen stellen, aber nicht c^* entschlüsseln lassen.

Das Ziel des Angreifers ist es das Bit b korrekt auszugeben. Diese Sicherheitsdefinition für symmetrische Verschlüsselung orientiert sich an der in [KL14, Definition 3.31] angegebenen, geht aber von einem nicht-uniformen PPT-Angreifer aus.

Ein symmetrisches SKE Verfahren $\Pi := (SKE.Gen, SKE.Enc, SKE.Dec)$ wird IND-CPA sicher genannt, wenn im Experiment 1 dem Angreifer kein Entschlüsselungsorakel *SKE.ODecrypt* zur Verfügung gestellt wird. Diese Beobachtung fast die Definition aus [KL14, Definition 3.22] zusammen.

Für asymmetrische Verschlüsselung gibt es eine analoge Definitionen in [KL14, Definition 10.1] und in [KL14, Definition 10.22]. Im Unterschied zu der Definition für SKE gibt *PKE.Gen* ein Schlüsselpaar (pk, sk) zurück, wobei der private Schlüssel sk zum Entschlüsseln und der öffentliche Schlüssel pk zum Verschlüsseln verwendet wird. Der öffentliche Schlüssel pk dem Angreifer zu Beginn der FIND-Phase übermittelt. Damit kann er selbst verschlüsseln und das Verschlüsselungsorakel wird nicht mehr benötigt.

Deterministische Private-Key Verschlüsselung Ein deterministisches Verschlüsselungsverfahren ist nicht IND-CCA sicher. Ein Angreifer kann einfach zwei beliebige Nachrichten wählen. Eine vom Verschlüsselungsorakel verschlüsseln lassen und ein Chiffrat c erhalten. Diese Nachrichten dem Herausforderer geben und prüfen ob das erhaltene Chiffrat mit c identisch ist. Für diese Arbeit ist aber ein deterministisches Verschlüsselungsverfahren nützlich, obwohl es nur eine schwächere Sicherheit garantiert. Eine Sicherheitsdefinition für diese schwächere Sicherheit wäre beispielsweise *deterministische CPA Sicherheit*. Sie orientiert sich an der Definition aus [BS17, Exercise 5.8]. Hierbei wird einem Angreifer verboten zweimal dieselbe Nachricht verschlüsseln zu lassen bzw. diese als m_i zu wählen. Das Experiment dazu ist in Experiment 4 beschrieben.

Definition 2.4 (det-IND-CPA Sicherheit)**.** *Ein symmetrisches SKE Verfahren $\Pi := (SKE.Gen, SKE.Enc, SKE.Dec)$ ist* deterministisch *wenn SKE.Enc ein deterministischer Algorithmus ist. Es wird det-IND-CPA sicher genannt, wenn für alle $\lambda \in \mathbb{N}$ in dem in Experiment 4 vorgestelltem Spiel der Vorteil $\mathrm{Adv}_{\Pi}^{\text{det-IND-CPA}}(\lambda)$ für alle nicht-uniformen PPT-Angreifer $\mathcal{A}$, durch eine vernachlässigbare Funktion $\varepsilon(\lambda)$ beschränkt ist:*

$$\mathrm{Adv}_{\Pi}^{\text{det-IND-CPA}}(\lambda) := \max_{\mathcal{A}}\{\Pr[\mathbf{Exp}_{\Pi,\mathcal{A}}^{\text{det-IND-CPA}-1}(\lambda)= 1] - \Pr[\mathbf{Exp}_{\Pi,\mathcal{A}}^{\text{det-IND-CPA}-0}(\lambda)= 1]\}.$$

Die Wahrscheinlichkeit wird über den im Sicherheitsexperiment von dem Angreifer $\mathcal{A}$ und den probabilistischen Algorithmen SKE.Gen und SKE.Enc verwendeten Zufall gebildet.

Experiment 4 : $\mathbf{Exp}^{\text{det-IND-CPA}-b}_{\Pi,\mathcal{A}}(\lambda)$ Sicherheitsexperiment für deterministische Private-Key Verschlüsselung

$k \leftarrow SKE.Gen(1^\lambda)$
$(b', \text{state}, m_0, m_1) \leftarrow \mathcal{A}$ (FIND)
$M_0 := \emptyset, M_1 := \emptyset$
while $b' = \perp$ **do**
  $c^* := SKE.Enc_k(m_b)$
  if $m_0 \in M_0 \vee m_1 \in M_1$ **then**
    return 0
  end
  $M_0 := \{m_0\} \cup M_0$
  $M_1 := \{m_1\} \cup M_1$
  $(b', \text{state}, m_0, m_1) \leftarrow \mathcal{A}$ (GUESS, state, c^*)
end
return b'

Signaturverfahren Um eine Nachricht vor Veränderung durch einen Angreifer zu schützen kann man Signaturverfahren einsetzen. Das Signaturverfahren berechnet dabei einen zusätzlichen Wert (die Signatur), der typischerweise gemeinsam mit der Nachricht gesendet wird. Wenn der Empfänger diese Signatur prüft, wird eine Veränderung durch den Angreifer auffallen. Dadurch kann eine Nachricht authentifiziert werden. Zur Verwendung des Verfahrens wird ein Schlüsselpaar erzeugt, von dem man den einen Schlüssel (Signaturschlüssel) zum erzeugen und den anderen (Verifikationsschlüssel) zum Verifizieren von Signaturen benutzen kann.

Definition 2.5 (Signaturverfahren)**.** *Ein Signaturverfahren wird durch die polynomialzeit Algorithmen Gen, Sign, Verify beschrieben, die in [KL14, Definition 3.8] wie folgt definiert sind:*

- *Der probabilistische* Algorithmus zur Schlüsselgenerierung $(vk, sk) \leftarrow Gen(1^\lambda)$ *nimmt als Eingabe die unäre Kodierung des Sicherheitsparameters und generiert ein Schlüsselpaar* (vk, sk)*, bestehend aus einem Signaturschlüssel* sk *und einem Verifikationsschlüssel* vk*.*
- *Der probabilistische* Algorithmus zur Signaturerzeugung $\sigma \leftarrow Sign_{sk}(m)$ *nimmt als Eingabe den Klartext* $m \in \{0,1\}^*$ *und den Signaturschlüssel* sk *und gibt die dazu passende Signatur* σ *aus.*
- *Der deterministische* Algorithmus zur Verifikation der Signatur $b \leftarrow Verify_{vk}(m, \sigma)$ *mit*

$b \in \{0,1\}$ *überprüft, ob die Signatur* σ *zur Nachricht* m *und dem* vk *passt. Wenn dies der Fall ist, gibt der Algorithmus* 1*, sonst* 0*, aus.*

Die Korrektheit ist gegeben, wenn folgendes gilt: $\forall \lambda \in \mathbb{N}, \forall m \in \{0,1\}^* :$

$$Pr[(vk, sk) \leftarrow Gen(1^\lambda) : Verify_{vk}(m, Sign_{sk}(m)) = 1] = 1$$

Die Wahrscheinlichkeit wird über den von den probabilistischen Algorithmen *Gen* und *Sign* verwendeten Zufall gebildet.

Ein Signaturverfahren ist EUF-CMA sicher, wenn jeder nicht-uniforme PPT-Angreifer im EUF-CMA Sicherheitsspiel nur vernachlässigbaren Vorteil (welcher der Erfolgswahrscheinlichkeit entspricht) hat.

Definition 2.6 (EUF-CMA Sicherheit)**.** *Ein Signaturverfahren* $\Sigma := (Gen, Sign, Verify)$ *wird EUF-CMA sicher genannt (existenzielle Unfälschbarkeit der Signatur beim adaptivem Chosen-Message-Angriff), wenn für alle* $\lambda \in \mathbb{N}$ *in dem in Experiment 5 vorgestelltem Spiel der Vorteil* $\mathrm{Adv}_{\Sigma}^{\text{EUF-CMA}}(\lambda)$ *für alle nicht-uniformen PPT-Angreifer* $\mathcal{A}$*, die polynomiell viele Anfragen an das Orakel OSign stellen, welches in Experiment 6 beschrieben ist, durch die vernachlässigbare Funktion* $\varepsilon(\lambda)$ *beschränkt ist:*

$$\mathrm{Adv}_{\Sigma}^{\text{EUF-CMA}}(\lambda) := \max_{\mathcal{A}}\{\Pr[\mathbf{Exp}_{\Sigma,\mathcal{A}}^{\text{EUF-CMA}}(\lambda) = 1]\}$$

Die Wahrscheinlichkeit wird über den im Sicherheitsexperiment von dem Angreifer $\mathcal{A}$ *und den probabilistischen Algorithmen Gen und Sign verwendeten Zufall gebildet.*

Experiment 5 : $\mathbf{Exp}_{\Sigma,\mathcal{A}}^{\text{EUF-CMA}}(\lambda)$ Sicherheitsexperiment für Signaturverfahren

```
(vk, sk) ← Gen(1^λ)
Q_s := ∅
(m*, σ) ← A^{OSign(·)}(pk)
if m* ∈ Q_s ∨ Verify_vk(m*, σ) = 0 then
    return 0
end
return 1
```

Experiment 6 : $O.Sign(m)$ Signaturorakel

```
Q_s := Q_s ∪ {m}
σ ← Sign_sk(m)
return σ
```

In diesem Spiel wählt der Herausforderer ein Schlüsselpaar und übergibt den Verifikationsschlüssel vk dem Angreifer. Dieser darf das Orakel *OSign* beliebige Nachrichten signieren lassen und muss am Ende eine Nachricht für die er keine Signatur angefragt hat mit korrekter Signatur ausgeben. Diese Sicherheitsdefinition für ein Signaturverfahren orientiert sich an der in [KL14, Definition 12.2] angegebenen, geht aber von einem nicht-uniformen PPT-Angreifer aus.

Hashfunktion Eine Hashfunktion komprimiert Eingaben beliebiger Länge in Zeichenketten fester Länge. Dabei ist es praktisch, wenn es schwierig ist zwei Eingaben zu finden, die auf den gleichen Wert abgebildet werden. Eine Hashfunktion kann eingesetzt werden, um statt großer Elemente nur ihre Hashwerte zu vergleichen.

Definition 2.7 (Hashfunktion)**.** *Eine Hashfunktion wird durch polynomialzeit Algorithmen Gen und H beschrieben, die in [KL14, Definition 4.9] wie folgt definiert sind:*

- *Der probabilistische* Algorithmus zur Schlüsselgenerierung $s \leftarrow Gen(1^\lambda)$ *nimmt als Eingabe die unäre Kodierung des Sicherheitsparameters und generiert ein Schlüssel s.*

- *Es existiert ein Polynom l, dass der deterministische* Algorithmus zur Berechnung eines Hashwertes $h \leftarrow H^s(x)$ *als Parameter den Schlüssel s und als Eingabe eine Zeichenkette* $x \in \{0,1\}^*$ *und berechnet eine Zeichenkette* $H^s(x) \in \{0,1\}^{l(\lambda)}$.

Eine Hashfunktion ist kollisionsresistent, wenn jeder nicht-uniforme PPT-Angreifer höchstens mit vernachlässigbarer Wahrscheinlichkeit eine Kollision findet.

Definition 2.8 (Kollisionsresistenz)**.** *Eine Hashfunktion $\mathcal{H}$ wird kollisionsresistent genannt, wenn für alle $\lambda \in \mathbb{N}$ in dem in Experiment 7 vorgestelltem Spiel der Vorteil* $\mathrm{Adv}^{\mathrm{coll}}_{\mathcal{H}}(\lambda)$ *für alle nicht-uniformen PPT-Angreifer $\mathcal{A}$ durch die vernachlässigbare Funktion $\varepsilon(\lambda)$ beschränkt ist:*

$$\mathrm{Adv}^{\mathrm{coll}}_{\mathcal{H}}(\lambda) := \max_{\mathcal{A}}\{\Pr[\mathbf{Exp}^{\mathrm{coll}}_{\mathcal{H},\,\mathcal{A}}(\lambda) = 1]\}$$

Die Wahrscheinlichkeit wird über den im Sicherheitsexperiment von dem Angreifer $\mathcal{A}$ und dem probabilistischen Algorithmus Gen verwendeten Zufall gebildet.

Experiment 7 : $\mathbf{Exp}^{\text{coll}}_{\mathcal{H},\mathcal{A}}(\lambda)$ Sicherheitsexperiment für Kollisionsresistenz

$s \leftarrow Gen(1^\lambda)$
$(x, x') \leftarrow \mathcal{A}\ (s)$
if $x \neq x' \wedge H^s(x) = H^s(x')$ **then**
  return 1
else
  return 0
end

In dem Spiel wählt der Herausforderer ein Schlüssel s und übergibt ihn dem Angreifer. Dieser muss eine Kollision ausgeben. Eine Kollision besteht aus zwei verschiedenen Zeichenketten, die den gleichen Hashwert haben. Diese Definition für eine kollisionsresistente Hashfunktion orientiert sich an der in [KL14, Definition 4.10] angegebenen, geht aber von einem nicht-uniformen PPT-Angreifer aus.

In dieser Arbeit wird zur Übersichtlichkeit der Schlüssel s nicht explizit gezogen und angegeben. Die Protokolle gehen von einem global generiertem Schlüssel aus.

Jede kollisionsresistente Funktion ist auch eine Einwegfunktion, d.h. in einem Experiment in dem ein Challenger einen Zufallswert x zieht und dem Angreifer $h(x)$ gibt, ist die Wahrscheinlichkeit, dass der Angreifer x ausgibt vernachlässigbar. Diese Eigenschaft lässt sich auch direkt auf Kollisionsresistenz reduzieren: Wenn dem Angreifer, der die Einwegeigenschaft brechen kann, die Challenge $h(x)$ übermittelt wird, antwortet er mit x'. Mit überwältigender Wahrscheinlichkeit ist $x \neq x'$. Damit hat der Angreifer eine Kollision x, x' gefunden.

Die Decisional Diffie-Hellman-Annahme (DDH-Annahme) Die folgende Definition ist aus [KL14, Definition 7.60] übernommen, geht aber von einem nicht-uniformen PPT-Angreifer aus.

Definition 2.9 (DDH-Annahme)**.** *Das DDH-Problem ist schwer in Gruppen, die durch den polynomialzeit Algorithmus $\mathcal{G}$ ausgegeben werden, wenn für alle nicht-uniformen PPT-Angreifer $\mathcal{A}$ eine vernachlässigbare Funktion $\varepsilon(\lambda)$ existiert, sodass*

$$|\Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^z) = 1] - \Pr[\mathcal{A}(\mathbb{G}, q, g, g^x, g^y, g^{xy}) = 1]| \leq \varepsilon(\lambda),$$

wobei $(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^\lambda)$, wo q die Ordnung und g ein Generator von $\mathbb{G}$ ist, und x, y, z aus $\mathbb{Z}_q$ zufällig gleichverteilt gezogen wurden. Die Wahrscheinlichkeit wird über den im Sicherheitsexperiment von dem Angreifer $\mathcal{A}$ verwendeten Zufall und die Wahl von x, y, z gebildet.

Die DDH-Annahme *für $\mathcal{G}$ ist, dass das DDH-Problem in den von $\mathcal{G}$ ausgegebenen Gruppen schwer ist.*

Non-interactive witness indistinguishable Beweissysteme Beweissysteme dienen dazu, einer Partei glaubhaft zu machen, dass ein bestimmtes Wort x in einer formalen Sprache ist. Beweissysteme für NP-Sprachen können in dem Beweis Informationen über den Zeugen verraten oder ihn sogar vollständig ausgeben. Manchmal ist es wichtig, dass aus dem Beweis nicht ersichtlich ist, welcher der mehreren möglichen Zeugen verwendet wurde. Ein solches Beweissystem heißt *witness indistinguishable*. Das ist beispielsweise für diese Sprache, die als Vereinigung zweier Teilsprachen mit je einer Zeugenrelationen R_1 bzw. R_2 definiert ist, interessant: $L := \{x \mid \exists w_1 : R_1(x, w_1)\} \cup \{x \mid \exists w_2 : R_2(x, w_2)\} = \{x \mid \exists (w_1, w_2) : R_1(x, w_1) \vee R_2(x, w_2)\}$ Man will einer Partei beweisen, dass $x \in L$ gilt, aber nicht verraten in welcher der beiden Teilmengen sich x befindet.

Definition 2.10 (NIWI Beweissystem)**.** *Ein non-interactive witness indistinguishable Beweissystem (NIWI Beweissystem) einer NP-Sprache L mit NP-Zeugenrelation R, für die gilt $x \in L \Leftrightarrow \exists w : R(x, w)$, wird durch die polynomialzeit Algorithmen Gen, Prove, Verify beschrieben, die in [FS90] wie folgt definiert sind:*

- *Der probabilistische* Algorithmus zur Parametergenerierung $crs \leftarrow Gen(1^\lambda)$ *nimmt als Eingabe die unäre Kodierung des Sicherheitsparameters und generiert einen Common Reference String crs.*
- *Der probabilistische* Algorithmus zur Beweiserzeugung $\pi \leftarrow Prove(crs, x, w)$ *nimmt als Eingabe crs, ein Wort $x \in L$ und einen Zeugen w mit $R(x, w)$ und generiert einen Beweis π für die Aussage $x \in L$.*
- *Der deterministische* Algorithmus zur Verifikation des Beweises $b \leftarrow Verify(crs, x, \pi)$ *mit $b \in \{0, 1\}$ überprüft, ob der Beweis π zur Wort x und crs passt. Wenn dies der Fall ist, gibt der Algorithmus* 1*, sonst* 0*, aus.*

Damit diese Algorithmen ein NIWI Beweissystem bilden, muss gelten:

Korrektheit $\forall \lambda \in \mathbb{N}, \forall (x, w)$:

$$R(x, w) \Rightarrow Pr[crs \leftarrow Gen(1^\lambda), \pi \leftarrow Prove(crs, x, w), Verify(crs, x, \pi) = 1] = 1.$$

Die Wahrscheinlichkeit wird über den von den probabilistischen Algorithmen Gen und Prove verwendeten Zufall gebildet.

computational Soundness Für alle nicht-uniformen PPT-Angreifer $\mathcal{A}$ gilt

$$Pr[crs \leftarrow Gen(1^\lambda), (x, \pi) \leftarrow \mathcal{A}(1^\lambda, crs) : x \notin L \wedge Verify(crs, x, \pi) = 1]$$

ist vernachlässigbar. Die Wahrscheinlichkeit wird über den von dem Angreifer $\mathcal{A}$ und dem probabilistischen Algorithmus Gen verwendeten Zufall gebildet.

Witness indistinguishable *Gibt es für ein $x \in L$ mehrere Zeugen $w \neq w'$ mit $R(x, w) \wedge R(x, w')$, sind die Ausgaben $Prove(crs, x, w)$ und $Prove(crs, x, w')$ computational ununterscheidbar. Es ist also aus dem Beweis π nicht zu erkennen, welcher Zeuge zum Erzeugen des Beweises genutzt wurde.*

In dieser Arbeit nehmen wir an, dass π für festes λ eine konstante Länge hat. Da *Prove* ein PPT-Algorithmus ist, ist seine Ausgabelänge durch ein Polynom in λ beschränkt. Auf diese Länge kann man π immer mit Padding ergänzen um eine konstante Länge zu erzwingen.

2.2 Universal Composition Framework

In diesem Abschnitt wird das Universal Composition Framework (UC) vorgestellt. Sicherheitsbeweise im UC-Framework haben den Vorteil, dass man die gewünschte Eigenschaft recht einfach mit einer idealen Funktionalität beschreibt und danach die Sicherheit des realen Protokolls im Bezug darauf beweist. Falls im Laufe des Beweises Probleme auffallen, kann man diese entweder akzeptieren und die ideale Funktionalität abschwächen oder das reale Protokoll verbessern. Protokolle, die bereits als UC-sicher bewiesen sind, können als "black box" in neuen Protokollen benutzt werden. Die "black box" ist dabei die ideale Funktionalität, die von dem realen Protokoll realisiert wird.

Das UC-Framework ist sehr umfangreich in [Can01] beschrieben und hier auf das Wesentliche zusammengefasst. Dabei werden bewusst Vereinfachungen gemacht. Beispielsweise wird für beide in dieser Arbeit vorgestellten Protokolle implizit ein Session-Identifier *sid* angenommen um die untersuchte Protokollinstanz von anderen laufenden Protokollinstanzen zu unterscheiden.

Maschinen- und Kommunikationsmodell Im Universal Composition Framework werden Abläufe als Interaktion von mehreren interaktiven Turingmaschinen (ITM) modelliert. Instanzen von ITMs werden ITI genannt. Diese Turingmaschinen haben neben ihrem Arbeitsband drei eingehende Kommunikationsbänder und ein ausgehendes Kommunikationsband. Um eine Nachricht zu senden wird sie zusammen mit dem Empfänger auf das ausgehende Kommunikationsband geschrieben und ein spezieller *external-write* Zustand betreten. Nachrichten können dabei auf drei verschiedene Arten beim Empfänger ankommen, als normale Netzwerkkommunikation (geschrieben auf *incoming communication tape*), als Eingabe einer Subroutine (geschrieben auf *input tape*) und als Ausgabe einer Subroutine (geschrieben auf

subroutine output tape). Wenn eine Turingmaschine *external-write* betritt, wird sie pausiert und der Empfänger darf mit der Berechnung fortfahren.

Eine Partei wird durch ihre Identität identifiziert. Identitäten der Parteien enthalten immer ihren Programmcode. Um eine Nachricht an eine Partei zu senden benötigt man den Inhalt der Nachricht und die Identität der Partei. Empfänger werden immer dann erzeugt, wenn sie die erste Nachricht bekommen.

Die Global Control Function prüft, ob der Sender berechtigt ist die gewünschte Nachricht an den Empfänger zu senden. Bei einem Aufruf von *external-write* erhält die Global Control Function als Eingabe die Identität des Senders, die Identität des Empfängers, das Band (tape), auf welches die Nachricht bei dem Empfänger geschrieben werden soll, und die Nachricht. Die Global Control Function gibt entweder 0 (verweigert damit das Senden der Nachricht) oder 1 (erlaubt das Senden der Nachricht) aus.

Korruption Die Korruption von Parteien wird dadurch modelliert, dass an die Parteien eine "corrupt"-Nachricht gesendet wird, die sie dazu veranlasst ihren internen Zustand dem Simulator oder Angreifer offenzulegen, alle empfangenen Nachrichten an ihn zu senden und von ihm gesendete Nachrichten im eigenen Namen weiterzuleiten. Der Fakt der Korruption wird immer der Umgebung mitgeteilt (sodass Angreifer und Simulator immer dieselben Parteien korrumpieren müssen). Bei der Art der Korruption gibt es unter anderem statische und adaptive Korruption. Bei statischer Korruption muss der Angreifer sich vor Beginn des Protokollablaufs darauf festlegen, welche Parteien er korrumpiert. Bei adaptiver Korruption kann es dies im Laufe des Protokollablaufs ändern.

Ausführungsmodell Um die Sicherheit eines Protokolls zu modellieren, muss eine dazugehörige ideale Funktionalität beschrieben werden. Danach wird eins von zwei Experimenten durchgeführt. In ihnen interagiert eine Umgebung entweder mit einem Angreifer und dem realen Protokoll, oder mit einem Simulator (der von dem Angreifer abhängen darf), der idealen Funktionalität und Dummy-Parteien (die nur ihre Eingaben an die ideale Funktionalität weiterleiten). Dieses Experiment ist definiert als:

Definition 2.11 (UC-Experiment)**.** *Sei π ein Protokoll. Sei $\mathcal{A}$ ein PPT-Algorithmus. Sei $\mathcal{Z}$ ein nicht-uniformer PPT-Algorithmus. Wir nennen $\mathcal{Z}$ eine Umgebung und $\mathcal{A}$ einen Angreifer. Die Ausgabe der ITI $\mathcal{Z}$ mit geeignetem polynomiellen Hinweis ist* $\mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$.

Die Global Control Function sorgt dafür, dass die erste durch $\mathcal{Z}$ gestartete ITI $\mathcal{A}$ ist. Die weiteren ITIs, denen $\mathcal{Z}$ Nachrichten sendet, sind die Parteien des Protokolls π. $\mathcal{Z}$ darf nur mit diesen ITIs kommunizieren. Wenn $\mathcal{A}$ oder eine der Parteien von π eine Antwort auf das subroutine

output tape *von $\mathcal{Z}$ schreibt, entfernt die Global Control Function den Programmcode aus der Identität der sendenden ITI, sodass $\mathcal{Z}$ deren Programmcode nicht lernt.*

Alle Parteien von π müssen ihre Nachrichten auf das incoming communication tape *von $\mathcal{A}$ schreiben und $\mathcal{A}$ darf auf das* incoming communication tape *aller Parteien Nachrichten mit beliebiger Absenderidentität schreiben.*

UC Realisierung Im Realen wird das Experiment mit Protokoll π, Angreifer $\mathcal{A}$ und Umgebung $\mathcal{Z}$ durchgeführt. Im Idealen interagiert dieselbe Umgebung $\mathcal{Z}$ mit Sim (Simulator) und Protokoll φ. Die Aufgabe des Simulators ist dabei, es für die Umgebung nicht erkennbar zu machen, welches der beiden Experimente durchgeführt wird. Ein Protokoll realisiert eine ideale Funktionalität sicher, wenn für alle Angreifer ein Simulator existiert, sodass keine Umgebung das reale und das ideale Experiment voneinander unterscheiden kann. Formal wird dies definiert als:

Definition 2.12 (UC Realisierung)**.** *Seien π, φ zwei Protokolle.*

Wir definieren:

$$\pi \geq \varphi :\Leftrightarrow \forall \mathcal{A}\ \exists \mathrm{Sim}\ \forall \mathcal{Z} : \mathrm{EXEC}_{\pi,\mathcal{A},\mathcal{Z}} \textit{ ist ununterscheidbar von } \mathrm{EXEC}_{\varphi,\mathrm{Sim},\mathcal{Z}}.$$

Komposition Der Vorteil von UC ist, dass bereits als sicher bewiesene Protokolle, als "black box" in ein neues Protokoll integriert werden können. Diese "black box" wird durch die zugehörige ideale Funktionalität modelliert.

Das zentrale Theorem von UC ist das *Universal Composition Theorem.*

Satz 2.13 (Universal Composition Theorem)**.** *Seien α, β, γ Protokolle und es gelte $\alpha \geq \beta$, dann gilt auch $\gamma^{\alpha} \geq \gamma^{\beta}$, wenn γ* subroutine respecting *ist. Subroutine respecting bedeutet im wesentlichen, dass in γ ITIs nur auf das* input tape *von ITIs schreiben, die Subroutinen von ihnen sind und auch nur von solchen auf ihrem* subroutine output tape *Nachrichten bekommen.*

Korollar 2.14. *Seine p_1, p_2 Protokolle, $\mathcal{F}_1, \mathcal{F}_2$ ideale Funktionalitäten und es gelte $p_1 \geq \mathcal{F}_1$ und $p_2 \geq \mathcal{F}_2$, dann gilt auch $p_2[\mathcal{F}_1 \to p_1] \geq \mathcal{F}_2$.*

Das bedeutet p_2 realisiert immer noch $\mathcal{F}_2$ selbst wenn darin alle Vorkommen von $\mathcal{F}_1$ durch p_1 ersetzt werden.

Dieses Korollar beschreibt, wie man komplexe Protokolle aus idealen Teilfunktionalitäten komponentenweise aufbauen kann.

Generalized UC Obwohl das UC-Framework ein mächtiges Werkzeug ist, um Protokolle zu modellieren, hat es doch seine Einschränkungen. In dem UC-Framework kann man ein globales Setup nicht darstellen. Generalized UC, vorgestellt in [Can+07], verallgemeinert UC. Neben lokalen idealen Funktionalitäten gibt es in Generalized UC auch globale. Eine globale ideale Funktionalität hat einen globalen Zustand, der für alle Interaktionen benutzt wird. Ein Beispiel für eine solche globale Funktionalität ist $\mathcal{G}_{acrs}$ oder auch $\mathcal{G}_{att}$ welche später vorgestellt werden. Jedes der von der Umgebung gestarteten Protokolle darf auf die globalen idealen Funktionalitäten zugreifen. Die Umgebung darf beliebige Protokolle parallel zu dem betrachteten Protokoll starten. Unter anderem können mehrere Instanzen des betrachteten Protokolls parallel ausgeführt werden.

Konventionen für Pseudocode In diesem Abschnitt werden die Schlüsselwörter erklärt, die im Pseudocode für die Beschreibung von idealen Funktionalitäten, Protokollen und Simulatoren benutzt werden.

receive message, on input, on subroutine output
: Bei diesen Befehlen wartet die ITI auf eine Eingabe. Wenn eine Eingabe nicht erwartet wurde, wird die Ausführung abgebrochen. Bei **receive message** wird die Nachricht auf dem *incoming communication tape*, bei **on input** auf dem *input tape* und bei **on subroutine output** auf dem *subroutine output tape* erwartet.

send, send subroutine input
: Sende eine Nachricht an das *incoming communication tape* bzw. *input tape* einer anderen ITI und gib den Kontrollfluss ab.

return
: Wie **send**, aber die Nachricht wird an das *subroutine output tape* gesendet.

send delayed output x to A while leaking y to B
: Sendet zunächst y an B und wartet auf Bestätigung von B. Danach wird x an A gesendet. Das bildet das Konzept des (partially) public/private delayed output von [Cano1, Kapitel 6.2, Abs. Delayed Output] ab.

simulate delayed output while leaking
: Gibt das Argument an die Simulation des Angreifers. Wenn dieser mit “ok” antwortet fährt die Simulator fort, ansonsten bricht er ab.

yield execution to Environment, when the Environment resumes execution

Wenn eine ITI die Ausführung nicht mit dem Senden einer Nachricht beendet, wird die Ausführung an die Umgebung zurückgegeben (yield). Die Umgebung kann sich dann für eine Partei entscheiden, die mit der Ausführung fortfahren darf (when the Environment resumes).

abort

die Partei, der Simulator (oder die Enklave) bricht das Protokoll ab. Das bedeutet, dass sie keine weiteren Nachrichten mehr annehmen, Nachrichten senden oder Berechnungen durchführen.

Die Instruktionen in den folgenden Programmen werden sequenziell ausgeführt. Beim Eintreffen einer Nachricht wird die Ausführung an der Stelle fortgeführt an der sie momentan wartet.

2.3 Bestehende ideale Funktionalitäten

In diesem Abschnitt werden die bereits bekannten und in anderen Arbeiten vorgestellten idealen Funktionalitäten aufgeführt.

Die ideale Funktionalität für einen Augmented Global Common Reference String
In Programm 8 ist die ideale Funktionalität $\mathcal{G}_{acrs}$ beschrieben, die einen Augmented Global Common Reference String (Augmented Global CRS) modelliert. Die Idee einer solchen Funktionalität stammt aus [Can+07, Kapitel 4.2]. Die hier angegebene Definition von $\mathcal{G}_{acrs}$ wurde von [PST17] übernommen. In dieser Definition ist $PKE \coloneqq (PKE.Gen, PKE.Enc, PKE.Dec)$ ein IND-CCA sicheres asymmetrisches Verschlüsselungsverfahren, $\Sigma \coloneqq (\Sigma.Gen, \Sigma.Sign, \Sigma.Verify)$ ein EUF-CMA sicheres Signaturverfahren und $NIWI \coloneqq (NIWI.Gen, NIWI.Prove, NIWI.Verify)$ ein non-interactive witness indistinguishable Beweissystem für eine NP-Sprache.

Für das in dieser Arbeit betrachtete Protokoll wird diese Funktionalität benötigt um Enklavensignaturen vor Parteien, die keinen Trusted Execution Processor haben, zu verbergen. Dafür ist die NP-Sprache des NIWI-Beweissystems definiert (wie in [PST17]) als:

$$\{(msg, P, C) \mid (r, \sigma, idk[P]) : C \coloneqq PKE.Enc_{epk}((\sigma, idk[P]), r)$$

$$\wedge (Verify_{mpk}(msg, \sigma) = 1 \vee \Sigma.Verify_{vk}(P, idk[P]) = 1)\}$$

Die Definition zeigt, dass es sich um eine NP-Sprache handelt mit dem Zeugen der Form $(r, \sigma, idk[P])$, wobei σ ist die Signatur der Nachricht msg und $idk[P]$ ist die Signatur der

Identität P. Die Sprache ist so definiert, dass bei einem Zeugen entweder σ oder $idk[P]$ korrekt sein muss, damit $NIWI.Prove$ einen gültigen Beweis ausgibt. In diesem konkreten Fall bedeutet *Witness indistinguishable*, dass aus dem Beweis nicht ersichtlich ist, ob ein korrektes σ oder korrektes $idk[P]$ für den Beweis verwendet wurde und in dem Chiffrat C enthalten ist. Die zusätzliche Eingabe r in $PKE.Enc$ stellt den zum Verschlüsseln verwendeten Zufall dar.

Programm 8 : $\mathcal{G}_{acrs}$ Ideale Funktionalität für einen Augmented Global CRS

$(epk, esk) \leftarrow PKE.Gen(1^\lambda)$
$(ssk, vk) \leftarrow \Sigma.Gen(1^\lambda)$
$crs \leftarrow NIWI.Gen(1^\lambda)$
on input "crs" **from** P **do**
  return $\mathcal{G}_{acrs}.mpk := (epk, vk, crs)$ **to** P

on input "idk" **from** P **do**
  if P *is corrupt* **then**
    return $idk[P] := \Sigma.Sign_{ssk}(P)$ **to** P
  else
    abort
  end

Die ideale Funktionalität für authentifizierten Nachrichtenaustausch $\mathcal{F}_{auth}$ stellt die ideale Funktionalität dar, die Parteien ermöglicht, Nachrichten authentifiziert zu versenden. In Programm 9 wird $\mathcal{F}_{auth}$ angegeben. Dabei wurde die Definition aus [Cano1, Kapitel 6.3] angepasst. Die zusätzlich Instruktion zur Behandlung von korrupten Sendern wird in dieser Arbeit nicht benötigt, da wir adaptive Korruption nicht betrachten.

Programm 9 : $\mathcal{F}_{auth}$ Ideale Funktionalität für authentifizierten Nachrichtenaustausch

on input (P_r, m) **from** P_s **do**
  send delayed output (P_s, m) **to** P_r **while leaking** (P_s, P_r, m) **to the Adversary**

Die ideale Funktionalität für sicheren Nachrichtenaustausch $\mathcal{F}_{smt}$ stellt die ideale Funktionalität dar, die Parteien ermöglicht vertraulich Nachrichten zu versenden. In Programm 10 wird $\mathcal{F}_{smt}$ angegeben, wie in [Cano1, Kapitel 6.4] dargestellt.

Programm 10 : $\mathcal{F}_{smt}$ Ideale Funktionalität für sicheren Nachrichtenaustausch

on input (P_r, m) **from** P_s **do**
  send delayed output (P_s, m) **to** P_r **while leaking** $(P_s, P_r, |m|)$ **to the Adversary**

Beispielsweise kann man $\mathcal{F}_{smt}$ aus $\mathcal{F}_{auth}$ und einer IND-CPA sicheren asymmetrischen Verschlüsselung konstruieren. Zunächst sendet P_s an P_r eine "initialize"-Nachricht. P_r antwortet mit einem frisch gezogenen öffentlichem Schlüssel pk, zu dem er sich den privaten Schlüssel sk merkt. Mit pk verschlüsselt P_s die Nachricht und sendet sie. P_r entschlüsselt diese Nachricht mit sk. Dieses Verfahren ist auch in [Cano1, Kapitel 6.4, Abs. On realizing $\mathcal{F}_{SMT}$] beschrieben.

Bei der Anwendung auf Nachrichten von und zu der Enklave muss beachtet werden, dass diese Nachrichten durch T weitergeleitet werden müssen. D.h. T kann auch Nachrichten unterdrücken. Dies bringt dem Angreifer aber keinen zusätzlichen Vorteil, denn der Angreifer kann diese Nachrichten auch unterdrücken indem er die von T durch $\mathcal{F}_{smt}$ versendeten Nachrichten abfängt.

Immer wenn im folgenden Protokollablauf eine Nachricht durch das Netzwerk gesendet werden soll, bedeutet dies, dass diese Nachricht über $\mathcal{F}_{smt}$ gesendet wird.

Die ideale Funktionalität für Attested Execution Die in Programm 11 angegebene Definition von $\mathcal{G}_{att}$ orientiert sich im Wesentlichen an der Definition einer Enklave in [PST17]. Diese ideale Funktionalität modelliert, wie sich ein Attested Execution Processor verhalten soll. $\mathcal{G}_{att}$ ist mit einer Liste reg, die alle Parteien mit Attested Execution Processor angibt, parametrisiert. Nur diese Parteien können Enklaven installieren und ausführen.

$\mathcal{G}_{att}$ modelliert zwei Varianten von Enklaven: transparente und nicht-transparente. Eine nicht-transparente Enklave hat Zugriff auf Zufall, den die Partei P, die den Attested Execution Processor benutzt, nicht kennt. Eine transparente Enklave meldet ihren Zufall in der Variable $bits$ an die Partei P. Dies modelliert, dass der Attested Execution Processor seinen internen Zustand nicht geheim halten kann, sondern ihn (z. B. über Seitenkanäle) verrät. Bei einer nicht-transparenten Enklave werden diese $bits$ nicht gefüllt. Dann kann die Enklave Informationen vor P verbergen. In beiden Fällen verhält sich die Enklave immer korrekt. Es wird nicht abgebildet, dass der Attested Execution Processor bei der Ausführung beeinflusst werden kann und dazu gebracht wird ein falsches Ergebnis zu unterschreiben.

Programm 11 : $\mathcal{G}_{att}[reg]$ Ideale Funktionalität für Attested Execution

(mpk, msk) $:= KeyGen(1^\lambda)$
receive message *"getpk"* **from** P **do**
  send mpk **to** P

receive message $install(\Psi)$ **from** P **do**
  if $P \notin reg$ **then**
    abort
  end
  draw eid **uniformly at random**
  store $\Psi, \vec{0}$ for index P, eid
  send eid **to** P

receive message $resume(eid, in)$ **from** P **do**
  if $P \notin reg$ **then**
    abort
  end
  find Ψ, mem for index P, eid
  $(out, mem') := \Psi(in, mem)$
  store Ψ, mem' for index P, eid
  $\sigma := Sign_{msk}(eid, \Psi, out)$
  $bits := \bot$
  if P *is corrupded and enclave leaks information* **then**
    $bits :=$ all random bits used in the execution of Ψ
  end
  send $(out, \sigma, bits)$ **to** P

Die ideale Funktionalität für ein Random Oracle Ein Random Oracle ist eine Überidealisierung einer kryptografischen Hashfunktion. Es wird in Sicherheitsbeweisen benutzt, weil es leichter zu handhaben ist als eine kollisionsresistente Hashfunktion. Wir verwenden es in einem der Protokolle um die in UC benötigte Extraktion der Eingaben zu ermöglichen. Seine ideale Funktionalität ist in Programm 12 beschrieben. Diese Definition entspricht z. B. der in [Nie02] angegebenen Definition.

Programm 12 : $\mathcal{F}_{RO}$ Ideale Funktionalität für ein Random Oracle

```
1 Globally store a map M
2 on input x from P do
3     if x is a key in M then
4         let y be the value from M at key x
5     else
6         draw y uniformly at random
7         store y in M under key x.
8     end
9     return y to P
```

Für die Sicherheit des in dieser Arbeit vorgestellten Protokolls *protoRO* ist dabei wichtig, dass

- Der Angreifer/die Umgebung keine Kollision (x, y) mit $x \neq y \wedge h(x) = h(y)$ finden kann. Das Random Oracle muss also kollisionsresistent sein. Der Angreifer/die Umgebung darf keine Kollision (x, y) finden können, denn sonst könnte sie mit den zwei Eingabemengen $S_0 := \{x\}, S_1 := \{y\}$ ein falsches Ergebnis herbeiführen, wenn der Schnitt auf Hashwerten berechnet wird. P_0 erhält als Ergebnis $r = \{x\}$ und P_1 erhält $r = \{y\}$. Beides ist nicht das korrekte Ergebnis $r = \emptyset$.

- Der Simulator für einen Wert $h(x)$ den Wert x finden kann, den das Random Oracle als Eingabe bekommen halt, als es $h(x)$ ausgegeben hat, wenn dieser existiert. Ein zweiter solcher Wert existiert nur mit vernachlässigbarer Wahrscheinlichkeit, wenn $h(x)$ als kollisionsresistent angenommen wird, denn die Wahrscheinlichkeit, dass das Random Oracle auf zwei verschiedene Anfragen dieselbe Antwort ausgibt ist vernachlässigbar. Der Simulator muss x zu durch P_i versendetem $h(x)$ finden können, damit er aus den Hashwerten, die P_i sendet, die eigentliche Eingabe extrahieren kann um sie einer idealen Funktionalität übergeben zu können.

Die ideale Funktionalität für sicheren Schlüsselaustausch Wenn zwei Parteien einen gemeinsamen Schlüssel benötigen können sie einen solchen mit einem Schlüsselaustauschverfahren erzeugen. Beispielsweise kann der Diffie-Hellman Schlüsselaustausch oder der RSA-Schlüsselaustausch für diesen Zweck verwendet werden. Auch kann direkt mit $\mathcal{F}_{smt}$ ein Schlüssel versendet werden. Für diese Arbeit wollen wir uns nicht auf ein Verfahren festlegen und definieren daher eine ideale Funktionalität, die die Anforderungen an ein Schlüsselaustauschverfahren beschriebt.

Diese ideale Funktionalität nennen wir $\mathcal{F}_{kex}$ und beschreiben diese in Programm 13. Wenn eine Partei P_S mit Hilfe von $\mathcal{F}_{kex}$ einen Schlüsselaustausch mit P_R durchführen will, wird $\mathcal{F}_{kex}$ den Simulator/Angreifer informieren, dass ein Schlüsselaustausch stattfindet. Im Gegensatz zu der Definition in [CK02] darf hier nicht, wenn P_S oder P_R korrupt sind, der Simulator/Angreifer den Schlüssel wählen. Außerdem gehen wir davon aus, dass der Schlüsselaustausch über einen authentifizierten Kanal stattfindet. Diese stärkere Definition des Schlüsselaustauschs lässt sich beispielsweise mit einem Blum-Coin-Toss ([Blu83]) über $\mathcal{F}_{smt}$ realisieren.

Programm 13 : $\mathcal{F}_{kex}$ Ideale Funktionalität für sicheren Schlüsselaustausch

```
1 receive message "new_key, P_R" from P_S do
2 |   draw k uniformly at random
3 |   send delayed output k to P_S while leaking
  |     ("key exchange is carried out", P_S, P_R) to the Adversary
4
5 receive message "get key" from P_R do
6 |   send delayed output k to P_R while leaking "receiver is fetching key" to the
  |     Adversary
```

3 Verbessertes Verfahren *protoRO*

Die folgenden Abschnitte stellen ein verbessertes Verfahren *protoRO* vor. Dieses soll mindestens so sicher sein, wie der "klassische" Ansatz, der in der Motivation in Abschnitt 1.1 beschrieben wurde, aber trotzdem zusätzliche Sicherheit garantieren, wenn höchstens eine der beteiligten Parteien korrumpiert ist. Der Fall, dass mehr als eine Partei korrumpiert ist, wird in dieser Arbeit nicht betrachtet.

Setting Im Protokoll *protoRO* gehen wir davon aus, dass ein Dienstanbieter T ein Computersystem betreibt, auf dem ein "Trusted Execution Processor" existiert, der durch $\mathcal{G}_{att}$ modelliert wird. Im Protokoll *protoRO* wird das Programm $\Psi_{protoRO}$ durch T als Enklave installiert. Weil die Ausführung auf einem "Trusted Execution Processor" stattfindet, werden alle Ausgaben von $\Psi_{protoRO}$ zusammen mit dem Programmcode signiert. Mit diesem Dienstanbieter T kommunizieren die Parteien P_0 und P_1, die den Schnitt ihrer Eingabemengen S_i berechnen wollen.

Eine wünschenswerte Eigenschaft des Verfahrens ist, dass nur T und nicht P_i Attested Execution Processors haben müssen. Das erlaubt, dass das Protokoll auch eingesetzt werden kann, wenn die beiden Parteien, die den Schnitt berechnen wollen, keinen Attested Execution Processor haben. Die Parteien P_i können zwischen verschiedenen Dienstanbietern einen auswählen, der Attested Execution Processors anbietet. Wenn eine Partei P_i keinen Attested Execution Processor hat, bringt das konzeptionelle Probleme mit sich, wenn diese Partei korrumpiert ist. Im Realen legt T eine Enklave an und P_i erhält echte Signaturen der Enklave. Im Idealen legt ein ehrliches T keine Enklave an, da T eine Dummy-Partei ist. Somit gibt es keine echten Enklavensignaturen und der Simulator kann keine gültigen Signaturen erzeugen, die von P_i angenommen werden. Dadurch kann die Umgebung Real von Ideal unterscheiden. In [PST17] heißt dieses Problem *"non-deniability issue"* (*Problem der nicht-Abstreitbarkeit*). In [PST17, S. 6.3] ist der Lösungvorschlag einen "Augmented Global CRS" anzunehmen und dann anstelle von Signaturen "non-interactive witness indistinguishable" (NIWI) Beweise auf diese Signaturen zu versenden. Dadurch kann für ein korruptes P_i der Beweis mithilfe des Identity Keys generiert werden und ehrliche Parteien brauchen trotzdem eine korrekte Signatur. Dieser Ansatz ist in *protoRO* auch umgesetzt.

Korruption Für unser Modell gehen wir davon aus, dass T manchmal den gesamten internen Zustand der Enklave beobachten kann. Wenn T ehrlich ist, wird er alle Geheimnisse der fehlerhaften Enklave ignorieren. Es ist also nur der Fall interessant, wenn T korrumpiert ist und die Enklave von T Informationen preisgibt. Diesen Fall meinen wir, wenn wir in der Arbeit T_{enclave} schreiben. In der Realität ist diese Situation z. B. durch fehlerhafte Implementation oder Seitenkanäle möglich. Außerdem bildet es auch die Bedrohung ab, dass der Prozessorhersteller möglicherweise Zugriff auf Geheimnisse in der Enklave hat. Man kann in einem korrumpierten T, dem die Enklave Informationen verrät, also sowohl einen Dienstanbieter sehen, der einen Angriff auf die Enklave kennt, als auch einen bösartigen Hardwarehersteller.

Ein weiterer Blickwinkel auf diese Art der Modellierung ist wie folgt: wenn man dem Prozessorhersteller misstraut, gibt es keinen direkten Anlass dem Hauptprozessor mehr zu vertrauen, als dem "Trusted Execution Processor". Typischerweise sind nämlich beide logischen Prozessoren Teil derselben Hardwareeinheit und kommen von dem selben Hersteller. Dieser Umstand ist in unserem Modell auch gut abgebildet: Wenn die Enklave (passiv) korrumpiert ist, wird immer auch T (der Hauptprozessor) korrumpiert.

Im Rahmen dieser Arbeit wird nur statische Korruption betrachtet. Der Angreifer darf sich zu Beginn des Ablaufs entscheiden ob er:

1. keinen korrumpiert,
2. P_0 oder P_1 aktiv korrumpiert,
3. T aktiv korrumpiert und die Enklave ehrlich bleibt (wir meinen diesen Fall, wenn wir sagen, dass der Angreifer die Partei T_{only} korrumpiert),
4. T aktiv korrumpiert und damit die Enklave belauscht (wir meinen diesen Fall, wenn wir sagen, dass der Angreifer die Partei T_{enclave} korrumpiert).

Der Angreifer sendet zu Beginn des Protokollablaufs (optional) eine *corrupt*-Nachricht an P_0, P_1 oder T. Der Angreifer hat die Kontrolle über das Netzwerk und darf sich bei jeder Nachricht entscheiden, ob diese zugestellt oder unterdrückt wird.

Ziele des Protokolls Um die Netzwerkkomplexität des Protokolls praktikabel zu halten, soll die Nachrichtengröße nicht von der Größe der Elemente in der Eingabemenge S_i abhängen, sondern immer nur Hashwerte über das Netzwerk versendet werden. Die Elemente der Eingabemenge können z. B Bilder sein. Zusätzlich soll zwischen den Parteien P_i die Kommunikation minimal gehalten werden.

Interessant ist auch die Frage, ob die Größe der Eingaben und die Größe des Schnitts schützenswert ist. In dieser Arbeit versuchen wir diese Größen vor dem Angreifer zu verbergen.

Die Eingabegröße ist prinzipiell nicht vollständig vor dem Angreifer zu verbergen, da der Angreifer die Menge an Netzwerkkommunikation beobachtet. Wir gehen daher von einer oberen Schranke ω für die Anzahl der Elemente in den Eingabemengen S_i. Implizit ist dies auch eine Schranke für die Größe des Schnitts, da für beliebige Mengen S_0, S_1 gilt: $|S_0 \cap S_1| \leq |S_0| \leq \omega$. Daher wird in *protoRO* sowohl S_i als auch der Schnitt auf ω viele Elemente mit zufälligen Elementen aufgefüllt. Dies schränkt das Protokoll ein, da nun die Parteien nicht mehr Eingabemengen mit beliebig vielen Elementen verwenden können.

Protokollablauf Im folgenden betrachten wir eine Skizze des Protokollablaufs von *protoRO*, wobei wir nicht auf jedes technische Detail eingehen. Für diese Beschreibung gehen wir davon aus, das alle Parteien ehrlich sind. An besonders interessanten Stellen gehen wir auf den Fall ein, dass eine der Parteien korrumpiert ist. Wenn wir von P_i sprechen gelten die Aussagen sowohl für P_0 als auch P_1.

Zu Beginn des Protokolls erhält eine der Parteien P_i die Eingabe S_i von der Umgebung. Die Partei fragt den Schlüssel *mpk* bei $\mathcal{G}_{att}$ an und speichert diesen. P_0 führt einen Schlüsselaustausch mit P_1 über die ideale Funktionalität $\mathcal{F}_{kex}$ durch. Beide erhalten den Schlüssel k. P_i konkateniert die Elemente seiner Eingabe S_i mit dem Schlüssel k und fragt für jedes $x_j \in S_i$ den Hashwert für $x_j \| k$ bei einem Random Oracle an und erhält die Menge s_i'. P_i füllt die Menge s_i' mit Padding auf. Die Partei P_i fragt den Common Reference String *crs*, den öffentlichen Schlüssel *epk* und den Verifikationsschlüssel *vk* bei $\mathcal{G}_{acrs}$ an und speichert diese Werte, um später im Protokollablauf NIWI-Beweise überprüfen zu können.

P_i beginnt den Schlüsselaustausch mit der Enklave von T, indem P_i eine Nachricht $m_i' := g^{a'}$ an den Dienstanbieter T sendet. Der Dienstanbieter T installiert eine Enklave mit Programmcode $\Psi_{protoRO}$. T erhält den eindeutigen Identifier *eid* der Enklave. T führt auf der Enklave *eid* *resume* mit der Nachricht m_i' von P_i aus. Die Enklave setzt den Schlüsselaustausch mit P_i fort, indem die Enklave $m_i := g^a$ generiert, zusammen mit der Nachricht m_i' von P_i signiert und an T ausgibt. Die Enklave hat zusätzlich die Möglichkeit in der Variable *bits* geheime Informationen, wie Zufall oder internen Zustand, auszugeben. Dieser Fall tritt nur ein, wenn die Enklave nicht ehrlich ist. Eine ehrliche Enklave befüllt die Variable *bits* nicht. Der Dienstanbieter T erzeugt das Chiffrat C_i mit der echten Signatur der Enklave und einem zufälligen Wert für *idk*. Außerdem berechnet er einen NIWI-Beweis π_i, in dem er beweist, dass in C_i die Signatur der Enklave korrekt ist. Der Dienstanbieter T gibt m_i zusammen mit dem NIWI-Bewies π_i, dem Chiffrat C_i und der *eid* der Enklave an P_i. T muss m' nicht zurück an P_i senden, da dieser Wert von P_i stammte. P_i überprüft den Beweis π_i. Wenn dieser nicht korrekt ist, bricht P_i ab und nimmt nicht weiter am Protokoll teil.

P_0 berechnet den gemeinsamen Schlüssel k_0 als $(m_0)^{a'}$. P_0 verschlüsselt s_0 mit dem Schlüssel

k_0 mit einem IND-CCA sicherem Verfahren und übergibt dieses an T. T übergibt dieses Chiffrat der Enklave eid, welche es entschlüsselt.

Wenn die Enklave von beiden Parteien P_i die Eingabe erhalten hat, berechnet sie auf den Hashwerten den Schnitt r' und gibt diesen aus. T übermittelt ihn gemeinsam mit dem jeweils zugehörigen NIWI-Beweis π_{ri} und dem Chiffrat C_{ri} an beide P_i, welche den NIWI-Beweis überprüfen und danach den echten Schnitt r aus r' ableiten, um ihn an die Umgebung auszugeben.

In Abbildung 3.1 wird der Fluss der Nachrichten im Protokoll *protoRO* gezeigt unter der Annahme eines Angreifers, der nicht in den Protokollablauf eingreift. Um den Fluss übersichtlich zu halten, sind Nachrichten zwischen P_0, P_1 und T direkt eingezeichnet und nicht über $\mathcal{F}_{smt}$ geleitet. Außerdem wird der Angreifer weggelassen sowie seine Bestätigungsnachrichten an $\mathcal{F}_{smt}$ und $\mathcal{F}_{kex}$. Außerdem fragen P_0 und P_1 mpk von $\mathcal{G}_{att}$ an, auch diese Nachrichten sind weggelassen worden.

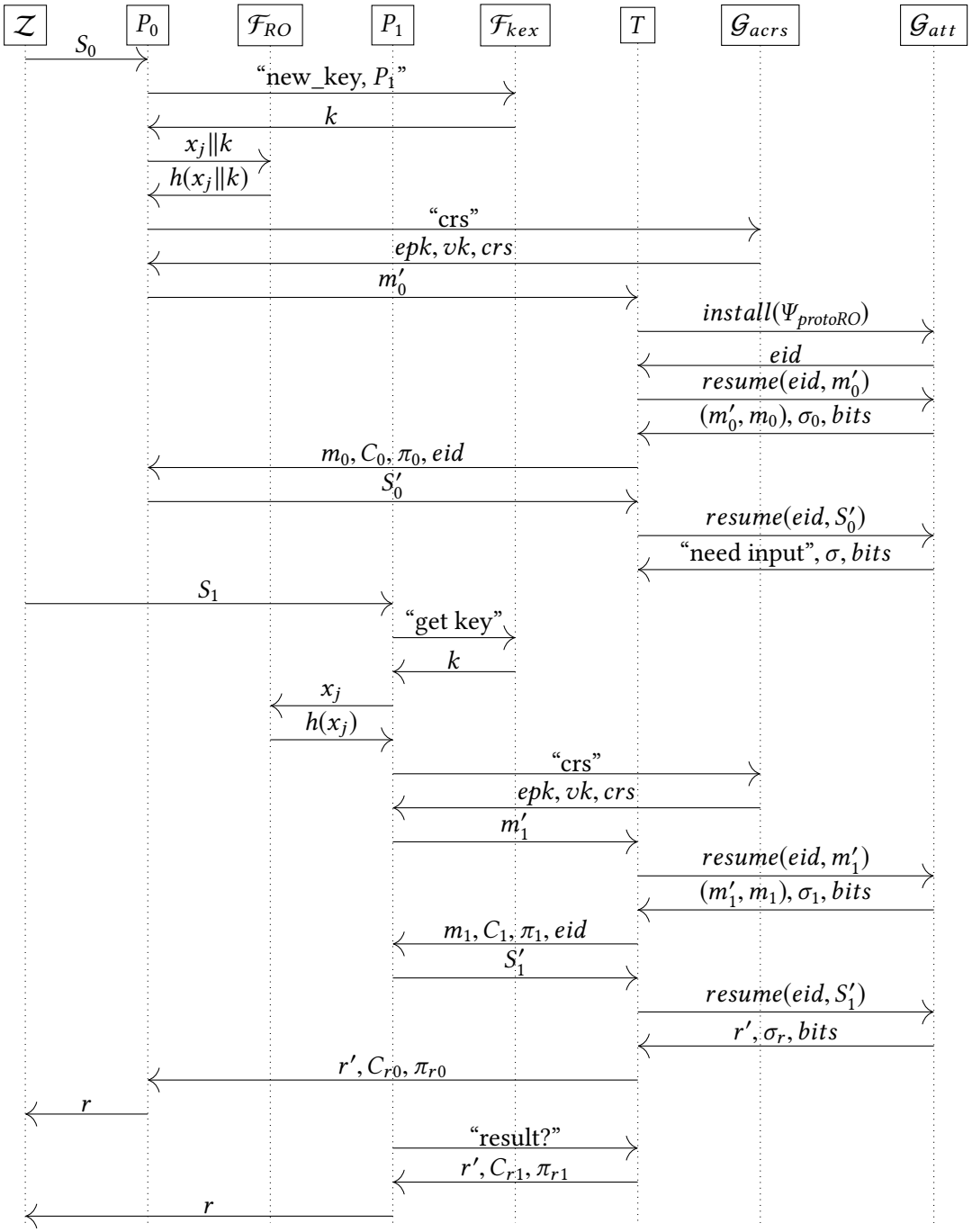

Abbildung 3.1: Protokollablauf von *protoRO* ohne Störung im Realen

3.1 Zu realisierende ideale Funktionalität für *protoRO*

Verfahren *protoRO* bietet nicht die starke Sicherheit, die in $\mathcal{F}_{PSI}$ aus Programm 22 für das Verfahren *protoHASH* modelliert ist. Diese schwächere Sicherheit ist in $\mathcal{F}_{PSI}^{weak}$ in Programm 14 abgebildet. $\mathcal{F}_{PSI}^{weak}$ wird von *protoRO* realisiert. In $\mathcal{F}_{PSI}^{weak}$ wird wiedergegeben, dass in Protokoll *protoRO* ein korrumpiertes T, dem die Enklave Informationen verrät, die Ausgabe r an die beiden Parteien P_i beeinflussen kann. Dafür wählt T (nachdem es die Größe des Schnitts erfahren hat) je zwei Zahlen a_i, b_i, wobei a_i bestimmt wie viele Elemente aus dem Schnitt r gewählt werden und b_i wie viele Elemente aus $S_i \setminus r$ oder Padding sind. Bei der zweiten Partei muss zusätzlich c_i angegeben werden, welches bestimmt, wie viele Elemente aus dem Schnitt in den Ausgaben beider Parteien gemeinsam sind. Weil ein reales korrumpiertes T, welches die Geheimnisse der Enklave lernt, die Elemente des Schnitts, die im Ergebnis landen, nicht selbst auswählt, darf der Simulator/Angreifer nur eine Anzahl senden und $\mathcal{F}_{PSI}^{weak}$ zieht die Elemente selbst. Für Elemente die nicht im Schnitt liegen kann ein korrumpiertes T, welches die Geheimnisse der Enklave lernt, nicht einmal entscheiden ob sie Padding sind oder nicht. Daher wird eine Menge Ω_i definiert, die dieses Padding darstellt. Nach dem Auswählen der b_i Elemente wird dieses Padding wieder entfernt, da im Realen P_i das Padding erkennt und vor seiner Ausgabe an die Umgebung entfernt. Diese Berechnung der Ausgaben wird in der Funktion *CalculateResult* durchgeführt die einmal für jedes P_i aufgerufen wird. Für ein korrumpiertes T mit ehrlicher Enklave werden die Werte von a_i und b_i von $\mathcal{F}_{PSI}^{weak}$ ignoriert und sind daher in den entsprechenden Simulatoren nicht angegeben.

Intuition für Berechnung von dem Ergebnis in *CalculateResult* Um ein besseres Verständnis für die Konstruktion von *CalculateResult* in $\mathcal{F}_{PSI}^{weak}$ zu erhalten, schauen wir uns genauer an, was ein Angreifer in *protoRO* tatsächlich beeinflussen kann. Wir betrachten den interessanten Fall, dass T korrumpiert ist und die Geheimnisse der Enklave lernt und geben das folgende Beispiel an: Wir nehmen an, dass die Eingabe und das Ergebnis auf $\omega = 5$ mit Padding aufgefüllt wird. Die Umgebung gibt P_0 die Eingabe $S_0 = \{a,b,c,d\}$ und P_1 die Eingabe $S_1 = \{a,b,c,e\}$. Die Parteien füllen diese Eingabe mit Padding auf und bilden Hashwerte der mit k konkatenierten Elemente. Nach diesen Aktionen hat P_0 die Menge $s_0' = \{A,B,C,D,X\}$ und P_1 die Menge $s_1' = \{A,B,C,E,Y\}$, dabei sind x und y Padding und $X = h(x\|k)$, $Y = h(y\|k)$. Diese Mengen werden mit einem Schlüssel verschlüsselt, den die entsprechende Partei mit der Enklave durch einen Diffie-Hellman Schlüsselaustausch ausgehandelt hat. Der Schlüsselaustausch mit der Enklave schützt nicht vor einem korrumpierten T,der die Geheimnisse der Enklave lernt, denn die Enklave verrät auch den Schlüssel an T. Damit sieht T die unverschlüsselten Mengen s_0' und s_1'. T kann die Mengen s_0' und s_1' schneiden und kann die Elemente drei Mengen

zuordnen: Elemente im Schnitt ($\{A,B,C\}$), Elemente, die nur in s_0' vorhanden sind, ($\{D,X\}$) und Elemente, die nur in s_1' vorhanden sind, ($\{E,Y\}$). Dabei kann T nicht zwischen den Elementen innerhalb einer Menge unterscheiden. T kann frei entscheiden, welche der Elemente aus s_0' er an P_0 zurücksendet und welche er durch neue Elemente als Padding ersetzt. In $\mathcal{F}_{PSI}^{weak}$ ist a_i die Anzahl der aus $\{A,B,C\}$ gewählten Elemente und b_i die Anzahl der aus $\{D,X\}$ bzw. $\{E,Y\}$ gewählten Elemente. Bei P_1 kann T sich zudem entscheiden, wie viele der Elemente, welche er an P_0 gesendet hat, er auch an P_1 sendet. Das wird in $\mathcal{F}_{PSI}^{weak}$ mit dem Parameter c_1 modelliert. Beispielsweise könnte der Angreifer $r_0' = \{C,D,1,2,3\}$ und $r_1' = \{C,A,Y,4,5\}$ wählen. Dabei sind 1,2,3,4,5 neue Elemente, die Padding darstellen. Die Parteien berechnen aus r_i' ihr Eregebnis wie folgt: Für jedes Element $x \in S_i$ wird geprüft ob $h(x\|k) \in r_i'$. Wenn dies der Fall ist, wird x dem Ergebnis r hinzugefügt. P_0 gibt $\{c,d\}$ und P_1 gibt $\{a,c\}$ an die Umgebung als Ergebnis zurück. Die Beeinflussung des Ergebnisses muss im Idealen genau so abgebildet sein.

P_i hat eine Möglichkeit zu erkennen, dass ein korrumpiertes T versucht hat den Schnitt zu vergrößern, wenn es in r_i Elemente des eigenen Paddings wieder zurück erhält. Bei einem ehrlichen T tritt dies nur mit vernachlässigbarer Wahrscheinlichkeit auf, da die Enklave das Padding in r' zufällig neu wählt. In *protoRO* prüft P_i nicht ob er das eigene Padding zurück erhält.

Im Idealen erhält der Simulator r_i' und wählt basierend darauf: $a_0 = 1$, $b_0 = 1$, $a_1 = 1$, $b_1 = 1$ und $c_1 = 1$. $\mathcal{F}_{PSI}^{weak}$ ruft zunächst $CalculateResult(0,1,1,0)$ auf. Dieses berechnet $\Omega := \{\perp_0\}$. Dies modelliert das eine Element Padding, welches P_0 seiner Eingabe hinzugefügt hat. Dann zieht $\mathcal{F}_{PSI}^{weak}$ intersectionElements $= \{c\}$, zufällig aus $\{a,b,c\}$, und zusätzlich $\{d\}$, zufällig aus $\{d,\perp_0\}$, und gibt dann $r_0 = \{c\} \cup \{\} \cup \{d\} = \{c,d\}$ an P_0 aus. Dabei werden in sameElements $:= \{c\}$ die Elemente aus dem Schnitt, die an P_0 versendet werden. Für P_1 wird $CalculateResult(1,1,1,1)$ aufgerufen. Auch hier ist $\Omega = \{\perp_0\}$. $\mathcal{F}_{PSI}^{weak}$ zieht intersectionElements $= \{a\}$ zufällig aus $\{a,b\}$, $\{c\}$ aus sameElements und $\{\perp_0\}$ aus $\{\perp_0,y\}$. Dann wird $r_1 = \{a\} \cup \{c\} \cup (\{\perp_0\} \setminus \Omega) = \{a,c\}$ berechnet und an P_1 ausgegeben. Damit ist die Ausgabe von $\mathcal{F}_{PSI}^{weak}$ genau wie im Realen.

Programm 14 : $\mathcal{F}_{PSI}^{weak}$ Die ideale Funktionalität für sicheren gemeinsamen Schnitt für Verfahren *protoRO*

Function *CalculateResult*(i, a_i, b_i, c_i)
  if T *is corrupted and enclave leaks information* **then**
    $\Omega_i \coloneqq \{\bot_0, \ldots, \bot_{\omega-|S_i|-1}\}$
    intersectionElements $\coloneqq a_i$ distinct random elements from $(r \setminus$ sameElements$)$
    $r_i \coloneqq$ intersectionElements $\cup$ (c_i distinct random elements from sameElements) $\cup$ ((b_i distinct random elements from $(S_i \setminus r) \cup \Omega_i) \setminus \Omega_i$)
    sameElements $\coloneqq$ intersectionElements
  else
    $r_i \coloneqq r$
  end
end

Phase: first input

receive message S_0 **from** P_0 **do**
  store S_0
  send "input received" **to the Adversary**

Phase: second input

receive message S_1 **from** P_1 **do**
  $r \coloneqq S_0 \cap S_1$
  if T *is corrupted and enclave leaks information* **then**
    send $|r|$ **to the Adversary**
    receive message "size received" **from the Adversary**
  end
  send "intersect" **to the Adversary**

Phase: send result

sameElements $\coloneqq \emptyset$
receive message "result to P_i", (a_i, b_i) **from the Adversary do**
  CalculateResult$(i, a_i, b_i, 0)$
  send r_i **to** P_i

receive message "execution resumed" **from** P_1 **do**
  send "intersect 2" **to the Adversary**

receive message "result to P_{1-i}", $(a_{1-i}, b_{1-i}, c_{1-i})$ **from the Adversary do**
  CalculateResult$(1 - i, a_{1-i}, b_{1-i}, c_{1-i})$
  send r_{1-i} **to** P_{1-i}

In Abbildung 3.2 wird der Fluss der Nachrichten im Idealen gezeigt unter der Annahme eines Angreifers, der nicht in den Protokollablauf eingreift und keine Partei korrumpiert.

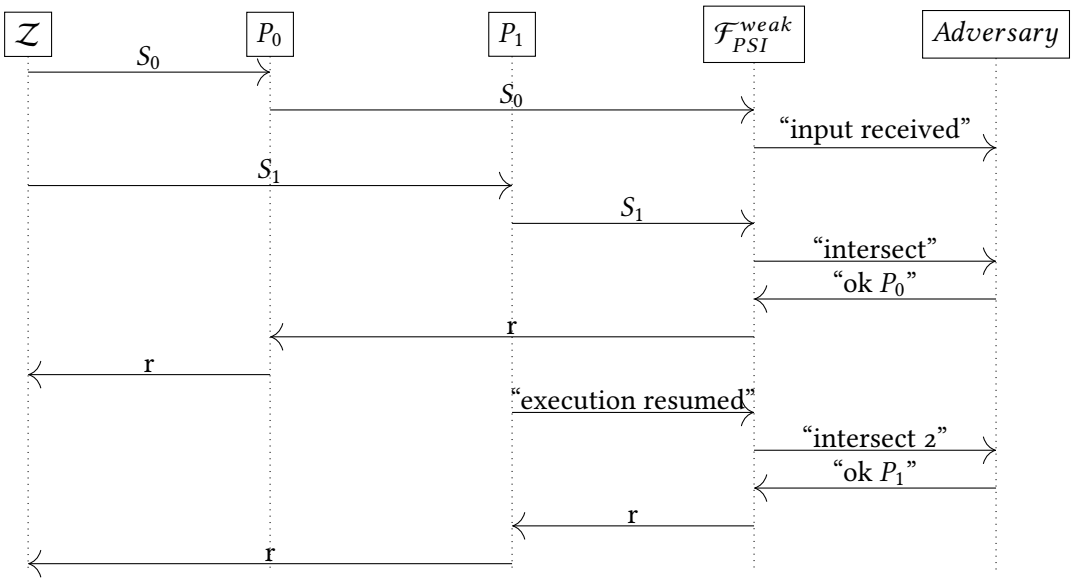

Abbildung 3.2: Protokollablauf von *protoRO* ohne Störung im Idealen

3.2 Programm in $\mathcal{G}_{att}$: $\Psi_{protoRO}$

Die Parteien P_0 und P_1 wollen gemeinsam das Programm $\Psi_{protoRO}$ in einer Enklave ausführen und dann mit dieser Enklave einen Schlüsselaustausch durchführen. $\Psi_{protoRO}$ ist in Programm 15 beschrieben.

Es wird immer ein Programmsegment von einer Eingabe bis zur nächsten Ausgabe ausgeführt. Dann wartet die Enklave, bis *resume* auf $\mathcal{G}_{att}$ aufgerufen wird um mit der Ausführung fortzufahren.

Zunächst wird der erste Teil des Schlüsselaustauschs von der Partei P_0 empfangen und der zweite Teil generiert. Dabei ist g ein Generator einer Gruppe $\mathcal{G}$, in der die DDH-Annahme gilt. Die von P_0 erhaltene Nachricht wird mit ausgegeben und von $\mathcal{G}_{att}$ signiert. Damit kann P_0 erkennen, ob die Nachricht korrekt bei Enklave ankam oder durch T verfälscht wurde. Die von P_0 erhaltene Eingabe wird mit Entschlüsselungsalgorithmus *Dec* eines IND-CCA sicheren, symmetrischen Verschlüsselungsverfahrens entschlüsselt. Analoge Schritte werden für die Nachrichten von der Partei P_1 durchgeführt. Bevor der Schnitt berechnet wird, überprüft die Enklave, ob das erste Element der beiden Eingaben übereinstimmt. Damit wird sichergestellt, dass die Enklave keine Information über die Eingabe preisgibt, wenn die beiden Eingaben nicht

tatsächlich von den Parteien P_i kommen. Das erste Element ist dabei nicht Teil der echten Eingabe von P_i sondern wird explizit für diese Prüfung hinzugefügt. Gegen ein korrumpiertes T, welches die Geheimnisse der Enklave lernt, bietet diese Konstruktion keinen Schutz. Ein solches T lernt den mit der Enklave ausgetauschten Schlüssel und kommt so an das erste Element der Eingabe von P_i.

Zuletzt wird der Schnitt berechnet und auf feste Länge mit zufälligen Elementen aufgefüllt um einem Angreifer nicht die Möglichkeit zu geben, die Schnittgröße zu erfahren. Diese Menge wird sortiert und ausgegeben.

Programm 15 : $\Psi_{protoRO}$ Programm in $\mathcal{G}_{att}$

Phase: first key exchange

receive message m_0' **do**
  draw $a, b \in \{0, \ldots, |\mathcal{G}| - 1\}$ **uniformly at random**
  $m_0 := g^a$
  return (m_0', m_0)

Phase: recieved first input

receive message S_0' **do**
  $k_0 := (m_0')^a$
  $s_0 := Dec_{k_0}(S_0')$
  return "need input"

Phase: second key exchange

receive message m_1' **do**
  $m_1 := g^b$
  return (m_1', m_1)

Phase: received second input

receive message S_1' **do**
  $k_1 := (m_1')^b$
  $s_1 := Dec_{k_1}(S_1')$
  if *the first element of* s_0 *and* s_1 *is not equal* **then**
    abort
  end

Phase: calculate result

  $r' := s_0 \cap s_1$
  pad r' with random values up to $(\omega + 1)$ elements
  sort everything but the first element of r'
  return r'

3.3 Programmcode der Partei P_i für Verfahren *protoRO*

In Programm 16 ist der Code der Partei P_i dargestellt. Die Partei P_i handelt mit der Partei P_{i-1} durch $\mathcal{F}_{kex}$ einen Schlüssel k aus. In der Zeile 11 bis Zeile 18 in Programm 16 bereitet P_i die Eingabe an die Enklave vor indem für jeder Wert der Eingabe einzeln mit k konkateniert wird und der Hashwert davon bei $\mathcal{F}_{RO}$ angefragt. Die sich ergebende Menge wird mit zufälligen Hashwerten (Padding) bis zur Größe ω aufgefüllt. Danach werden alle Elemente der Menge sortiert. Als erstes Element wird der Menge ein spezielles Element $h(\perp \| k)$ hinzugefügt, welches für P_i und P_{i-1} gleich sein muss. Dabei bezeichnet $\perp$ ein Element, welches nicht in S_i enthalten sein kann. P_i startet den Schlüsselaustausch mit der Enklave über T und prüft dann, dass der Schlüsselaustausch mit einer Enklave mit Programm $\Psi_{protoRO}$ durchgeführt wurde. Mit diesem Schlüssel wird die vorbereitete Eingabe verschlüsselt und an die Enklave gesendet. Wenn P_i das Ergebnis mit NIWI-Beweis erhält, prüft er zuerst den Beweis und das erste Element des Schnitts. Danach schneidet er das Ergebnis mit seiner eigenen Eingabe, in dem er erneut Hashwerte seiner Eingabe konkateniert mit k bildet und prüft ob diese im Enklavenergebnis liegen. Die so erhaltene Menge meldet er an die Umgebung.

Der Code von Partei P_0 und Partei P_1 ist sehr ähnlich. Ein Unterschied ist dadurch bedingt, dass eine der Parteien $\mathcal{F}_{kex}$ initiieren muss und bezieht sich auf die Zeile 6 in Programm 16. Ein anderer Unterschied ist, dass P_1 in Zeile 31 sein Ergebnis von T anfragen muss. Diese Situation entsteht, weil die Umgebung das Ergebnis von P_0 erhalten hat und entscheiden muss, an welche Partei den Kontrollfluss übergeben wird. Es ist ungeschickt wenn die Umgebung direkt an T den Kontrollfluss übergibt, denn im Idealen würde dann T direkt mit der Idealen Funktionalität kommunizieren. Es ist aber natürlicher $\mathcal{F}_{int'}$ als eine ideale Funktionalität zur Berechnung des Schnitts zwischen zwei Parteien zu definieren und dabei T so weit wie möglich aus dieser Definition herauszuhalten.

Verschiedene Beobachtungen waren für die Konstruktion des Protokolls relevant:

Warum wird ein Schlüssel als zusätzliche Eingabe der Hashfunktion benötigt? Der Schlüssel in der Eingabe der Hashfunktion ermöglicht es dem Protokoll im besten Fall sicherer als der klassische Ansatz zu sein. Wenn die Enklave eine Vermutung für einen Wert $a \in S_0$ hat, könnte die Enklave im hashbasierten Ansatz diese Vermutung überprüfen, indem sie prüft, ob $h(a)$ in der eingegebenen Menge ist. Der zusätzliche unbekannte Schlüssel schafft hier Abhilfe, weil die Enklave den Schlüssel k nicht kennt, und so eine Vermutung nicht prüfen kann.

Warum müssen die Hashwerte der Elemente gebildet werden? P_i verwendet $h(x_j \| k)$ anstelle vom $x_j \| k$ damit dieses Protokoll mindestens so sicher ist wie der "klassische" hash-

Programm 16 : Programmcode der Partei P_i

Phase: setup

on input S_i **from** *Environment* **do**
  send subroutine input "getpk" **to** $\mathcal{G}_{att}$

on input mpk **from** $\mathcal{G}_{att}$ **do**
  if $i = 0$ **then**
    send subroutine input "new key, P_1" **to** $\mathcal{F}_{kex}$
  else
    send subroutine input "get key" **to** $\mathcal{F}_{kex}$
  end

Phase: prepare input for enclave

on subroutine output k **from** $\mathcal{F}_{kex}$ **do**
  $s_i := \emptyset$
  for $x_j \in (\{\perp\} \cup S_i)$ **do**
    send subroutine input $x_j \| k$ **to** $\mathcal{F}_{RO}$
    on subroutine output $h(x_j \| k)$ **from** $\mathcal{F}_{RO}$ **do**
      append $h(x_j \| k)$ to s_i
  end
  draw elements h_t **uniformly at random** and add h_t to s_i until $|s_i| = \omega$
  sort everything but the first element of s_i
  send subroutine input "crs" **to** $\mathcal{G}_{acrs}$

Phase: key exchange

receive message (epk, vk, crs) **from** $\mathcal{G}_{acrs}$ **do**
  draw $a' \in \{0, \ldots, |\mathcal{G}| - 1\}$ **uniformly at random**
  $m_i' := g^{a'}$
  send m_i' **to** T

Phase: encrypt and send input to enclave

receive message (m_i, C_i, π_i, eid) **from** T **do**
  assert $NIWI.Verify(P_i, (mpk, crs), (eid, \Psi_{protoRO}, (m_i', m_i)), C_i, \pi_i) = 1$
  $k_i' := m_i^{a'}$
  $S_i' := Enc_{k_i'}(s_i)$
  send S_i' **to** T

Phase: receive enclave output and calculate result for environment

if $i = 1$ **then**
  when the Environment resumes execution do
    send "result?" **to** T
end

receive message (r', C_r, π_r) **from** T **do**
  assert $NIWI.Verify(P_i, (mpk, crs), (eid, \Psi_{protoRO}, r'), C_r, \pi_r) = 1$
  if $h(\perp \| k)$ *is not the first element of* r' **then**
    abort
  end
  calculate r, by iterating over $a \in S_i$ and adding a to r, iff $h(a||k) \in r'$
  return r **to** *Environment*

basierte Ansatz. Das ist selbst dann der Fall, wenn alle von P_i verschiedenen Parteien miteinander kooperieren.

Warum müssen die Hashwerte vor dem Versenden an die Enklave sortiert werden? Partei P_i muss s_i' sortieren (oder mischen) damit die Enklave nichts durch die Reihenfolge der Elemente in s_i' lernt.

Die Umgebung wählt zufällig, ob als Eingabe $S_0 := \{1, 3\}, S_1 := \{2, 3\}$ oder $S_0 := \{1, 2\}, S_1 := \{1, 3\}$ gesendet wird. Die Enklave erhält dann die Hashwerte der einzelnen Elemente. Wenn die Hashwerte nicht sortiert (oder zumindest gemischt) werden, kann die Enklave (und damit der reale Angreifer, wenn er sich entscheidet die Enklave zu überwachen) herausfinden, welches Eingabenpaar gewählt wurde. Im einen Fall werden die zweiten Hashwerte gleich sein und im anderen die ersten.

Im Idealen erhält Sim von $\mathcal{G}_{att}$ nur die Größe des Schnitts, die aber in beiden Fällen 1 ist. Daher kann der Simulator nicht simulieren.

Wenn dieses Protokoll real angewendet werden soll, um die gemeinsamen Telefonnummern im Adressbuch zweier Mobiltelefone zu bestimmen, zeigt sich der Angriff in diesem Szenario: P_0 und P_1 haben beide Kontakte aus zwei verschiedenen Ländern L_1 und L_2. Die Landesvorwahl von L_1 ist kleiner als die von L_2. Beide erhalten die Telefonnummern sortiert, bevor dieses Protokoll gestartet wird. Nun kann die Enklave erfahren, ob P_0 und P_1 eher mehr gemeinsame Freunde in L_1 oder in L_2 haben. Das kann verhindert werden, indem die Eingaben zuerst gemischt (oder ihre Hashwerte sortiert) werden, bevor sie in diesem Protokoll verwendet werden.

Zudem erlaubt das Sortieren der Enklave den Schnitt in $O(|S_0| + |S_1|)$ zu berechnen. Das ist ein Vorteil des Sortierens gegenüber dem Mischen.

Wie werden der Nachrichten der Enklave authentifiziert? Um den public key zur Verifikation von Signaturen von $\mathcal{G}_{att}$ zu bekommen, ruft Partei P_i "getpk" auf. Die Parteienliste reg von $\mathcal{G}_{att}$ mit $P_i \notin reg$ stellt sicher, dass P_i nur "getpk" auf $\mathcal{G}_{att}$ aufrufen kann und keine weiteren Funktionen. Das modelliert, dass diese Parteien keinen Attested Execution-Prozessor haben müssen, sondern nur der Dienstanbieter T.

P_i erhält Nachrichten, die von T weitergeleitet wurden und aus einer Enklave stammen, die Programm $\Psi_{protoRO}$ ausführt. Diese enthalten immer auch einen NIWI-Beweis über die Signatur der Enklave. Weil T $idk[P_i]$ nicht kennt, wird durch diesen Beweis die Kommunikation authentifiziert und sichergestellt, dass die Nachrichten aus einer Enklave stammen, die das korrekte Programm ausführt.

Warum werden die Eingaben für die Enklave verschlüsselt? Durch die Verschlüsselung der Hashwerte der Elemente der Eingabe für die Enklave können die Hashwerte vor T bei nicht-transparenter Enklave verborgen werden. Anderenfalls könnte T den Schnitt der Hashwerte berechnen und die Größe des Schnitts lernen.

Wie wird das Ergebnis berechnet? Um den eigentlichen Schnitt zu berechnen muss P_i seine Eingabe wieder verwenden, da die Funktion, die $\mathcal{F}_{RO}$ berechnet, nicht invertiert werden kann. Wenn $\Psi_{protoRO}$ bei der Berechnung des Schnitts zufällige Werte hinzugefügt hat um vor T und einem Netzwerkangreifer die Größe des Schnitts zu verbergen, werden diese in diesem Schritt entfernt.

3.4 Programmcode der Partei T für Verfahren *protoRO*

Partei T modelliert den Dienstanbieter, der einen Attested Execution Processor bereitstellt. Seine Aufgabe besteht großteils darin Nachrichten von P_0 und P_1 von und zur Enklave zu leiten. Außerdem muss T, wegen dem *Problem der nicht-Abstreitbarkeit*, die Signaturen der Enklave vor P_i verbergen. T verbirgt die Signaturen der Enklave, indem nicht die Signaturen sondern NIWI-Beweise auf diese Signaturen weitergeleitet werden. Da T korrekte Signaturen kennt, kann er gültige NIWI-Beweise ohne $idk[P_i]$ berechnen. Sein Programm ist in Programm 17 dargestellt.

Im Wesentlichen installiert T nur eine Enklave mit Programm $\Psi_{protoRO}$ und leitet Nachrichten zwischen P_i und der Enklave weiter. Des weiteren muss er sicherstellen, dass P_i keine Enklavensignatur lernt, indem er NIWI-Beweise berechnet. Die vom NIWI-Beweissystem verwendete Sprache ist definiert als:

$$\{(msg, P, C) \mid (r, \sigma, idk[P]) : C = PKE.Enc_{epk}((\sigma, idk[P]), r)$$

$$\wedge (Verify_{mpk}(msg, \sigma) = 1 \vee \Sigma.Verify_{vk}(P, idk[P]) = 1)\}$$

Man kann erkennen, dass in C immer σ oder $idk[P]$ korrekt sein muss. Ein ehrliches T kann ein gültiges σ und etwas beliebiges als $idk[P_i]$ verschlüsseln und den Verschlüsselungszufall r speichern. Sei msg die Ausgabe der Enklave. Dann ist $(r, \sigma, idk[P_i])$ ein Zeuge dafür, dass (msg, P_i, C) in der Sprache ist. Das kann von $NIWI.Prove$ benutzt werden um einen Beweis für P_i zu erzeugen.

Programm 17 : Programmcode der Partei T

Phase: setup and first key exchange

receive message m'_0 **from** P_0 **do**
  send subroutine input "crs" **to** $\mathcal{G}_{acrs}$

receive message $(epk, \_, crs)$ **from** $\mathcal{G}_{acrs}$ **do**
  send subroutine input $install(\Psi_{protoRO})$ **to** $\mathcal{G}_{att}$

on subroutine output eid **do**
  store eid
  send subroutine input $resume(eid, m'_0)$ **to** $\mathcal{G}_{att}$

on subroutine output $((m'_0, m_0), \sigma_0, bits)$ **from** $\mathcal{G}_{att}$ **do**
  draw idk, ξ **uniformly at random**
  $C_0 := Enc_{epk}((\sigma_0, idk), \xi)$
  $\pi_0 := NIWI.Prove(crs, (P_0, (eid, \Psi_{protoRO}, (m'_0, m_0)), C_0), (\xi, \sigma_0, idk))$
  send (m_0, C_0, π_0, eid) **to** P_0

Phase: first input for enclave

receive message S'_0 **from** P_0 **do**
  send subroutine input $resume(eid, S'_0)$ **to** $\mathcal{G}_{att}$

on subroutine output ("need input", $\sigma, bits$) **from** $\mathcal{G}_{att}$ **do**
  yield the execution **to** *Environment*

Phase: second key exchange

receive message m'_1 **from** P_1 **do**
  send subroutine input $resume(eid, m'_1)$ **to** $\mathcal{G}_{att}$

on subroutine output $((m'_1, m_1), \sigma_1, bits)$ **from** $\mathcal{G}_{att}$ **do**
  draw idk, ξ **uniformly at random**
  $C_1 := Enc_{epk}((\sigma_1, idk), \xi)$
  $\pi_1 := NIWI.Prove(crs, (P_1, (eid, \Psi_{protoRO}, (m'_1, m_1)), C_1), (\xi, \sigma_1, idk))$
  send (m_1, C_1, π_1, eid) **to** P_1

Phase: second input for enclave

receive message S'_1 **from** P_1 **do**
  send subroutine input $resume(eid, S'_1)$ **to** $\mathcal{G}_{att}$

Phase: send result

on subroutine output $(r', \sigma_r, bits)$ **from** $\mathcal{G}_{att}$ **do**
  draw idk, ξ **uniformly at random**
  $C_{r0} := Enc_{epk}((\sigma_r, idk), \xi)$
  $\pi_{r0} := NIWI.Prove(crs, (P_0, (eid, \Psi_{protoRO}, r'), C_{r0}), (\xi, \sigma_r, idk))$
  send (r', C_{r0}, π_{r0}) **to** P_0

receive message "result?" **from** P_1 **do**
  draw idk, ξ **uniformly at random**
  $C_{r1} := Enc_{epk}((\sigma_r, idk), \xi)$
  $\pi_{r1} := NIWI.Prove(crs, (P_1, (eid, \Psi_{protoRO}, r'), C_{r1}), (\xi, \sigma_r, idk))$
  send (r', C_{r1}, π_{r1}) **to** P_1

3.5 Man-in-the-Middle durch korrumpiertes T mit ehrlicher Enklave

Im Protokoll *protoRO* setzt P_i $h(\perp \| k)$ an den Anfang der an die Enklave übermittelten Eingabe. Die Enklave bricht die Berechnung ab, wenn dieser Wert für beide Parteien nicht gleich ist. Bei einem korrumpierten T mit ehrlicher Enklave sind sich P_0 und P_1 dadurch sicher, dass sie nur mit sich gegenseitig den Schnitt ausrechnen oder die Berechnung abgebrochen wird. Ohne diese Überprüfung ist das Protokoll nicht sicher:

Die Umgebung wählt $S_0 = S_1 = \{1\}$. T verbindet P_1 aber nicht mit der Enklave sondern simuliert den Code von P_1 mit leerer Eingabemenge. Der Schlüssel k, den T nicht hat, wird hierfür nicht benötigt. Die Enklave berechnet einen leeren Schnitt und gibt ihn an P_0 zurück. P_1 wird das Protokoll nicht erfolgreich beenden.

Eine Idee, dies zu unterbinden wäre, dass P_i den von der Enklave signierten Wert für *eid* vergleicht. P_0 und P_1 würden dann den Wert von *eid* austauschen und abbrechen, wenn die Werte nicht übereinstimmen. Dabei ist wichtig, dass dies erst geschieht wenn beide den Schlüsselaustausch in die Enklave abgeschlossen haben, und sich somit sicher sind, dass einer der Schlüssel in der Enklave *eid* ihrer ist. Dadurch wird die ehrliche Enklave Chiffrate von einem korrumpiertem T nicht annehmen.

3.6 Sicherheitsbeweis für *protoRO*

Für die verschiedenen Korruptionssituationen werden jeweils einzelne Simulatoren angegeben. Das ist möglich, weil wir von statischer Korruption ausgehen, d. h. ein Angreifer muss sich vor Beginn des Protokolls entscheiden ob und wen er korrumpieren will. Abhängig davon wählt der Simulator die entsprechende Variante. Der Fall, dass keine Partei korrumpiert ist und der Angreifer nur Netzwerknachrichten abfangen kann, ist in Unterabschnitt 3.6.1 beschrieben. In Unterabschnitt 3.6.2 ist der Fall beschrieben, dass Partei P_0 korrumpiert ist. Für P_1 ist ein analoger Sicherheitsbeweis möglich, der Simulator dazu ist in Anhang 1 angegeben. Partei T kann auf zwei verschiedene Arten korrumpiert werden. In Unterabschnitt 3.6.3 wird der Beweis geführt für den Fall, dass T korrumpiert ist, aber die Enklave ehrlich bleibt. Dies entspricht der Korruption des Cloud-Betreibers, der nicht in den Zustand der Enklave schauen kann. In Unterabschnitt 3.6.3 wird auch der Beweis für ein korrumpiertes T, welches die Geheimnisse der Enklave lernt, betrachtet, da nur kleine Änderungen am Simulator benötigt wurden. Dies entspricht der Situation, dass der Prozessorhersteller des Cloud-Betreibers korrupt ist.

Intern simuliert der Simulator den realen Angreifer $\mathcal{A}$. Immer wenn $\mathcal{A}$ mit anderen Parteien interagieren will, wird diese Nachricht zunächst dem Simulator vorgelegt, und dieser

entscheidet wie weiter verfahren werden soll. Wenn der Simulator eine Nachricht, die der Angreifer versenden will erwartet, ist im Pseudocode "**on** *message* **from** $\mathcal{A}$ **do**" angegeben. Die erwartete Nachricht ist dabei in *message*. Die Simulation von $\mathcal{A}$ wird damit pausiert (und der Simulator selbst ist an der Reihe). Soll $\mathcal{A}$ eine Antwort übermittelt werden, steht "Pass *message* **to** $\mathcal{A}$". Damit wird die Simulation von $\mathcal{A}$ fortgesetzt.

Korruption wird dadurch modelliert, dass die korrumpierte Partei alle Eingaben an den Angreifer/Simulator übermittelt und Antworten des Angreifers/Simulators unverändert weiterleitet. Eine Nachricht $corr_{in}(m$ **from** P_S **to** $P_R)$ bedeutet, dass die korrumpierte Partei P_R eine Nachricht m von P_S empfangen hat. Eine Nachricht $corr_{out}(m$ **from** P_S **to** $P_R)$ weist die korrumpierte Partei P_S an, eine Nachricht m an P_R zu senden. Bei Korruption der Partei P_i wird davon ausgegangen, dass der Angreifer durch die korrumpierte Partei jederzeit beliebige Nachrichten mit der Umgebung austauschen kann.

Da Netzwerknachrichten zwischen den Parteien über $\mathcal{F}_{smt}$ übermittelt werden, müssen dem Angreifer diese Nachrichten nicht direkt simuliert werden, sondern nur ihre Länge angegeben werden. Nachrichten an korrumpierte Parteien müssen vollständig simuliert werden, da diese Parteien durch $\mathcal{F}_{smt}$ die Nachricht tatsächlich übermittelt bekommen. Für die Längen der Elemente wird definiert: l_{dh} Länge der Darstellung eines Gruppenelements von $\mathcal{G}$, l_{proof} Länge der Ausgabe von $NIWI.Proof$ zusammen mit dem zugehörigem Chiffrat, l_{hash} Länge eines Hashwertes, l_{eid} Länge eines Enklavenidentifiers eid, l("identifier") Länge des Schlüsselworts "identifier", $e(l)$ Länge des Chiffrates eines Klartexts der Länge l mit Enc.

Um den Pseudocode übersichtlicher zu halten, entfällt die Überprüfung ob $\mathcal{A}$ zulässt, dass eine Nachricht gesendet wird, wenn der Sender oder der Empfänger dieser Nachricht korrumpiert ist. Wird ein Nachricht von oder an eine korrumpierte Partei über $\mathcal{F}_{smt}$ abgefangen, hätte der Angreifer sie auch selbst nicht senden bzw. bei Empfang ignorieren können. Das explizite Abfangen bietet also für korrumpierte Parteien keinen Nutzen und wird in den Simulatoren nicht abgebildet.

Manche Nachrichten im Protokoll können von korrumpierten Parteien in anderer Reihenfolge als angegeben gesendet werden. Um das Protokoll natürlich und realitätsnah zu halten wurde diese Reihenfolge nicht durch zusätzliche Nachrichten und Sequenznummern eingeschränkt. Das bedeutet aber auch, dass ein Angreifer Netzwerknachrichten bevor das Ergebnis von der Enklave berechnet wurde beliebig tauschen kann, solange für die ehrlichen Parteien die Reihenfolge jeweils unverändert bleibt. Netzwerknachrichten an ehrliche Teilnehmer müssen in derselben Reihenfolge bleiben, denn sonst fällt die Änderung der ehrlichen Partei auf. Auch Nachrichten, bei denen inhaltliche Abhängigkeiten bestehen dürfen nicht vertauscht werden. Wenn beispielsweise P_0 korrumpiert ist, heißt das konkret, dass Anfragen an $\mathcal{F}_{RO}$ und $\mathcal{G}_{acrs}$ zu beliebigen Zeitpunkt erfolgen können. An $\mathcal{F}_{kex}$ darf nur eine Anfrage gestellt werden und

diese muss gesendet werden, bevor P_1 seine Anfrage senden will. Für das Übermitteln des Ergebnisses gilt eine analoge Überlegung. Wenn der simulierte Angreifer eine Nachricht sendet, die nicht vom Simulator erwartet wird, wird sie ignoriert.

Alle im Sicherheitsbeweis vorgestellten Simulatoren haben einen gemeinsamen Codeblock zu beginn. Dieser ist in Programm 18 angegeben. Zum einen startet dieser Code die lokale Simulation des Angreifers. Zum anderen leitet er alle Nachrichten zwischen Umgebung und Angreifer unverändert weiter. Dieser Code wird mit $\mathcal{P}$ und $\mathcal{P}'$ parametrisiert. Dabei ist $\mathcal{P}$ die korrumpierte Partei (oder $\perp$ wenn keiner korrumpiert ist) und $\mathcal{P}'$ die Korruptionsart. Dies ist wichtig für T, da hier zwei verschiedene Arten der Korruption möglich sind. Diese Weiterleitungen sind beliebig oft möglich, was durch den $*$ am entsprechenden Block gekennzeichnet ist.

Programm 18 : $\mathrm{Sim}_{\square}^{protoRO}(\mathcal{P}, \mathcal{P}')$ Gemeinsamer Code für Setup aller Simulatoren

```
1  on input m from Environment do
2  |  start a local simulation of 𝒜 with input m
3
4  *on input m from Environment do
5  |  pass m to 𝒜
6
7  *on m to Environment from 𝒜 do
8  |  return m to Environment
9
10 if 𝒫 ≠ ⊥ then
11 |  on corr_𝒫' to 𝒫 from 𝒜 do
12 |  |  send corr_𝒫' to 𝒫
13 |
14 |  *on corr_out(m from 𝒫 to 𝒢_att, ℱ_RO) from 𝒜 do
15 |  |  send corr_out(m from 𝒫 to 𝒢_att, ℱ_RO) to 𝒫
16 |
17 |  *receive message corr_in(m from 𝒢_att, ℱ_RO to 𝒫) from 𝒫 do
18 |  |  pass corr_in(m from 𝒢_att, ℱ_RO to 𝒫) to 𝒜
19 end
```

3.6.1 Keine Partei ist korrumpiert

In Programm 19 ist der Fall beschrieben, dass der Angreifer am Anfang des Ablaufs sich dafür entscheidet keine Partei zu korrumpieren. Dabei befindet sich aus Sicht der Umgebung das ideale Experiment zu jedem Zeitpunkt in demselben Zustand, in dem auch das reale Protokoll wäre.

Satz 3.1. *Sei* (Enc, Dec) *ein IND-CCA sicheres symmetrisches Verschlüsselungsverfahren. Sei das Signaturverfahren von* $\mathcal{G}_{att}$ *EUF-CMA sicher. Sei das Signaturverfahren von* $\mathcal{G}_{acrs}$ *EUF-CMA sicher und das Verschlüsselungsverfahren von* $\mathcal{G}_{acrs}$ *IND-CPA sicher. Sei* $(NIWI.Prove, NIWI.Verify)$ *ein NIWI-Beweisverfahren. Dann gilt* $protoRO^{\mathcal{F}_{smt},\mathcal{F}_{kex},\mathcal{G}_{att},\mathcal{G}_{acrs},\mathcal{F}_{RO}} \geq \mathcal{F}_{PSI}^{weak}$*, wenn der Angreifer* $\mathcal{A}$ *keinen korrumpiert.*

Beweis. Der Beweis wird durch 4 Hybride H_i dargestellt, wobei H_0 den realen Ablauf abbildet und H_3 den idealen. Es wird gezeigt, dass die Umgebung die Hybride H_i und H_{i+1} nicht von einander unterscheiden kann. Sei out_i die Ausgabe der Umgebung bei Interaktion mit Hybrid H_i.

Hybrid H_1: Sei H_1 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_1$ und Simulator $Sim_1^{protoRO}$, wobei $\mathcal{F}_1$ und $Sim_1^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_1$ ist eine ideale Funktionalität, die die Eingaben aller Parteien an den Simulator weiterleitet. Wenn der Simulator ein Ergebnis berechnet hat, dann gibt er dieses an die ideale Funktionalität weiter, damit es weiter an die Parteien geleitet wird. $Sim_1^{protoRO}$ simuliert das gesamte Protokoll exakt wie im Realen, mit Verwendung der echten Eingaben und generiert das reale Ergebnis für die Umgebung. Die Ausgaben der Umgebung sind somit identisch verteilt. Daher gilt:

$$Pr[out_0 = 1] - Pr[out_1 = 1] = 0$$

Hybrid H_2: Sei H_2 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_2$ und Simulator $Sim_2^{protoRO}$, wobei $\mathcal{F}_2$ und $Sim_2^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_2$ ist identisch zu $\mathcal{F}_1$. $Sim_2^{protoRO}$ ist identisch zu $Sim_1^{protoRO}$ außer, dass er dem Angreifer nicht mehr die echten Längen der Nachrichten im Protokoll sendet sondern die, im Pseudocode $Sim^{protoRO}$ angegebenen Konstanten. Da die realen Nachrichten aber jeweils konstante Länge und Anzahl haben, erhält der Angreifer exakt dieselben Eingaben. Die Ausgaben der Umgebung sind somit identisch verteilt. Daher gilt:

$$Pr[out_1 = 1] - Pr[out_2 = 1] = 0$$

Hybrid H_3: Sei H_3 das Experiment zwischen der Umgebung, dem idealen Protokoll mit

idealer Funktionalität $\mathcal{F}_3 = \mathcal{F}_{int'}$ und Simulator $\text{Sim}_3^{protoRO} = \text{Sim}^{protoRO}$.

Da in H_2 nur Längen der Nachrichten übermittelt werden, muss der Inhalt der Nachrichten nicht berechnet werden. $\text{Sim}_3^{protoRO}$ braucht nicht mehr die Eingaben S_i um das Protokoll durchzuführen. Es wird nur noch der Zeitpunkt benötigt, zu dem P_i die Eingabe S_i erhält. Diese Information wird aber auch von $\mathcal{F}_{int'}$ durch die Nachrichten "intersect" und "intersect 2" zur Verfügung gestellt. Sollte der Angreifer das Protokoll abbrechen, kann auch $\text{Sim}^{protoRO}$ die Ausführung abbrechen. Die Ausgabe ist nur anders, wenn in den Eingaben S_i eine Kollision auftritt. Da $|S_0 \cup S_1| \leq 2\omega$ und die Wahrscheinlichkeit für eine Kollision zwischen zwei Ausgaben eines Random Oracles kleiner gleich $2^{-l_{hash}}$ ist, gilt:

$$|Pr[out_2 = 1] - Pr[out_3 = 1]| \leq (2\omega)^2 \cdot 2^{-l_{hash}}$$

□

Programm 19 : $\text{Sim}^{protoRO}$ Simulator wenn keine Partei korrumpiert ist

Phase: setup

$\text{Sim}^{protoRO}_{\square}(\bot, \bot)$ Angreifer initialisieren und Nachrichten weiterleiten wie in Programm 18

Phase: first key exchange and input for enclave

receive message "input received" **from** $\mathcal{F}_{PSI}^{weak}$ **do**
  simulate delayed output while leaking ("key exchange is carried out", P_0, P_1) **from** $\mathcal{F}_{kex}$ **to** $\mathcal{A}$
  simulate delayed output while leaking (P_0, T, l_{dh}) **to** $\mathcal{A}$
  simulate delayed output while leaking $(T, P_0, l_{dh} + l_{proof} + l_{eid})$ **to** $\mathcal{A}$
  simulate delayed output while leaking $(P_0, T, e((\omega + 1) \cdot l_{hash}))$ **to** $\mathcal{A}$
  yield the execution **to** *Environment*

Phase: second key exchange and input for enclave

on input *"intersect"* **from** $\mathcal{F}_{PSI}^{weak}$ **do**
  simulate delayed output while leaking "receiver is fetching key" **from** $\mathcal{F}_{kex}$ **to** $\mathcal{A}$
  simulate delayed output while leaking (P_1, T, l_{dh}) **to** $\mathcal{A}$
  simulate delayed output while leaking $(T, P_1, l_{dh} + l_{proof} + l_{eid})$ **to** $\mathcal{A}$
  simulate delayed output while leaking $(P_1, T, e((\omega + 1) \cdot l_{hash}))$ **to** $\mathcal{A}$

Phase: send results

  simulate delayed output while leaking $(T, P_0, (\omega + 1) \cdot l_{hash} + l_{proof})$ **to** $\mathcal{A}$
  return "result to P_0" **to** $\mathcal{F}_{PSI}^{weak}$

on input "intersect 2" **from** $\mathcal{F}_{PSI}^{weak}$ **do**
  simulate delayed output while leaking $(P_1, T, l(\text{"result?"}))$ **to** $\mathcal{A}$
  simulate delayed output while leaking $(T, P_1, (\omega + 1) \cdot l_{hash} + l_{proof})$ to $\mathcal{A}$
  return "result to P_1" **to** $\mathcal{F}_{PSI}^{weak}$

3.6.2 Partei P_0 ist korrumpiert

In Programm 20 ist der Fall beschrieben, dass der Angreifer am Anfang des Ablaufs sich dafür entscheidet Partei P_0 zu korrumpieren.

Satz 3.2. *Sei* (Enc, Dec) *ein IND-CCA sicheres symmetrisches Verschlüsselungsverfahren. Sei das Signaturverfahren von* $\mathcal{G}_{att}$ *EUF-CMA sicher. Sei das Signaturverfahren von* $\mathcal{G}_{acrs}$ *EUF-CMA sicher und das Verschlüsselungsverfahren von* $\mathcal{G}_{acrs}$ *IND-CPA sicher. Sei* $(NIWI.Prove, NIWI.Verify)$ *ein NIWI-Beweisverfahren. Sei* g *ein Generator einer Gruppe* $\mathcal{G}$*, in der die DDH-Annahme gilt. Dann gilt* $protoRO^{\mathcal{F}_{smt},\mathcal{F}_{kex},\mathcal{G}_{att},\mathcal{G}_{acrs},\mathcal{F}_{RO}} \geq \mathcal{F}_{PSI}^{weak}$*, wenn der Angreifer* $\mathcal{A}$ *die Partei* P_0 *korrumpiert.*

Beweis. Der Beweis wird durch 4 Hybride H_i dargestellt, wobei H_0 den realen Ablauf abbildet und H_3 den idealen. Es wird gezeigt, dass die Umgebung die Hybride H_i und H_{i+1} nicht von einander unterscheiden kann. Sei out_i die Ausgabe der Umgebung bei Interaktion mit Hybrid H_i.

Hybrid H_1: Sei H_1 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_1$ und Simulator $\text{Sim}_1^{protoRO}$, wobei $\mathcal{F}_1$ und $\text{Sim}_1^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_1$ ist eine ideale Funktionalität, die die Eingaben aller ehrlichen Parteien an den Simulator weiterleitet. Wenn der Simulator ein Ergebnis berechnet hat, dann gibt er dieses an die ideale Funktionalität weiter, damit es weiter an die Parteien geleitet wird. $\text{Sim}_1^{protoRO}$ simuliert das gesamte Protokoll wie im Realen, mit Verwendung der echten Eingaben und generiert das reale Ergebnis für die Umgebung. Für die Nachrichten zwischen T und P_1 werden nicht mehr ihre echten Längen dem Angreifer gemeldet, sondern die im Pseudocode $\text{Sim}_{P_0}^{protoRO}$ angegebenen Konstanten. $NIWI.Prove(crs, (P_1, \_, \_), \_)$ wird nicht aufgerufen, da die Ausgabe nicht benötigt wird. Der Simulator fordert $idk[P_0]$ durch die korrumpierte Partei P_0 von $\mathcal{G}_{acrs}$ an; Der Simulator ersetzt die Signatur σ der Enklave in $NIWI.Prove(crs, (P_0, \_, \_), \_)$ durch eine zufällige Signatur und verwendet den echten $idk[P_0]$; Da die Signaturen von $\mathcal{G}_{att}$ nicht benötigt werden, kann der Simulator $\Psi_{protoRO}$ selbst ausführen, anstatt die Enklave zu installieren.

Die Umgebung bekommt nicht mit, dass $idk[P_0]$ angefordert wird. Der Zeuge in $NIWI.Prove$ kann geändert werden ohne, dass die Umgebung die Situationen unterscheiden kann, da die von $NIWI.Prove$ ausgegebenen Beweise *witness indistinguishable* sind. Sei also die Wahrscheinlichkeit, dass diese Situationen unterschieden werden können kleiner als die vernachlässigbare Funktion $p_1(\lambda)$. Daher gilt:

$$|Pr[out_0 = 1] - Pr[out_1 = 1]| \leq p_1(\lambda)$$

Hybrid H_2: Sei H_2 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_2$ und Simulator $Sim_2^{protoRO}$, wobei $\mathcal{F}_2$ und $Sim_2^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_2$ nimmt die Eingabe S_0 vom Simulator durch die korrumpierte Partei P_0 entgegen. $Sim_2^{protoRO}$ ist identisch zu $Sim_1^{protoRO}$ außer: Der Simulator extrahiert die Eingabe von P_0 wie in Zeile 24 in Programm 20 angegeben. Um zu extrahieren muss der Simulator vorerst alle Anfragen von P_0 an das Random Oracle speichern. Dann nimmt der Simulator die in s_0' enthaltenen Werte (Hashwerte der Eingabe von P_0) und sucht diese in den gespeicherten Random Oracle Ausgaben um dazu die passende Eingabe zu bestimmen. Falls keine solche Ausgabe gefunden wurde, zählt der Wert als Padding und wird verworfen. Die Wahrscheinlichkeit dass zwei passende Ausgaben gefunden werden ist vernachlässigbar, denn die polynomial-vielen Ausgaben wurden vom Random Oracle zufällig gezogen. Der Simulator berechnet das Ergebnis r indem er das extrahierte S_0 mit dem von $\mathcal{F}_2$ erhaltenen S_1 schneidet. Er berechnet r' wie ab Zeile 38 in Programm 20.

Wenn der Simulator den Code von einem ehrlichen P_0 ausführt, erzeugt er aus dem extrahierten S_0 ein S_0', welches bis aufs Padding gleich dem S_0' in H_1 ist. Die Umgebung merkt, dass das Padding ausgetauscht wurde nur wenn das vom Simulator gewählte Padding zufällig ein Element aus s_0 enthält, da dadurch der berechnete Schnitt dieses Element enthält. Ob aber ein von der Umgebung gewählter Wert oder ein zufällig gezogener Wert gleich einem davon unabhängig zufällig gezogenen ist, ist statistisch ununterscheidbar.

Die Chance, dass unter den von nicht korrumpierten P_0 oder P_1 versendeten 2ω Hashwerten eine Kollision aufgetreten ist, ist kleiner gleich $(2\omega)^2 \cdot 2^{-l_{hash}}$. Eine korrumpierte Partei (zusammen mit Umgebung und Angreifer) kann polynomiell viele Anfragen an das Random Oracle senden. Sei das Polynom, durch welches die Anzahl der Anfragen beschränkt ist, $p(\lambda)$. Die Wahrscheinlichkeit, dass unter allen Antworten eine Kollision ist, ist kleiner gleich $(p(\lambda) + \omega)^2 \cdot 2^{-l_{hash}}$. Wenn das nicht der Fall ist, liefert das Protokoll die korrekte Ausgabe und der Simulator kann r direkt berechnen. Daher gilt:

$$|Pr[out_1 = 1] - Pr[out_2 = 1]| \leq 4\omega^2 \cdot 2^{-l_{hash}}$$

Hybrid H_3: Sei H_3 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_3 = \mathcal{F}_{int'}$ und Simulator $\text{Sim}_3^{protoRO} = \text{Sim}_{P_0}^{protoRO}$.

Die Informationen aus dem Protokollablauf werden in H_2 nicht mehr benötigt, und das Protokoll muss daher in H_3 nicht mehr simuliert werden. Es wird nur noch der Zeitpunkt benötigt, zu dem P_i die Eingabe S_i erhält. Diese Information wird aber auch von $\mathcal{F}_{int'}$ durch die Nachrichten “intersect” und “intersect 2” zur Verfügung gestellt. Sollte der Angreifer das Protokoll abbrechen, kann auch $\text{Sim}^{protoRO}$ die Ausführung abbrechen. Die Ausgaben der Umgebung sind somit identisch verteilt. Daher gilt:

$$Pr[out_2 = 1] - Pr[out_3 = 1] = 0$$

□

Programm 20 : $\text{Sim}_{P_0}^{protoRO}$ Simulator wenn Partei P_0 korrumpiert ist

Phase: setup

$\text{Sim}_{\square}^{protoRO}(P_0, P_0)$ Angreifer initialisieren und Nachrichten weiterleiten wie in Programm 18

on $corr_{out}$(“new_key”, P_1 **from** P_0 **to** $\mathcal{F}_{kex}$) **from** $\mathcal{A}$ **do**

  simulate delayed output while leaking (“key exchange is carried out”, P_0, P_1) **from** $\mathcal{F}_{kex}$ **to** $\mathcal{A}$

  draw k **uniformly at random**

  pass $corr_{in}(k$ **from** $\mathcal{F}_{kex}$ **to** $P_0)$ **to** $\mathcal{A}$

Phase: first key exchange

on $corr_{out}(m_0'$ **from** P_0 **to** $T)$ **from** $\mathcal{A}$ **do**

  send $corr_{out}$(“crs” **from** P_0 **to** $\mathcal{G}_{acrs}$)

receive message $corr_{in}((epk, \_, crs)$ **from** $\mathcal{G}_{acrs}$ **to** $P_0)$ **do**

  send $corr_{out}$(“idk” **from** P_0 **to** $\mathcal{G}_{acrs}$)

receive message $corr_{in}(idk[P_0]$ **from** $\mathcal{G}_{acrs}$ **to** $P_0)$ **do**

  draw eid **uniformly at random**

  draw $a \in \{0, \ldots, |\mathcal{G}| - 1\}$ **uniformly at random**

  $m_0 := g^a$

  draw σ_0, ξ **uniformly at random**

  $C_0 := Enc_{epk}((\sigma_0, idk[P_0]), \xi)$

  $\pi_0 := NIWI.Prove(crs, (P_0, (eid, \Psi_{protoRO}, (m_0', m_0)), C_0), (\xi, \sigma_0, idk[P_0]))$

  pass $corr_{in}((m_0, C_0, \pi_0, eid)$ **from** T **to** $P_0)$ **to** $\mathcal{A}$

Phase: extract input from P_0

on $corr_{out}(S_0'$ **from** P_0 **to** $T)$ **from** $\mathcal{A}$ **do**

  $k_0 := (m_0')^a$

  $s_0 := Dec_{k_0}(S_0')$

  $s_0' :=$ drop first element of s_0

  $S_0 := \{x | h(x \| k) \in s_0'\}$ by looking every element of s_0' in the List of Random Oracle queries

  send $corr_{out}(S_0$ **from** P_0 **to** $\mathcal{F}_{PSI}^{weak})$ **to** P_0

on subroutine output “input received” **from** $\mathcal{F}_{PSI}^{weak}$ **do**

  yield the execution **to** *Environment*

Programm 20 : Fortsetzung von $\text{Sim}_{P_0}^{protoRO}$

Phase: simulate network traffic of P_1

on input *"intersect"* **from** $\mathcal{F}_{PSI}^{weak}$ **do**
  simulate delayed output while leaking "receiver is fetching key" **from** $\mathcal{F}_{kex}$ **to** $\mathcal{A}$
  simulate delayed output while leaking (P_1, T, l_{dh}) **to** $\mathcal{A}$
  simulate delayed output while leaking $(T, P_1, l_{dh} + l_{proof} + l_{eid})$ **to** $\mathcal{A}$
  simulate delayed output while leaking $(P_1, T, e((\omega + 1) \cdot l_{hash}))$ **to** $\mathcal{A}$
  if *first element of* $s_0 \neq h(\perp \| k)$ **then**
    abort
  return "result to P_0" **to** $\mathcal{F}_{PSI}^{weak}$

Phase: simulate result

receive message $corr_{in}(r$ **from** $\mathcal{F}_{PSI}^{weak}$ **to** $P_0)$ **do**
  $r' := \{h(\gamma \| k) | \gamma \in r\}$
  pad r' with random elements
  sort r'
  add $h(\perp || k)$ to r' as first element
  draw σ_r, ξ **uniformly at random**
  $C_{r0} := Enc_{epk}((\sigma_r, idk[P_0]), \xi)$
  $\pi_{r0} := NIWI.Prove(crs, (P_0, (eid, \Psi_{protoRO}, r'), C_{r0}), (\xi, \sigma_r, idk[P_0]))$
  pass $corr_{in}((r', C_{r0}, \pi_{r0})$ **from** T **to** $P_0)$ **to** $\mathcal{A}$

on input *"intersect 2"* **from** $\mathcal{F}_{PSI}^{weak}$ **do**
  simulate delayed output while leaking $(P_1, T, l(\text{"result?"}))$ **to** $\mathcal{A}$
  simulate delayed output while leaking $(T, P_1, (\omega + 1) \cdot l_{hash} + l_{proof})$ to $\mathcal{A}$
  return "result to P_1" **to** $\mathcal{F}_{PSI}^{weak}$

3.6.3 Partei T ist mit oder ohne seine Enklave korrumpiert

In Programm 21 ist der Fall beschrieben, dass der Angreifer am Anfang des Ablaufs sich dafür entscheidet Partei T zu korrumpieren. Der Angreifer kann Partei T mit oder ohne Enklave korrumpieren. type gibt dabei an für welche Korruptionsart der Angreifer sich entscheidet. Bei T_{enclave} wird davon ausgegangen, dass die Enklave für T transparent ist. T (und damit auch der Angreifer) lernt alle Zufallsbits der Enklave und kann damit den Zustand der Enklave bestimmen. Dies modelliert den Fall, dass der Prozessor auf dem die Enklave läuft korrupt ist. Bei T_{only} wird davon ausgegangen, dass die Enklave Informationen vor einem korrupten T geheim halten kann.

Satz 3.3. *Sei* (Enc, Dec) *ein IND-CCA sicheres symmetrisches Verschlüsselungsverfahren. Sei das Signaturverfahren von* $\mathcal{G}_{att}$ *EUF-CMA sicher. Sei das Signaturverfahren von* $\mathcal{G}_{acrs}$ *EUF-CMA sicher und das Verschlüsselungsverfahren von* $\mathcal{G}_{acrs}$ *IND-CPA sicher. Sei* $(NIWI.Prove, NIWI.Verify)$ *ein NIWI-Beweisverfahren. Sei* g *ein Generator einer Gruppe* $\mathcal{G}$*, in der die DDH-Annahme gilt. Dann gilt* $protoRO^{\mathcal{F}_{smt},\mathcal{F}_{kex},\mathcal{G}_{att},\mathcal{G}_{acrs},\mathcal{F}_{RO}} \geq \mathcal{F}_{PSI}^{weak}$*, wenn der Angreifer* $\mathcal{A}$ *die Partei* T *korrumpiert, aber die Enklave ehrlich bleibt.*

Beweis. Damit in Realen eine der Parteien P_i das Protokoll erfolgreich abschließt, muss T eine Enklave mit dem Programm $\Psi_{protoRO}$ installieren, da die Parteien P_i gültige NIWI-Beweis erwarten, den T nur erzeugen kann indem er eine gültige Signatur der Enklave verwendet. T muss das Ergebnis r' unverändert an P_i übermittelt, da T für ein anderes Ergebnis keinen passenden NIWI-Beweis erstellen kann. Damit die Enklave ein Ergebnis berechnet muss das erste Element der beiden Eingaben gleich sein. P_i prüft das erste Element von dem Ergebnis. Der Angreifer kann andere Eingaben in die Enklave geben, wenn er das erste Element der Eingabe von P_i rät. Die Wahrscheinlichkeit ein korrektes erstes Element zu raten ist $\frac{1}{2^{l_{hash}}}$. T lernt das korrekte erste Element wenn er das Protokoll korrekt bis zur Ausgabe von r' ausführt. P_i hat aber schon die *eid* der Enklave gelernt und wird kein Ergebnis einer anderen Enklave akzeptieren.

Der Beweis wird durch 9 Hybride H_i dargestellt, wobei H_0 den realen Ablauf abbildet und H_8 den idealen. Es wird gezeigt, dass die Umgebung die Hybride H_i und H_{i+1} nicht von einander unterscheiden kann. Sei out_i die Ausgabe der Umgebung bei Interaktion mit Hybrid H_i.

Hybrid H_1: Sei H_1 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_1$ und Simulator $\mathrm{Sim}_1^{protoRO}$, wobei $\mathcal{F}_1$ und $\mathrm{Sim}_1^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_1$ ist eine ideale Funktionalität, die die Eingaben aller Parteien an den Simulator weiterleitet. Wenn der Simulator ein Ergebnis berechnet hat, dann gibt er dieses an die ideale Funktionalität weiter, damit es weiter an die Parteien geleitet wird. $\mathrm{Sim}_1^{protoRO}$ simuliert das gesamte Protokoll

exakt wie im Realen, mit Verwendung der echten Eingaben und generiert das reale Ergebnis für die Umgebung. Die Ausgaben der Umgebung sind somit identisch verteilt. Daher gilt:

$$Pr[out_0 = 1] - Pr[out_1 = 1] = 0$$

Hybrid H_2: Sei H_2 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_2$ und Simulator $\text{Sim}_2^{protoRO}$, wobei $\mathcal{F}_2$ und $\text{Sim}_2^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_2$ ist identisch zu $\mathcal{F}_1$. $\text{Sim}_2^{protoRO}$ prüft wie $\text{Sim}_1^{protoRO}$ den NIWI-Beweis für die Nachricht m_i, die ein korrumpiertes T sendet, und bricht ab falls dieser nicht stimmt. Anders als $\text{Sim}_1^{protoRO}$ gibt $\text{Sim}_2^{protoRO}$ die von der Enklave erzeugte Nachricht $m_{i,enk}$ an das simulierte P_i und verwirft das von T gesendete m_i.

Wir betrachten mehrere Ereignisse:

$E_{\text{correctly}}$ Die vom korrumpierten T gesendete Nachricht m_i stammt aus einer Enklave. In diesem Fall ist $m_i = m_{i,enk}$ und $\text{Sim}_2^{protoRO}$ rechnet genau so wie $\text{Sim}_1^{protoRO}$ weiter.

$E_{\text{wrongNIWI}}$ $E_{\text{correctly}}$ tritt nicht ein und der mit m_i gesendete NIWI-Beweis ist nicht korrekt. Hier wird sowohl in H_1 als auch in H_2 abgebrochen, da in beiden der NIWI-Beweis geprüft wird. Der der Angreifer kann also H_1 von H_2 nicht unterscheiden.

E_{idk} $E_{\text{correctly}}$ und $E_{\text{wrongNIWI}}$ treten nicht ein und der in C_i enthaltene $idk[P_i]$ ist korrekt. Aus einem Angreifer $\mathcal{A}$, bei dem E_{idk} mit nicht vernachlässigbarer Wahrscheinlichkeit eintritt, kann man einen Angreifer $\mathcal{B}$ auf die EUF-CMA-Eigenschaft konstruieren. $\mathcal{B}$ simuliert dabei $\mathcal{G}_{acrs}$. $\mathcal{B}$ erhält von dem EUF-CMA-Challenger den vk und verwendet diesen in $\mathcal{G}_{acrs}$ ohne ein eigenes Schlüsselpaar zu ziehen. Um die für $idk[T]$ benötigte Signatur (muss $\mathcal{G}_{acrs}$ ausgeben können, weil T korrumpiert ist) zu erzeugen wird der EUF-CMA-Challenger angefragt. $\mathcal{A}$ berechnet einen gültigen NIWI-Beweis, $\mathcal{B}$ entschlüsselt das Chiffrat C_i, welches mit dem Beweis mitgeschickt wird, mit dem esk, den er als $\mathcal{G}_{acrs}$ gezogen hat und erhält damit $(\sigma_i, idk[P_i])$. $idk[P_i]$ kann als gebrochene Signatur auf P_i dem Challenger gemeldet werden, da P_i nicht korrumpiert ist. Damit ist die EUF-CMA-Eigenschaft gebrochen.

E_{sign} $E_{\text{correctly}}$ und $E_{\text{wrongNIWI}}$ treten nicht ein und die in C_i enthaltene Enklavensignatur ist korrekt. Aus einem Angreifer $\mathcal{A}$, bei dem E_{sign} mit nicht vernachlässigbarer Wahrscheinlichkeit eintritt, kann man einen Angreifer $\mathcal{B}$ auf die EUF-CMA-Eigenschaft konstruieren. $\mathcal{B}$ simuliert unter anderem $\mathcal{G}_{att}$ und bettet darin vk vom Challenger ein, sodass dieser als mpk bei "getpk" zurückgegeben wird. Immer wenn für $\mathcal{G}_{att}$ eine Signatur erstellt werden müsste, wird diese vom Challenger angefragt. Wenn (wie in der obigen Reduktion) aus

$\mathcal{A}$ eine gültige Signatur σ_i extrahiert wird, die keine Enklavenausgabe bestätigt, ist diese Ausgabe gemeinsam mit σ_i eine gefälschte Signatur, die dem EUF-CMA-Challenger gemeldet werden kann.

$E_{\text{NIWIbroken}}$ $E_{\text{correctly}}$ und $E_{\text{wrongNIWI}}$ treten nicht ein und in C_i ist weder $idk[P_i]$ noch σ_i korrekt. In diesem Fall wäre das NIWI-Beweissystem nicht *sound*, denn hier ist das Tupel $(eid, C_i, P_i, (m_i', m_i))$ nicht in der Sprache des Beweissystems.

Die Ereignisse decken den Ereignisraum vollständig ab. In allen Fällen unterscheidet der Angreifer H_1 von H_2 nur mit höchstens vernachlässigbarer Wahrscheinlichkeit.

Hybrid H_3: Sei H_3 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_3$ und Simulator $\text{Sim}_3^{protoRO}$, wobei $\mathcal{F}_3$ und $\text{Sim}_3^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_3$ ist identisch zu $\mathcal{F}_2$. $\text{Sim}_3^{protoRO}$ ist identisch zu $\text{Sim}_2^{protoRO}$ außer, dass $\text{Sim}_3^{protoRO}$ das Experiment abbricht und *proof-failure* ausgibt, wenn folgende Bedingungen gemeinsam zutreffen:

1. P_i hat die Nachricht (m_i, C_i, π_i, eid) empfangen
2. T hat an die Enklave eid die Eingabe m_i' nicht korrekt übermittelt
3. P_i hat $NIWI.Verify(P_i, (mpk, crs), (eid, \Psi_{protoRO}, (m_i', m_i)), C_i, \pi_i)$ ausgeführt und es hat 1 ausgegeben

proof-failure wird nur mit vernachlässigbarer Wahrscheinlichkeit ausgegeben. Da T an die Enklave eid die Eingabe m_i' nicht korrekt übermittelt, bekommt T keine Signatur für $eid, \Psi_{protoRO}, m_i'$ von $\mathcal{G}_{att}$. Die Reduktion funktioniert analog zu der in H_2, E_{idk}, E_{sign}, $E_{\text{NIWIbroken}}$ beschriebenen.

Weil *proof-failure* nur mit vernachlässigbarer Wahrscheinlichkeit ausgegeben wird, ist H_3 von H_2 ununterscheidbar.

Hybrid H_4: Sei H_4 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_4$ und Simulator $\text{Sim}_4^{protoRO}$, wobei $\mathcal{F}_4$ und $\text{Sim}_4^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_4$ ist identisch zu $\mathcal{F}_3$. $\text{Sim}_4^{protoRO}$ ist identisch zu $\text{Sim}_3^{protoRO}$ außer, dass $k_i = k_i'$ nicht wie im Protokoll angegeben berechnet werden, sondern zufällig (als g^c) gezogen werden.

Könnte H_4 von H_3 unterschieden werden, müsste ein Unterscheider existieren. Dieser Unterscheider muss m_0' und m_1' unverändert an die Enklave übergeben.

Wenn m_0' und m_1' unverändert übermittelt werden, würde dieser Unterscheider die DDH-Annahme brechen, denn gegeben ein Tupel (g^a, g^b, g^c), kann dieses Tupel in den Protokollablauf eingebettet werden. Zunächst muss dafür zufällig gewählt werden, ob der Schlüssel k_0 oder k_1 angegriffen werden soll. Dann wird $m_i := g^a$ und $m_i' := g^b$ gewählt. Für $k_i = k_i'$ wird g^c

gewählt. Gilt nun $c = a \cdot b$ liegt das Spiel wie in H_3 vor. Ist c zufällig gezogen liegt das Spiel wie in H_4 vor. Damit gilt unter der DDH-Annahme:

$$|Pr[\mathcal{A}(g^a, g^b, g^{ab}) = 1] - Pr[\mathcal{A}(g^a, g^b, g^c) = 1]| \leq p_4(\lambda)$$

Im Protokoll wird der Schlüsselaustausch 2 mal durchgeführt (zwischen P_0 und der Enklave und P_1 und der Enklave). Bei einer Reduktion auf die DDH-Annahme kann die DDH-Challenge nur in einen dieser Fälle eingebaut werden. Die DDH-Annahme wird nacheinander auf die beiden Fälle angewendet. Jede dieser Änderungen für sich betrachtet kann der Angreifer mit höchstens der Wahrscheinlichkeit $p_4(\lambda)$ erkennen. Damit ist die Wahrscheinlichkeit, dass der Angreifer H_4 von H_3 unterscheiden kann beschränkt durch:

$$|Pr[out_4 = 1] - Pr[out_3 = 1]| \leq 2 \cdot p_4(\lambda)$$

Hybrid H_5: Sei H_5 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_5$ und Simulator $\text{Sim}_5^{protoRO}$, wobei $\mathcal{F}_5$ und $\text{Sim}_5^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_5$ ist identisch zu $\mathcal{F}_4$. $\text{Sim}_5^{protoRO}$ ist identisch zu $\text{Sim}_4^{protoRO}$ außer, dass anstatt S'_i, $Enc_k(s_i)$ für zufällig gezogene s_i wie in Zeile 14 in Programm 21 gesendet wird. Die Enklave rechnet aber mit den korrekten Werten weiter. Wenn der Angreifer ein anderes Chiffrat als S'_i an die Enklave senden will, entschlüsselt der Simulator dieses. Wenn die Entschlüsselung fehlschlägt, verhält sich der Simulator als hätte die Enklave abbgebrochen. Für S'_1 wird zusätzlich abgebrochen wenn die ersten Elemente von $Dec_k(S'_i)$ nicht übereinstimmen.

Könnte dies ein Angreifer unterscheiden wäre die Verschlüsselung nicht IND-CCA sicher: Im IND-CCA Experiment wählt man m_0 als die simulierte Nachricht, von der S'_i eine Verschlüsselung ist, und m_1 zufällig (wie in Zeile 14) und sendet das erhaltene Challenge-Chiffrat als S'_i. Wenn der Angreifer der Enklave ein $\tilde{S}'_i \neq S'_i$ geben will, wird dieses mit dem vom Challenger bereitgestellten Entschlüsselungsorakel entschlüsselt und mit dem Ergebnis wie in H_5 beschrieben verfahren. Wenn der IND-CCA-Challenger $b = 0$ wählt und m_0 verschlüsselt sieht der Angreifer einen Ablauf genau wie in H_4. Wählt der Challenger $b = 1$ ist der Ablauf wie in H_5. Das ausgegebene Choice-Bit des Unterscheiders kann man also direkt als Ausgabe des IND-CCA Angreifers ausgeben. Unter der Annahme dass alle IND-CCA Angreifer einen Vorteil kleiner als $p_5(\lambda)$ haben gilt, da die Annahme zweimal angewendet werden muss (für S'_0 und S'_1):

$$|Pr[out_5 = 1] - Pr[out_4 = 1]| \leq 2 \cdot p_5(\lambda)$$

Hybrid H_6: Sei H_6 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_6$ und Simulator $Sim_6^{protoRO}$, wobei $\mathcal{F}_6$ und $Sim_6^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_6$ ist identisch zu $\mathcal{F}_5$ außer, dass die Ausgabe von P_i nicht mehr gesetzt werden darf sondern in $\mathcal{F}_{PSI}^{weak}$ direkt berechnet wird. Außerdem gibt $\mathcal{F}_6$ S_i nicht an $Sim_6^{protoRO}$ weiter. Damit ist $\mathcal{F}_6 = \mathcal{F}_{int'}$. $Sim_6^{protoRO}$ ist identisch zu $Sim_5^{protoRO}$ außer, dass die Ergebnisse für P_i nicht mehr an $\mathcal{F}_6$ gesendet werden.

T kann das Ergebnis r', welches durch die Enklave eid berechnet wurde, nicht fälschen (Reduktion auf die soundness-Eigenschaft des NIWI-Beweissystems wie in H_2, $E_{\text{NIWIbroken}}$, Reduktion auf die EUF-CMA-Eigenschaft von den im $\mathcal{G}_{acrs}$ verwendeten Signaturverfahren wie in H_2, E_{idk}, Reduktion auf die EUF-CMA-Eigenschaft von den im $\mathcal{G}_{att}$ verwendeten Signaturverfahren wie in H_2, E_{sign}). Sei die Wahrscheinlichkeit, dass der Angreifer das Ergebnis r' fälschen kann $p_6'(\lambda)$. Diese Funktion ist vernachlässigbar, was durch die Reduktion gezeigt wird.

Das Enklavenergebnis ist korrekt, da die Enklave direkt mit den (gehashten) Eingaben von P_i rechnet. Daher macht es keinen Unterschied ob das korrekte Ergebnis von der Enklave oder direkt von der idealen Funktionalität berechnet wird. Die Wahrscheinlichkeit, dass eine Kollision in Hyb_5 aufgetreten war und jetzt nicht mehr auftritt, da die ideale Funktionalität nicht auf Hashwerten rechnet. Die Umgebung und der Angreifer können polynomiell viele Anfragen an das Random Oracle senden. Sei das Polynom, durch welches die Anzahl der Anfragen beschränkt ist, $p(\lambda)$. Die Wahrscheinlichkeit, dass unter allen Antworten eine Kollision ist, ist kleiner gleich $(p(\lambda) + 2\omega)^2 \cdot 2^{-l_{hash}} = p_6''(\lambda)$. Diese Wahrscheinlichkeit ist vernachlässigbar.

Die Experimente sind also nicht mit mehr als vernachlässigbarer Wahrscheinlichkeit $p_6(\lambda) = p_6'(\lambda) + p_6''(\lambda)$ unterscheidbar

$$|Pr[out_6 = 1] - Pr[out_5 = 1]| \leq p_6(\lambda)$$

Hybrid H_7 : Sei H_7 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_7$ und Simulator $Sim_7^{protoRO}$, wobei $\mathcal{F}_7$ und $Sim_7^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_7 = \mathcal{F}_{int'}$. $Sim_7^{protoRO}$ ist identisch zu $Sim_6^{protoRO}$ außer, dass die Enklave mit den (ersetzen) Eingabe die von (den von Sim emulieren) P_i kommen rechnet.

Der Schlüssel k ist zufällig gewählt und vor T verborgen. Die Anzahl der Random Oracle Anfragen des Angreifers sei durch das Polynom $p(x)$ beschränkt. Wir betrachten pro Anfrage folgende Ereignisse:

E_{hitKey} Die Anfrage endet auf den vom Simulator gezogenen zufälligen Key $k \in \{0,1\}^{l_{kex}}$.

E_{wrongKey} Die Anfrage endet nicht auf $k \in \{0,1\}^{l_{kex}}$.

In Ereignis E_{wrongKey} verhalten sich H_6 und H_7 genau gleich.

Ereignis E_{hitKey} kann nur eintreten, wenn der Angreifer k für die Anfrage verwendet. Die Wahrscheinlichkeit, dass der Angreifer den Schlüssel für die erste Anfrage rät ist $Pr[E_{\text{hitKey}}] \leq 2^{-l_{kex}}$. Mit dieser Wahrscheinlichkeit kann er H_6 und H_7 anhand der ersten Anfrage unterscheiden. So können nacheinander die $p(x)$ Orakelanfragen bearbeitet werden. Damit ist die Gesamtwahrscheinlichkeit, dass die Experimente unterscheidbar sind:

$$|Pr[out_6 = 1] - Pr[out_7 = 1]| \leq p(\lambda) \cdot 2^{-l_{kex}}$$

Daher sehen für T alle Elemente in r' ununterscheidbar von Zufall aus.

Hybrid H_8: Sei H_8 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_8$ und Simulator $\text{Sim}_8^{protoRO}$, wobei $\mathcal{F}_8$ und $\text{Sim}_8^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_8 = \mathcal{F}_{int'}$. $\text{Sim}_8^{protoRO}$ ist identisch zu $\text{Sim}_7^{protoRO}$ außer, dass die Enklave wieder mit dem echten, aus dem Diffie-Hellman-Schlüsselaustausch stammenden Schlüssel arbeitet. Dadurch führt die Enklave wieder das Programm $\Psi_{protoRO}$ aus und $\mathcal{G}_{att}$ arbeitet korrekt. Damit ist $\text{Sim}_8^{protoRO} = \text{Sim}_{T_{\text{only}}}^{protoRO}$.

Der Beweis ist Analog zu dem für H_4:

$$|Pr[out_8 = 1] - Pr[out_7 = 1]| \leq 2 \cdot p_4(\lambda)$$

□

Satz 3.4. *Es gilt* $protoRO^{\mathcal{F}_{smt},\mathcal{F}_{kex},\mathcal{G}_{att},\mathcal{G}_{acrs},\mathcal{F}_{RO}} \geq \mathcal{F}_{PSI}^{weak}$, *wenn der Angreifer* $\mathcal{A}$ *die Partei T korrumpiert und die Enklave belauscht.*

Beweis. Der Beweis wird durch 5 Hybride H_i dargestellt, wobei H_0 den realen Ablauf abbildet und H_4 den idealen. Es wird gezeigt, dass die Umgebung die Hybride H_i und H_{i+1} nicht von einander unterscheiden kann. Sei out_i die Ausgabe der Umgebung bei Interaktion mit Hybrid H_i.

Hybrid H_1: Sei H_1 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_1$ und Simulator $\text{Sim}_1^{protoRO}$, wobei $\mathcal{F}_1$ und $\text{Sim}_1^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_1$ ist eine ideale Funktionalität, die die Eingaben aller Parteien an den Simulator weiterleitet. Wenn der Simulator ein Ergebnis berechnet hat, dann gibt er dieses an die ideale Funktionalität weiter, damit es weiter an die Parteien geleitet wird. $\text{Sim}_1^{protoRO}$ simuliert das gesamte Protokoll exakt wie im Realen, mit Verwendung der echten Eingaben und generiert das reale Ergebnis für die Umgebung. Die Ausgaben der Umgebung sind somit identisch verteilt. Daher gilt:

$$Pr[out_0 = 1] - Pr[out_1 = 1] = 0$$

Hybrid H_2: Sei H_2 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_2$ und Simulator $\text{Sim}_2^{protoRO}$, wobei $\mathcal{F}_2$ und $\text{Sim}_2^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_2 = \mathcal{F}_1$. $\text{Sim}_2^{protoRO}$ schreibt alle Anfragen vom simulierten P_i an das Random Oracle in einer Tabelle pro P_i mit. Im Code zur Ergebnisberechnung vom simulierten P_i führt $\text{Sim}_2^{protoRO}$ keine Random Oracle Anfragen durch, sondern benutzt die erstellte Tabelle. Alle diese Anfragen können aus der Tabelle beantwortet werden, weil genau dieselben Anfragen wie zur Berechnung von s_i gestellt werden und das Random Oracle auf dieselben Anfragen konsistent die gleiche Antwort liefert. Daher gilt:

$$Pr[out_1 = 1] - Pr[out_2 = 1] = 0$$

Hybrid H_3: Sei H_3 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_3$ und Simulator $\text{Sim}_3^{protoRO}$, wobei $\mathcal{F}_3$ und $\text{Sim}_3^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_3 = \mathcal{F}_2$. $\text{Sim}_3^{protoRO}$ verwendet als Eingabe der Parteien nicht mehr S_0 und S_1, sondern berechnet s_i' indem er zufällige Hashwerte zieht und dabei auf die Größe des Schnitts achtet, wie in $\text{Sim}_{T_{\text{enclave}}}^{protoRO}$ angegeben. Der Code zur Berechnung von s_i' aus P_i wird nicht mehr ausgeführt. Die Tabellen für das Beantworten der Random Oracle Anfragen für die Ergebnisberechnung werden wie folgt erzeugt: für $\perp \| k$ wird z eingetragen, für $\{x \| k \mid x \in (S_0 \cap S_1)\}$ werden die Elemente aus $s_0' \cap s_1'$ in zufälliger Reihenfolge eingetragen, für $\{x \| k \mid x \in (S_i \setminus S_{1-i})\}$ werden verschiedene zufällige Elemente aus $s_i' \setminus s_{1-i}'$ eingetragen.

Aus Überlegungen, wie in H_7 bei korrumpiertem T mit ehrlicher Enklave folgt, dass T Elemente in r' (und somit Hybrid H_2 von Hybrid H_3) mit Wahrscheinlichkeit

$$|Pr[out_2 = 1] - Pr[out_3 = 1]| \leq p(\lambda) \cdot 2^{-l_{kex}}$$

von Zufall unterscheiden kann.

Hybrid H_4: Sei H_4 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_4$ und Simulator $\text{Sim}_4^{protoRO}$, wobei $\mathcal{F}_4$ und $\text{Sim}_4^{protoRO}$ wie folgt definiert sind: $\mathcal{F}_4 = \mathcal{F}_{PSI}^{weak}$. $\text{Sim}_4^{protoRO}$ berechnet nicht mehr mit dem vom korrumpierten T erhaltenen r' das Ergebnis der Partei P_i mit ihrem echten Programmcode, sondern berechnet a_i, b_i, wie in $\text{Sim}_{T_{\text{enclave}}}^{protoRO}$ angegeben und lässt die ideale Funktionalität damit aus den entsprechenden Mengen r_i ziehen. Damit braucht $\text{Sim}_4^{protoRO}$ die Eingaben S_i nicht, sodass $\mathcal{F}_4$ sie ihm nicht übermitteln muss.

Elemente in r' bilden für die erste Partei, die einen Schnitt erhält, 3 Teilmengen: $r' \cap s_0' \cap s_1'$, $r' \cap (s_i' \setminus s_{1-i}')$ und Padding. Bei der Auswertung von r' durch den simulierten Code von P_i in H_3 wird für alle Elemente in $x \in s_i'$, die nicht Padding sind, geprüft ob $x \in r'$. Wenn dies der Fall ist, wird das zugehörige Element aus S_i dem Ergebnis r hinzugefügt. Dabei wurde die

Tabelle so angelegt, dass $s'_0 \cap s'_1$ den Elementen in $S_0 \cap S_1$ entsprechen. Die Abbildung zwischen diesen Elementen wird nur für diese Auswertung benötigt. Daher können in H_4 die Elemente auch von $\mathcal{F}_{PSI}^{weak}$ aus den entsprechenden Mengen gezogen werden und der Simulator muss nur die Anzahl übermitteln. Da $\mathcal{F}_{PSI}^{weak}$ das Ergebnis berechnet, erstellt der Simulator keine Tabelle. Beim der zweiten Partei, die einen Schnitt erhält, kommt hinzu, dass der Angreifer weiß, welche Elemente er bereits an die erste Partei gesendet hat. Diese kann er als gesonderte 4. Teilmenge betrachten. Dieses zusätzliche Wissen des Angreifers, wird in der Variable c_i abgebildet.

Die Experimente verhalten sich nur anders, wenn das Random Oracle eine Kollision für die höchstens 2ω Anfragen mit den Elementen der Eingaben ausgibt. Weil das Random Oracle seine ausgaben zufällig zieht, ist diese Wahrscheinlichkeit kleiner gleich $(2\omega)^2 2^{-l_{hash}}$. Damit gilt:

$$|Pr[out_3 = 1] - Pr[out_4 = 1]| \leq (2\omega)^2 2^{-l_{hash}}$$

□

Programm 21 : $\mathrm{Sim}^{protoRO}_{T_{\text{type}}}$, type $\in$ {only, enclave}

Phase: setup

$\mathrm{Sim}^{protoRO}_{\square}(T, T_{\text{type}})$ Angreifer initialisieren und Nachrichten weiterleiten wie in Programm 18
on subroutine output "input received" **from** $\mathcal{F}_{PSI}^{weak}$ **do**
  send $corr_{out}$("crs" **from** T **to** $\mathcal{G}_{acrs}$)

receive message $corr_{in}((epk, \_, crs)$ **from** $\mathcal{G}_{acrs}$ **to** $T)$ **do**
  simulate delayed output while leaking ("key exchange is carried out", P_0, P_1) **from** $\mathcal{F}_{kex}$ **to** $\mathcal{A}$

Phase: first key exchange and input for enclave

  draw $a' \in \{0, \ldots, |\mathcal{G}| - 1\}$ **uniformly at random**
  $m'_0 := g^{a'}$
  pass $corr_{in}(m'_0$ **from** P_0 **to** $T)$ **to** $\mathcal{A}$

on $corr_{out}((m_0, C_0, \pi_0, eid)$ **from** T **to** $P_0)$ **from** $\mathcal{A}$ **do**
  assert $NIWI.Verify(P_0, (mpk, crs), (eid, \Psi_{protoRO}, (m'_0, m_0)), C_0, \pi_0) = 1$
  $k'_0 := m_0^{a'}$
  draw s'_0 as ω random elements of length l_{hash}
  sort s'_0
  draw z as random element of length l_{hash}
  $s_0 :=$ add z as first element to s'_0
  $S'_0 := Enc_{k'_0}(s_0)$
  pass $corr_{in}(S'_0$ **from** P_0 **to** $T)$ **to** $\mathcal{A}$

if *type* = enclave **then**
  receive message $corr_{in}(|r|$ **from** $\mathcal{F}_{PSI}^{weak}$ **to** $T)$ **do**
    send $corr_{out}$("size received" **from** T **to** $\mathcal{F}_{PSI}^{weak}$)
end

Phase: second key exchange and input for enclave

on input *"intersect"* **from** $\mathcal{F}_{PSI}^{weak}$ **do**
  simulate delayed output while leaking "receiver is fetching key" **from** $\mathcal{F}_{kex}$ **to** $\mathcal{A}$
  draw $b' \in \{0, \ldots, |\mathcal{G}| - 1\}$ **uniformly at random**
  $m'_1 := g^{b'}$
  pass $corr_{in}(m'_1$ **from** P_1 **to** $T)$ **to** $\mathcal{A}$

Programm 21 : Fortsetzung von $\text{Sim}^{protoRO}_{T_{\text{type}}}$

on $corr_{out}((m_1, C_1, \pi_1, eid)$ **from** T **to** $P_1)$ **from** $\mathcal{A}$ **do**
  assert $NIWI.Verify(P_1, (mpk, crs), (eid, \Psi_{protoRO}, (m'_1, m_1)), C_1, \pi_1) = 1$
  $k'_1 := m_1^{b'}$
  if *type* = enclave **then**
    draw $t := |r|$ distinct elements out of s'_0 **uniformly at random**
    draw $s'_1 :=$ $t \cup$ new elements **uniformly at random** to fill s'_1 up to size ω
  else if *type* = only **then**
    draw s'_1 as ω elements **uniformly at random** of length l_{hash}
  end
  sort s'_1
  $s_1 :=$ add z as first element of s'_1
  $S'_1 := Enc_{k'_1}(s_1)$
  pass $corr_{in}(S'_1$ **from** P_1 **to** $T)$ **to** $\mathcal{A}$

Phase: calculate and send result

on $corr_{out}((r'_0, C_{r0}, \pi_{r0})$ **from** T **to** $P_0)$ **from** $\mathcal{A}$ **do**
  assert $NIWI.Verify(P_0, (mpk, crs), (eid, \Psi_{protoRO}, r'_0), C_{r0}, \pi_{r0}) = 1$
  $a_0 := |r'_0 \cap (s'_0 \cap s'_1)|$
  $b_0 := |r'_0 \cap s'_0| - a_0$
  return "result to P_0", (a_0, b_0) **to** $\mathcal{F}^{weak}_{PSI}$

on input *"intersect 2"* **from** $\mathcal{F}^{weak}_{PSI}$ **do**
  pass $corr_{in}$("result?" **from** P_1 **to** T)

on $corr_{out}((r'_1, C_{r1}, \pi_{r1})$ **from** T **to** $P_1)$ **from** $\mathcal{A}$ **do**
  assert $NIWI.Verify(P_1, (mpk, crs), (eid, \Psi_{protoRO}, r'_1), C_{r1}, \pi_{r1}) = 1$
  $a_1 := |(r'_0 \cap r'_1) \cap (s'_0 \cap s'_1)|$
  $c_1 := |r'_1 \cap (s'_0 \cap s'_1)| - a_1$
  $b_1 := |r'_1 \cap s'_1| - a_1 - c_1$
  return "result to P_1", (a_1, b_1, c_1) **to** $\mathcal{F}^{weak}_{PSI}$

3.7 Alternative Lösung für das Problem der nicht-Abstreitbarkeit

Da die NIWI-Beweise nicht praktikabel für reale Protokolle sind, lohnt es sich alternative Beweisstrategien zu betrachten, die dieses Problem anders behandeln.

Eine Variante wäre es zu fordern, dass P_i auch Enklaven anlegen darf, dies aber im normalen Protokollablauf nicht tun muss. Dieser Ansatz findet sich damit ab, dass das Protokoll nur dann ununterscheidbar von seiner idealen Funktionalität ist, wenn P_i einen Attested Execution Processor hat und sich damit selbst Enklavensignaturen erzeugen kann. Da die Enklaven anonym attestieren, sind die dadurch erzeugten Enklavensignaturen gleichwertig mit denen, die T erzeugt. Diese Idee entspricht dem Verständnis von globalen ideale Funktionalitäten wie es in [PST17] beschrieben ist.

Eine andere Variante ist es dem Simulator zu erlauben auch im Namen ehrlicher Parteien globale Funktionalitäten aufzurufen. Das entspricht gängigen Beweismustern für globale ideale Funktionalitäten, wie sie im "generalized UC-Modell" benutzt werden.

In beiden Fällen muss aber trotzdem eine Hintertür eingebaut werden, die erlaubt, die Eingabe S_i von korrumpierten P_i zu extrahieren. P_i sendet seine Eingabe nur verschlüsselt mit einem Schlüssel, der zuvor mit einer Enklave ausgetauscht wurde. Eine Möglichkeit für eine solche Hintertür wäre den CRS direkt in $\Psi_{protoRO}$ hardcoded zu integrieren und dann $\Psi_{protoRO}$ bei Eingabe von $idk[P_i]$ S_i' ausgeben zu lassen. Alternativ könnte man auch definieren, dass bei korrumpierten P_i die Enklaven für den Simulator transparent sind.

Der Sicherheits-Beweis ist großteils unverändert. Anstatt der NIWI-Beweise werden direkt Signaturen ausgetauscht und validiert. Wenn im Simulator für korrumpiertes P_i ein NIWI-Beweis mithilfe eines Identity Keys erzeugt werden soll, muss der Simulator für das (ehrliche) T das Programm $\Psi_{protoRO}$ als Enklave installieren, dann die Interaktion mit der Enklave wie ein ehrliches T weiterführen um schließlich mithilfe der Hintertür S_i' zu extrahieren. In dem Fall, dass P_i korrumpiert sind folgende Änderungen an dem Beweis vorzunehmen. Hybrid H_4 fällt weg. In Hybrid H_5 erzeugt der Simulator eine Enklave im Namen von T, bedient sie ein ehrliches T es tun würde und nutzt die Hintertür um an k_i zu gelangen. In dem Fall, dass T korrumpiert ist, werden die NIWI-Beweise durch Signaturen ausgetauscht. So wie zuvor die NIWI-Beweise für diesen Fall nicht fälschbar waren sind jetzt auch die Signaturen der Enklave nicht fälschbar.

4 Verfahren *protoHASH* mit Einsatz mit kollisionsresistenter Hashfunktion

Da es unsichere Protokolle gibt, die im Random Oracle Modell beweisbar sicher sind, lohnt es sich Möglichkeiten zu betrachten den Schnitt zu berechnen, wenn statt eines Random Oracles eine kollisionsresistente Hashfunktion verwendet wird. Eine der Eigenschaften vom Random Oracle war, dass der Simulator die Anfragen durchsuchen kann und dadurch die Eingabe von P_i extrahieren. Um auch weiterhin extrahieren zu können fordern wir für das in diesem Kapitel vorgestellte Verfahren *protoHASH*, dass die Hashwerte durch eine Enklave berechnet werden. Es ist weiterhin ein Ziel des Protokolls, dass nur Hashwerte der Eingaben über das Netzwerk versendet werden. Das bedeutet, dass jede Partei P_i einen Attested Execution Processor benötigt um eine Enklave zu installieren, die diese Hashwerte berechnet.

Das Verfahren *protoHASH* realisiert die ideale Funktionalität $\mathcal{F}_{PSI}$, die in Programm 22 beschrieben ist, die wir allerdings erst nach dem Protokollablauf beschreiben.

Protokollablauf Wir betrachten den Protokollablauf von *protoHASH*. Für diese Beschreibung gehen wir davon aus, das alle Parteien ehrlich sind. An besonders interessanten Stellen gehen wir auf den Fall ein, dass eine der Parteien korrumpiert ist. Wenn wir von P_i sprechen gelten die Aussagen sowohl für P_0 als auch P_1.

Zu Beginn des Protokolls erhält eine der Parteien P_i die Eingabe S_i von der Umgebung. Die Partei fragt den Schlüssel *mpk* bei $\mathcal{G}_{att}$ an und speichert diesen, um in Zukunft die Signaturen der Enklave zu prüfen. P_0 führt einen Schlüsselaustausch mit P_1 über die ideale Funktionalität $\mathcal{F}_{kex}$ durch. P_i nutzt den sich daraus ergebenden Schlüssel k um die Elemente von S_i und das Padding mit einem deterministischem Algorithmus Enc' zu verschlüsseln und s_i^*. P_i installiert eine eigene Enklave mit Programmcode Ψ_{hash} und erhält eid_i dieser Enklave. P_i beginnt den Schlüsselaustausch mit der Enklave von T, indem P_i eine Nachricht $m_i' := g^{a'}, eid_i$ an den Dienstanbieter T sendet. Der Dienstanbieter T fragt den Schlüssel *mpk* bei $\mathcal{G}_{att}$ an und benutzt diesen um eine Enklave mit Programmcode $\Psi_{protoHASH}[mpk]$ zu installieren. T erhält den eindeutigen Identifier *eid* der Enklave. T führt auf der Enklave *eid resume* mit der Nachricht m_i', eid_i von P_i aus. Die Enklave setzt den Schlüsselaustausch mit P_i fort, indem die Enklave

$m_i := g^a$ generiert, zusammen mit der Nachricht m_i', eid_i von P_i signiert und an T ausgibt. Die Enklave hat zusätzlich die Möglichkeit in der Variable $bits$ geheime Informationen, wie Zufall oder internen Zustand, auszugeben. Dieser Fall tritt nur ein, wenn die Enklave nicht ehrlich ist. Eine ehrliche Enklave befüllt die Variable $bits$ nicht. Der Dienstanbieter T gibt m_i zusammen mit der Signatur der Enklave σ_i und der eid an P_i. T muss m' und eid_i nicht zurück an P_i senden, da diese von P_i stammen. P_i verwendet mpk um die Signatur der Enklave zu überprüfen. Wenn die Signatur nicht korrekt ist, bricht P_i ab und nimmt nicht weiter am Protokoll Teil. Wenn die Signatur korrekt ist, sendet P_0 die eid an P_1.

Sobald P_1 weiß, dass er mit der Enklave eid einen gemeinsamen Schlüssel hat, prüft er, dass P_0 mit der gleichen Enklave verbunden ist, indem er seinen eigenen Wert mit dem von P_0 vergleicht. Wenn alle Parteien ehrlich sind, kann diese Überprüfung nicht fehlschlagen. Sollte T korrumpiert sein schützt diese Überprüfung davor, dass T die Parteien mit verschiedenen Enklaven verbindet (Wenn P_0 oder P_1 korrumpiert sind, dann ist T ehrlich und wird sie nicht mit verschiedenen Enklaven verbinden). Damit P_0 diese Überprüfung auch durchführen kann, sendet P_1 eine Nachricht eid an P_0 (Diese Antwort ist genauso gut wie eine Bestätigung "ok", denn ein unehrliches P_1 kann die von P_0 erhaltene eid zurückschicken).

P_i gibt die bereits verschlüsselte Eingabe s_i^* in seine Enklave eid_i. Diese bildet elementweise Hashwerte, sortiert die Menge. Anschließend wird diese Menge s_0 gemeinsam mit einer Signatur σ_0' der Enklave eid_0 ausgegeben. P_0 berechnet den gemeinsamen Schlüssel k_i als $(m_i)^{a'}$. P_0 verschlüsselt (s_0, σ_0') mit dem Schlüssel k_i mit einem IND-CCA sicherem Verfahren und sendet dieses Chiffrat an T. Es ist wichtig, dass dies erst passiert, nachdem P_i den Wert von eid ausgetauscht haben und damit bestätigen, dass sie mit derselben Enklave verbunden sind. So wird sichergestellt, dass T keine Information über die Eingaben erhält, wenn er P_i nicht mit derselben Enklave verbindet. T übergibt dieses Chiffrat der Enklave eid, welche es entschlüsselt und die darin enthaltene Signatur prüft. P_1 verfährt analog.

Wenn die Enklave von beiden Parteien P_i die Eingabe erhalten hat, berechnet sie auf den Hashwerten den Schnitt r' und gibt diesen aus. T übermittelt ihn gemeinsam mit der zugehörigen Enklavensignatur an beide P_i, welche die Signatur prüfen und danach den echten Schnitt r aus r' ableiten, um ihn an die Umgebung auszugeben.

Dieser gesamte Protokollablauf von *protoHASH* ist in Abbildung 4.1 gezeigt unter der Annahme eines Angreifers, der nicht in den Protokollablauf eingreift. Um den Fluss übersichtlich zu halten, sind Nachrichten zwischen P_0, P_1 und T direkt eingezeichnet und nicht über $\mathcal{F}_{smt}$ geleitet. Außerdem wird der Angreifer weggelassen sowie seine Bestätigungsnachrichten an $\mathcal{F}_{smt}$ und $\mathcal{F}_{kex}$. Zudem fragen P_0, P_1 und T mpk von $\mathcal{G}_{att}$ an, auch diese Nachrichten sind weggelassen worden.

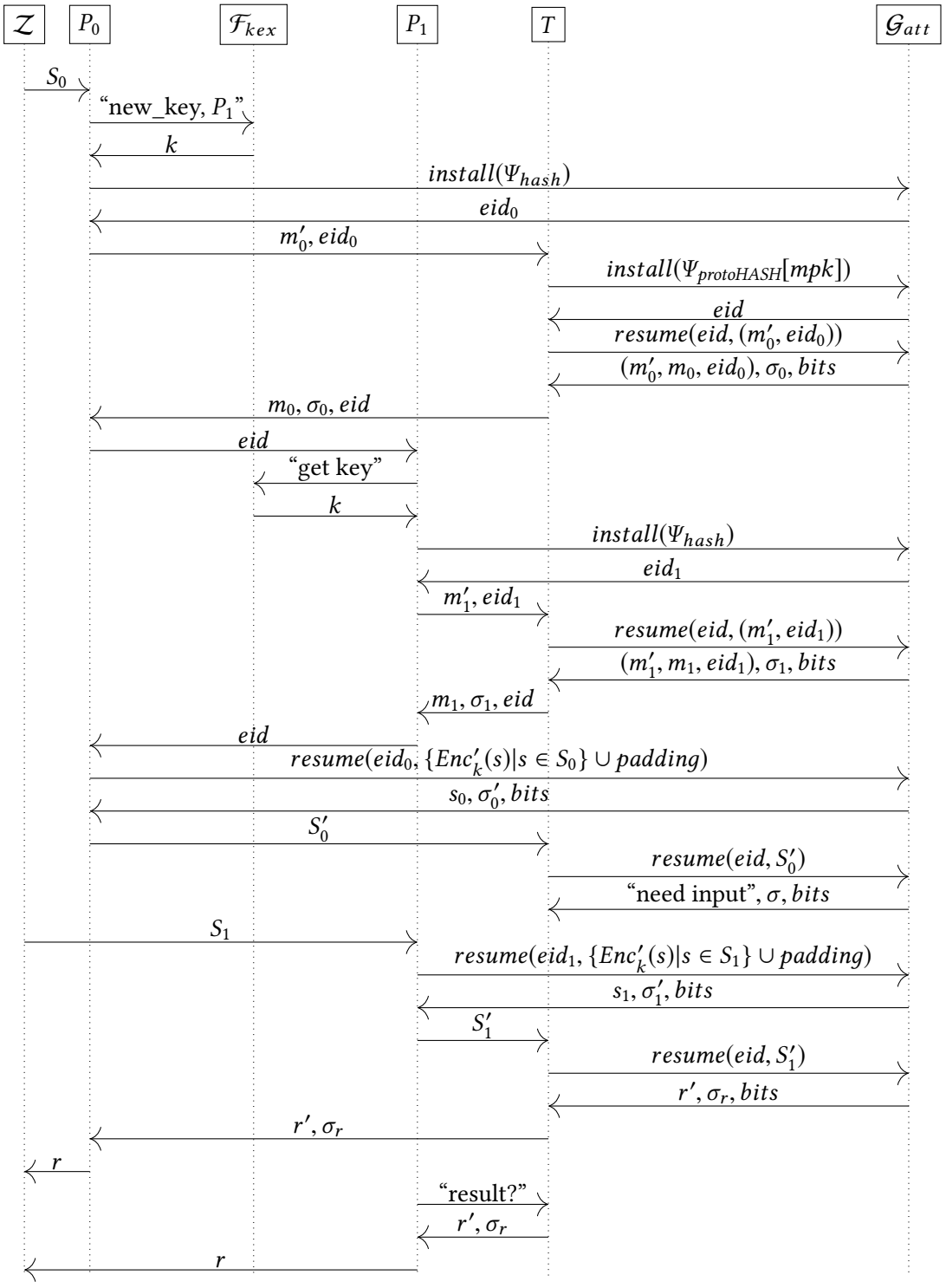

Abbildung 4.1: Protokollablauf von *protoHASH* ohne Störung im Realen

4.1 Zu realisierende ideale Funktionalität für *protoHASH*

Die ideale Funktionalität $\mathcal{F}_{PSI}$ beschreibt das gewünschte Verhalten des Protokolls zur Schnittmengenberechnung und ist in Programm 22 angegeben.

Im Protokoll *protoHASH* kann T mit oder ohne Enklave korrumpiert werden. Nur im Fall, dass T korrumpiert wird und die Enklave Informationen an T verrät, lernt T (und somit der Simulator oder Angreifer) die Größe des Schnitts $|r|$.

Programm 22 : $\mathcal{F}_{PSI}$ Die verbesserte ideale Funktionalität für sicheren gemeinsamen Schnitt

Phase: first input

receive message S_0 **from** P_0 **do**
  store S_0
  send "input received" **to the Adversary**

Phase: second input

receive message S_1 **from** P_1 **do**
  $r \coloneqq S_0 \cap S_1$
  if T *is corrupted and enclave leaks information* **then**
    send $|r|$ **to the Adversary**
    receive message "size received" **from the Adversary**
  end
  send "intersect" **to the Adversary**

Phase: send result

receive message "result to P_i" **from the Adversary do**
  send r **to** P_i

receive message "execution resumed" **from** P_1 **do**
  send "intersect 2" **to the Adversary**

receive message "result to P_{1-i}" **from the Adversary do**
  send r **to** P_{1-i}

Ein Protokoll, dass diese ideale Funktionalität realisiert, muss insbesondere folgende Eigenschaften haben:

- Der Angreifer lernt immer, dass ein Schnitt berechnet wurde, und kann den Protokollablauf stören (für P_0, für P_1 oder für beide).
- Wenn er keine Partei korrumpiert, lernt er nur, dass das Protokoll durchgeführt wurde.
- Wird P_0 oder P_1 korrumpiert, so kann der Angreifer die entsprechende Eingabe ändern und lernt den Schnitt.

- Wird T korrumpiert, aber die Enklave bleibt ehrlich, lernt T nur, dass das Protokoll durchgeführt wurde.
- Wird T korrumpiert und die Enklave gibt ihre Geheimnisse preis, so lernt der Angreifer die Größe des Schnitts $|r|$.

4.2 Programm Ψ_{hash} für Enklave von Partei P_i

In Programm 23 ist der Code von Ψ_{hash} angegeben, welchen die Partei P_i als Enklave installiert um die Hashwerte der bereits verschlüsselten Eingabe und des Paddings zu berechnen. Außerdem sortiert die Enklave die Hashwerte, bevor sie von $\mathcal{G}_{att}$ signiert an die Partei zurückgegeben werden.

Programm 23 : Ψ_{hash} Programm für Enklave von Partei P_i

```
1 receive message s* do
2 |   s := {h(x)|x ∈ s*})
3 |   sort s
4 |   return s
```

4.3 Programm in $\mathcal{G}_{att}$: $\Psi_{\textbf{protoHASH}}$

In Programm 24 ist der Code angegeben, der als Enklave *eid* von Partei T installiert wird um den Schnitt zwischen P_0 und P_1 zu berechnen. Genau wie das Programm 15 für Protokoll *protoRO* wird auch hier der Diffie-Hellman Schlüsselaustausch zwischen der Enklave und der jeweiligen Partei durchgeführt, die Eingabe der Partei entschlüsselt, der Schnitt berechnet und ausgegeben. Im Unterschied zu dem Programm in Protokoll *protoRO* müssen hier, nachdem die Eingabe entschlüsselt wurde, die Signaturen auf die Hashwerte geprüft werden. Diese Signaturen wurden durch eine Enklave eid_i erzeugt, die bei Partei P_i installiert ist und den Code Ψ_{hash} ausführt. eid_i wird mit den Nachrichten für den Schlüsselaustausch zusammen übergeben. Dies stellt sicher, dass die Enklave *eid* später nur noch Ausgaben der Enklave eid_i akzeptiert. Damit die Enklave *eid* Signaturen von eid_i verifizieren kann, wird das Programm von *eid* mit dem Verifikationsschlüssel *mpk* parametrisiert. Die Enklave selbst kann nicht prüfen, ob dieser Schlüssel korrekt ist. Diese Aufgabe übernimmt die Partei P_i, wenn sie die Signaturen der Enklave *eid* verifiziert.

Programm 24 : $\Psi_{protoHASH}[mpk]$ Programm in $\mathcal{G}_{att}$

Phase: first key exchange

receive message m'_0, eid_0 **do**
  draw $a, b \in \{0, \ldots, |\mathcal{G}| - 1\}$ **uniformly at random**
  $m_0 := g^a$
  return (m'_0, m_0, eid_0)

Phase: second key exchange

receive message m'_1, eid_1 **do**
  $m_1 := g^b$
  return (m'_1, m_1, eid_1)

Phase: received first input

receive message S'_0 **do**
  $k_0 := (m'_0)^a$
  $(s_0, \sigma'_0) := Dec_{k_0}(S'_0)$
  assert $Verify(mpk, (eid_0, \Psi_{hash}, s_0), \sigma'_0) = 1$
  return "need input"

Phase: received second input

receive message S'_1 **do**
  $k_1 := (m'_1)^b$
  $(s_1, \sigma'_1) := Dec_{k_1}(S'_1)$
  assert $Verify(mpk, (eid_1, \Psi_{hash}, s_1), \sigma'_1) = 1$

Phase: calculate result

  $r' := s_0 \cap s_1$
  pad r' with random values up to ω elements
  sort r'
  return r'

4.4 Programmcode der Partei P_0 für Verfahren *protoHASH*

In Programm 25 ist der Code von Partei P_0 für Protokoll *protoHASH* dargestellt. Die Partei P_0 handelt mit der Partei P_1 durch $\mathcal{F}_{kex}$ einen Schlüssel k aus. Danach installiert P_0 eine Enklave mit Programm Ψ_{hash} und bekommt als Antwort eid_0. eid_0 wird mitgeschickt, wenn P_0 den Schlüsselaustausch mit der Enklave eid beginnt. Als Antwort erhält P_0 den Teil des Schlüsselaustauschs der Enklave zusammen mit der Signatur. Die Signatur wurde auch über die Eingabe von P_0 an die Enklave eid gebildet und sichert damit zu, dass T diese Eingabe nicht manipuliert hat. Außerdem sichert die Signatur zu, dass die Enklave das Programm $\Psi_{protoHASH}[mpk]$ mit korrektem mpk ausführt. Danach tauschen P_0 und P_1 die eid der Enklave aus, um sicherzustellen, dass beide mit der gleichen Enklave verbunden sind und beide den Schlüsselaustausch beendet haben. Dieser Schritt bietet Schutz gegen ein korrumpiertes T. P_0 verschlüsselt seine Eingabe elementweise mit dem Schlüssel k, fügt Padding hinzu und schickt dies an die Enklave eid_0, damit diese Hashwerte bildet. Das Ergebnis verschlüsselt P_0 an die Enklave eid und erhält den Schnitt der Hashwerte. Danach schneidet er das Ergebnis mit seiner eigenen Eingabe, in dem P_0 seine Eingabe verschlüsselt, davon Hashwerte bildet und prüft ob diese im Enklavenergebnis liegen. Die so erhaltene Menge meldet er an die Umgebung.

Der Code für die Partei P_1 ist dem von Partei P_0 sehr ähnlich und wird nicht explizit angegeben. Die Änderungen sind durch den Kontrollfluss bedingt und den Umstand, dass eine Partei den Schlüsselaustausch beginnen muss und eine Partei das Ergebnis des Protokolls zuerst erhält.

Programm 25 : Programmcode der Partei P_0

Phase: setup

on input S_0 **from** *Environment* **do**
  send subroutine input "getpk" **to** $\mathcal{G}_{att}$

receive message mpk **from** $\mathcal{G}_{att}$ **do**
  send subroutine input "new key, P_1" **to** $\mathcal{F}_{kex}$

on subroutine output k **from** $\mathcal{F}_{kex}$ **do**
  send subroutine input $install(\Psi_{hash})$ **to** $\mathcal{G}_{att}$

Phase: key exchange

on subroutine output eid_0 **from** $\mathcal{G}_{att}$ **do**
  draw $a' \in \{0, \ldots, |\mathcal{G}| - 1\}$ **uniformly at random**
  $m'_0 := g^{a'}$
  send (m'_0, eid_0) **to** T

receive message (m_0, σ_0, eid) **from** T **do**
  assert $Verify(mpk, (eid, \Psi_{protoHASH}[mpk], (m'_0, m_0, eid_0)), \sigma_0) = 1$
  send eid **to** P_1

receive message eid' **from** P_1 **do**
  if $eid \neq eid'$ **then**
    abort
  end

Phase: prepare input for enclave

  $s_0^* := \{Enc'_k(s_j) | s_j \in S_0\}$
  draw elements s_t **uniformly at random** and add $Enc'_k(s_t)$ to s_0^* until $|s_0^*| = \omega$
  send subroutine input $resume(eid_0, s_0^*)$ **to** $\mathcal{G}_{att}$

on subroutine output $(s_0, \sigma'_0, bits)$ **from** $\mathcal{G}_{att}$ **do**
  $k'_0 := m_0^{a'}$
  $S'_0 := Enc_{k'_0}((s_0, \sigma'_0))$
  send S'_0 **to** T

Phase: receive enclave output and calculate result for environment

receive message (r', σ_r) **from** T **do**
  assert $Verify(mpk, (eid, \Psi_{protoHASH}[mpk], r'), \sigma_r) = 1$
  calculate r by iterating over $a \in S_0$ and adding a to r, iff $h(Enc'_k(a)) \in r'$
  return r **to** *Environment*

4.5 Programmcode der Partei T für Verfahren *protoHASH*

In Programm 26 ist der Code von Partei T für Protokoll *protoHASH* dargestellt. T installiert eine Enklave mit Programm $\Psi_{protoHASH}[mpk]$ und leitet Nachrichten zwischen P_i und der Enklave weiter. Um den Programmcode mit dem korrektem Parameter installieren zu können, muss T den mpk von $\mathcal{G}_{att}$ anfragen.

Programm 26 : Programmcode der Partei T

Phase: setup and first key exchange

receive message m'_0, eid_0 **from** P_0 **do**
  send subroutine input "getpk" **to** $\mathcal{G}_{att}$

receive message mpk **from** $\mathcal{G}_{att}$ **do**
  send subroutine input $install(\Psi_{protoHASH}[mpk])$ **to** $\mathcal{G}_{att}$

on subroutine output eid **do**
  send subroutine input $resume(eid, (m'_0, eid_0))$ **to** $\mathcal{G}_{att}$

on subroutine output $((m'_0, m_0, eid_0), \sigma_0, bits)$ **from** $\mathcal{G}_{att}$ **do**
  send (m_0, σ_0, eid) **to** P_0

Phase: second key exchange

receive message m'_1, eid_1 **from** P_1 **do**
  send subroutine input $resume(eid, (m'_1, eid_1))$ **to** $\mathcal{G}_{att}$

on subroutine output $((m'_1, m_1, eid_1), \sigma_1, bits)$ **from** $\mathcal{G}_{att}$ **do**
  send (m_1, σ_1, eid) **to** P_1

Phase: first input for enclave

receive message S'_0 **from** P_0 **do**
  send subroutine input $resume(eid, S'_0)$ **to** $\mathcal{G}_{att}$

on subroutine output ("need input", $\sigma, bits$) **from** $\mathcal{G}_{att}$ **do**
  yield the execution **to** *Environment*

Phase: second input for enclave

receive message S'_1 **from** P_1 **do**
  send subroutine input $resume(eid, S'_1)$ **to** $\mathcal{G}_{att}$

Phase: send result

on subroutine output $(r', \sigma_r, bits)$ **from** $\mathcal{G}_{att}$ **do**
  send (r', σ_r) **to** P_0

receive message "result?" **from** P_1 **do**
  send (r', σ_r) **to** P_1

4.6 Sicherheitsbeweis für *protoHASH*

Für die verschiedenen Korruptionssituationen werden jeweils einzelne Simulatoren angegeben. Das ist möglich, weil wir von statischer Korruption ausgehen, d. h. ein Angreifer muss sich vor Beginn des Protokolls entscheiden ob und wen er korrumpieren will. Abhängig davon wählt der Simulator die entsprechende Variante. Der Fall, dass keine Partei korrumpiert ist und der Angreifer nur Netzwerknachrichten abfangen kann, ist in Unterabschnitt 4.6.1 beschrieben. In Unterabschnitt 4.6.2 ist der Fall beschrieben, dass Partei P_0 korrumpiert ist. Für P_1 ist ein analoger Sicherheitsbeweis möglich, der nicht explizit angegeben wird. Partei T kann auf zwei verschiedene Arten korrumpiert werden. In Unterabschnitt 4.6.3 wird der Beweis geführt für den Fall, dass T korrumpiert ist, aber die Enklave ehrlich bleibt. Dies entspricht der Korruption des Cloud-Betreibers, der nicht in den Zustand der Enklave schauen kann. In Unterabschnitt 4.6.3 wird auch der Beweis für ein korrumpiertes T, welches die Geheimnisse der Enklave lernt, da nur kleine Änderungen am Simulator benötigt wurden. Dies entspricht der Situation, dass der Prozessorhersteller des Cloud-Betreibers korrupt ist.

4.6.1 Keine Partei ist korrumpiert

In Programm 27 ist der Fall beschrieben, dass der Angreifer sich am Anfang des Ablaufs von Protokoll *protoHASH* dafür entscheidet keine Partei zu korrumpieren. In diesem Fall muss der Simulator $\text{Sim}^{protoHASH}$ keine Eingaben extrahieren und keine Enklaven installieren, da die Netzwerknachrichten über $\mathcal{F}_{smt}$ versendet werden. $\text{Sim}^{protoHASH}$ muss dem Angreifer nur die korrekte Länge und Abfolge der Nachrichten vortäuschen, da der Angreifer den Inhalt der Nachrichten im Realen nicht sieht.

Satz 4.1. *Sei $h(x)$ eine kollisionsresistente Hashfunktion. Dann gilt $protoHASH^{\mathcal{F}_{smt},\mathcal{F}_{kex},\mathcal{G}_{att}} \geq \mathcal{F}_{PSI}$, wenn der Angreifer $\mathcal{A}$ keinen korrumpiert.*

Der Beweis verläuft analog zu dem Beweis in Unterabschnitt 3.6.1 und wird hier nicht explizit angeführt. Als größere Änderung wollen wir hervorheben, dass in H_3 die Vernachlässigbarkeit einer Kollision zwischen den Eingaben S_0 und S_1 aus der Kollisionsresistenz von $h(x)$ gefolgert werden muss und nicht mehr auf eine Random Oracle Eigenschaft zurückgeführt werden kann.

Programm 27 : $\text{Sim}^{protoHASH}$ Simulator wenn keine Partei korrumpiert ist

Phase: setup

$\text{Sim}_{\square}^{protoHASH}(\perp, \perp)$ Angreifer initialisieren und Nachrichten weiterleiten wie in Programm 18

Phase: both key exchanges and inputs for enclave

receive message "input received" **from** $\mathcal{F}_{PSI}$ **do**
  simulate delayed output while leaking ("key exchange is carried out", P_0, P_1) **from** $\mathcal{F}_{kex}$ **to** $\mathcal{A}$
  simulate delayed output while leaking $(P_0, T, l_{dh} + l_{eid})$ **to** $\mathcal{A}$
  simulate delayed output while leaking $(T, P_0, l_{dh} + l_{sig} + l_{eid})$ **to** $\mathcal{A}$
  simulate delayed output while leaking (P_0, P_1, l_{eid}) **to** $\mathcal{A}$
  simulate delayed output while leaking "receiver is fetching key" **from** $\mathcal{F}_{kex}$ **to** $\mathcal{A}$
  simulate delayed output while leaking $(P_1, T, l_{dh} + l_{eid})$ **to** $\mathcal{A}$
  simulate delayed output while leaking $(T, P_1, l_{dh} + l_{sig} + l_{eid})$ **to** $\mathcal{A}$
  simulate delayed output while leaking (P_1, P_0, l_{eid}) **to** $\mathcal{A}$
  simulate delayed output while leaking $(P_0, T, e(\omega \cdot l_{hash} + l_{sig}))$ **to** $\mathcal{A}$
  yield the execution **to** *Environment*

on input *"intersect"* **from** $\mathcal{F}_{PSI}$ **do**
  simulate delayed output while leaking $(P_1, T, e(\omega \cdot l_{hash} + l_{sig}))$ **to** $\mathcal{A}$

Phase: send results

  simulate delayed output while leaking $(T, P_0, \omega \cdot l_{hash} + l_{sig})$ **to** $\mathcal{A}$
  return "result to P_0" **to** $\mathcal{F}_{PSI}$

on input "intersect 2" **from** $\mathcal{F}_{PSI}$ **do**
  simulate delayed output while leaking $(P_1, T, l(\text{"result?"}))$ **to** $\mathcal{A}$
  simulate delayed output while leaking $(T, P_1, \omega \cdot l_{hash} + l_{sig})$ to $\mathcal{A}$
  return "result to P_1" **to** $\mathcal{F}_{PSI}$

4.6.2 Partei P_0 ist korrumpiert

In Programm 28 ist der Fall beschrieben, dass der Angreifer am Anfang des Ablaufs sich dafür entscheidet Partei P_0 zu korrumpieren. Dabei ist nicht relevant, ob die von P_0 mit Programm Ψ_{hash} installierte Enklave Geheimnisse an P_0 verrät oder nicht, da diese Enklave keinen Zufall verwendet.

Satz 4.2. *Sei* (Enc, Dec) *ein IND-CCA sicheres symmetrisches Verschlüsselungsverfahren. Sei* (Enc', Dec') *ein det-IND-CPA sicheres deterministisches symmetrisches Verschlüsselungsverfahren. Sei das Signaturverfahren von* $\mathcal{G}_{att}$ *EUF-CMA sicher. Sei* $h(x)$ *eine kollisionsresistente Hashfunktion. Sei* g *ein Generator einer Gruppe* $\mathcal{G}$*, in der die DDH-Annahme gilt.*

Dann gilt $protoHASH^{\mathcal{F}_{smt},\mathcal{F}_{kex},\mathcal{G}_{att}} \geq \mathcal{F}_{PSI}$*, wenn der Angreifer* $\mathcal{A}$ *die Partei* P_0 *korrumpiert.*

Beweis. Der Beweis wird durch 5 Hybride H_i dargestellt, wobei H_0 den realen Ablauf abbildet und H_4 den idealen. Es wird gezeigt, dass die Umgebung die Hybride H_i und H_{i+1} nicht von einander unterscheiden kann. Sei out_i die Ausgabe der Umgebung bei Interaktion mit Hybrid H_i.

Hybrid H_1: Sei H_1 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_1$ und Simulator $\text{Sim}_1^{protoHASH}$, wobei $\mathcal{F}_1$ und $\text{Sim}_1^{protoHASH}$ wie folgt definiert sind: $\mathcal{F}_1$ ist eine ideale Funktionalität, die die Eingaben aller ehrlichen Parteien an den Simulator weiterleitet. Wenn der Simulator ein Ergebnis berechnet hat, dann gibt er dieses an die ideale Funktionalität weiter, damit es weiter an die Parteien geleitet wird. $\text{Sim}_1^{protoHASH}$ simuliert das gesamte Protokoll wie im Realen, mit Verwendung der echten Eingaben und generiert das reale Ergebnis für die Umgebung. Für die Nachrichten zwischen T und P_1 werden nicht mehr ihre echten Längen dem Angreifer gemeldet, sondern die im Pseudocode $\text{Sim}_{P_0}^{protoHASH}$ angegebenen Konstanten. Alle Enklaven werden nicht im Namen ihrer eigentlichen Besitzer anlegt, sondern im Namen der korrumpierten Partei P_0. Dies ist möglich, da die Signaturen der Enklaven anonym sind, da sie alle mit dem selben Schlüssel erzeugt werden. Der Simulator kann die Nachrichten an diese Enklaven, die von der Umgebung durch die korrumpierte Partei P_0 kommen, abfangen und zurückweisen. Direkt kann die Umgebung durch Kommunikation mit $\mathcal{F}_{att}$ nicht mit diesen Enklaven interagieren, da $\mathcal{F}_{att}$ die Enklaven immer gemeinsam mit dem Besitzer speichert, und die Umgebung nur im Namen neuer Parteien mit $\mathcal{F}_{att}$ kommunizieren kann.

Damit ist die Sicht der Umgebung auf das Protokoll identisch und es gilt:

$$Pr[out_0 = 1] - Pr[out_1 = 1] = 0$$

Hybrid H_2: Sei H_2 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_2$ und Simulator $Sim_2^{protoHASH}$, wobei $\mathcal{F}_2$ und $Sim_2^{protoHASH}$ wie folgt definiert sind: $\mathcal{F}_2$ ist identisch zu $\mathcal{F}_1$ außer, dass $\mathcal{F}_2$ die Menge S_0 als Eingabe von P_0 entgegen nimmt. $Sim_2^{protoHASH}$ ist identisch zu $Sim_1^{protoHASH}$ außer, dass $Sim_2^{protoHASH}$ die Eingabe von P_0 extrahiert, indem er die Kommunikation zwischen P_0 und $\mathcal{G}_{att}$ mitschreibt, nach einer passenden Enklave eid_0 sucht, die erste Eingabe s_0^* von P_0 an die Enklave eid_0 und deren Ausgabe s_0 ermittelt. $Sim_2^{protoHASH}$ berechnet $S_0 := \{Dec'_k(s) \mid s \in s_0^*\}$ und meldet S_0 an $\mathcal{F}_2$. Zusätzlich wird in der Enklave *eid*, nachdem sie ihre Eingabe entschlüsselt hat und die Signatur von eid_0 in $\Psi_{protoHASH}[mpk]$ in Zeile 11 in Programm 24 geprüft wurde, mit der echten Ausgabe s_0 der Enklave eid_0 weiter gerechnet.

Wir betrachten zwei Ereignisse:

E_{same} In $\Psi_{protoHASH}[mpk]$ bewirkt die Änderung von s_0 durch den Simulator nichts, da der neue Wert identisch zum Alten ist.

$E_{different}$ In $\Psi_{protoHASH}[mpk]$ bewirkt der Simulator eine echte Änderung des Inhalts von s_0, wenn er s_0 ersetzt.

In E_{same} sind die Hybride H_2 und H_3 identisch und die Umgebung kann sie nicht unterscheiden.

Wenn in $E_{different}$ die Umgebung H_2 von H_1 unterscheiden kann, kann man damit, wie in der nachfolgenden Reduktion angegeben, die EUF-CMA Sicherheit des Signaturschemas brechen: Der Simulator ersetzt den Wert von s_0 erst nachdem die Enklave *eid* die Signatur für das alte s_0 verifiziert wurde und dabei $Verify(mpk, (eid_0, \Psi_{hash}, s_0), \sigma'_0) = 1$ war. Wenn der Simulator eine echte Änderung des Inhalts bewirken kann, indem er s_0 ersetzen kann, dann hat der *Verify*-Aufruf in der Enklave *eid* eine Signatur als korrekt akzeptiert, die $\mathcal{G}_{att}$ nie erzeugt hat. Für die Reduktion auf die EUF-CMA Sicherheit wählt man *mpk* als den vom Challenger vorgegebenen Signaturschlüssel. Wenn $\mathcal{G}_{att}$ eine Enklavenausgabe signieren muss, wird diese Signatur beim Signaturorakel angefragt.

Wenn *Verify* eine Signatur wie oben beschrieben akzeptiert, wird diese zusammen mit der Nachricht eid_0, Ψ_{hash}, s_0 dem Challenger als Fälschung ausgeben. Damit ist die Wahrscheinlichkeit, dass H_2 von H_1 unterschieden werden kann durch eine vernachlässigbare Funktion $f_2(\lambda)$ beschränkt und es gilt:

$$|Pr[out_1 = 1] - Pr[out_2 = 1]| \leq f_2(\lambda)$$

Hybrid H_3: Sei H_3 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_3$ und Simulator $Sim_3^{protoHASH}$, wobei $\mathcal{F}_3$ und $Sim_3^{protoHASH}$ wie folgt definiert

sind: $\mathcal{F}_3 = \mathcal{F}_{int'}$. $\text{Sim}_3^{protoHASH}$ ist identisch zu $\text{Sim}_2^{protoHASH}$ außer, dass i $\text{Sim}_3^{protoHASH}$ als S_1 die von $\mathcal{F}_{int'}$ berechnete Ausgabe r benutzt. Dies sorgt dafür, dass die Enklave eid bis auf Padding dasselbe Ergebnis wie zuvor ausgibt, denn es gilt: $S_0 \cap r = S_0 \cap (S_0 \cap S_1) = S_0 \cap S_1 = r$. Der einzige Unterschied zwischen H_2 und H_3 ist, dass r nicht mehr auf Hashwerten berechnet wird. r kann sich nur verändern, wenn eine Kollision zwischen zwei Elementen $a, b \in S_0 \cup S_1$ mit $Enc'_k(a) \neq Enc'_k(b) \wedge h(Enc'_k(a)) = h(Enc'_k(b))$ vorliegt. Dann wäre beispielsweise $h(Enc'_k(a)) \in r'$ enthalten, aber a nicht in dem von $\mathcal{F}_{int'}$ berechneten r und der Schnitt ist in H_2 größer als in H_3. Die Hashfunktion ist kollisionsresistent. Deshalb ist die Wahrscheinlichkeit, dass eine solche Kollision auftritt, kleiner als eine vernachlässigbare Funktion $f_3(\lambda)$ und damit gilt:

$$|Pr[out_2 = 1] - Pr[out_3 = 1]| \leq f_3(\lambda)$$

Hybrid H_4 Sei H_4 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_4$ und Simulator $\text{Sim}_4^{protoHASH}$, wobei $\mathcal{F}_4$ und $\text{Sim}_4^{protoHASH}$ wie folgt definiert sind: $\mathcal{F}_4 = \mathcal{F}_{int'}$ und $\text{Sim}_4^{protoHASH} = \text{Sim}_{P_0}^{protoHASH}$. In der Enklave eid wird wieder das Programm $\Psi_{protoHASH}[mpk]$ ausgeführt, ohne die Änderungen aus H_2.

Analog wie in H_2 kann ein Angreifer H_3 von H_4 nur unterscheiden in dem er die von der Enklave eid verifizierte Signatur σ'_0 fälscht. Das ist höchstens mit einer vernachlässigbaren Wahrscheinlichkeit $f_4(\lambda)$ möglich und es gilt:

$$|Pr[out_3 = 1] - Pr[out_4 = 1]| \leq f_4(\lambda)$$

□

Programm 28 : $\text{Sim}_{P_0}^{protoHASH}$ Simulator wenn Partei P_0 korrumpiert ist

Phase: setup

$\text{Sim}_{\square}^{protoHASH}(P_0, P_0)$ Angreifer initialisieren und Nachrichten weiterleiten wie in Programm 18 bis auf Kommunikation von und zu $\mathcal{F}_{RO}$, welches nicht existiert.

on $corr_{out}$("new_key", P_1 **from** P_0 **to** $\mathcal{F}_{kex}$) **from** $\mathcal{A}$ **do**
  send subroutine input "getpk" **to** $\mathcal{G}_{att}$

receive message mpk **from** $\mathcal{G}_{att}$ **do**
  simulate delayed output while leaking ("key exchange is carried out", P_0, P_1) **from** $\mathcal{F}_{kex}$ **to** $\mathcal{A}$
  draw k **uniformly at random**
  pass $corr_{in}(k$ **from** $\mathcal{F}_{kex}$ **to** $P_0)$ **to** $\mathcal{A}$

Phase: first key exchange

on $corr_{out}((m_0', eid_0)$ **from** P_0 **to** $T)$ **from** $\mathcal{A}$ **do**
  send $corr_{out}(install(\Psi_{protoHASH}[mpk])$ **from** P_0 **to** $\mathcal{G}_{att})$ **to** P_0

receive message $corr_{in}(eid$ **from** $\mathcal{G}_{att}$ **to** $P_0)$ **from** P_0 **do**
  send $corr_{out}(resume(eid, (m_0', eid_0))$ **from** P_0 **to** $\mathcal{G}_{att})$ **to** P_0

receive message $corr_{in}(((m_0', m_0, eid_0), \sigma_0, bits)$ **from** $\mathcal{G}_{att}$ **to** $P_0)$ **from** P_0 **do**
  pass $corr_{in}((m_0, \sigma_0, eid)$ **from** T **to** $P_0)$ **to** $\mathcal{A}$

on $corr_{out}(eid$ **from** P_0 **to** $P_1)$ **from** $\mathcal{A}$ **do**
  simulate delayed output while leaking "receiver is fetching key" **from** $\mathcal{F}_{kex}$ **to** $\mathcal{A}$
  send $corr_{out}(install(\Psi_{hash})$ **from** P_0 **to** $\mathcal{G}_{att})$ **to** P_0

Phase: second key exchange

receive message $corr_{in}(eid_1$ **from** $\mathcal{G}_{att}$ **to** $P_0)$ **from** P_0 **do**
  simulate delayed output while leaking $(P_1, T, l_{dh} + l_{eid})$ **to** $\mathcal{A}$
  draw $b' \in \{0, \ldots, |\mathcal{G}| - 1\}$ **uniformly at random**
  $m_1' := g^{b'}$
  send $corr_{out}(resume(eid, (m_1', eid_1))$ **from** P_0 **to** $\mathcal{G}_{att})$ **to** P_0

receive message $corr_{in}(((m_1', m_1, eid_1), \sigma_1, bits)$ **from** $\mathcal{G}_{att}$ **to** $P_0)$ **from** P_0 **do**
  simulate delayed output while leaking $(T, P_1, l_{dh} + l_{sig} + l_{eid})$ **to** $\mathcal{A}$
  pass $corr_{in}(eid$ **from** P_1 **to** $P_0)$ **to** $\mathcal{A}$

Programm 28 : Fortsetzung von $\text{Sim}_{P_0}^{protoHASH}$

on $corr_{out}(S_0'$ **from** P_0 **to** $T)$ **from** $\mathcal{A}$ **do**
  send $corr_{out}(resume(eid, S_0')$ **from** P_0 **to** $\mathcal{G}_{att})$ **to** P_0

Phase: extract input from P_0

receive message $corr_{in}((\text{“need input”}, \sigma, bits)$ **from** $\mathcal{G}_{att}$ **to** $P_0)$ **from** P_0 **do**
  suche in allen vom Angreifer gestellten Anfragen an $\mathcal{G}_{att}$ nach der Enklaveninstallation, die eid_0 zurückgegeben hat. Für diese Enklaveninstallation: prüfe, dass mit dieser eid_0 mindestens einmal *resume* aufgerufen wurde und das installierte Programm Ψ_{hash} ist. Wenn dies nicht der Fall ist, breche ab.
  bestimme s_0^* aus dem ersten *resume* Aufruf von eid_0
  $S_0 := \{Dec_k'(s) \mid s \in s_0^*\}$
  send $corr_{out}(S_0$ **from** P_0 **to** $\mathcal{F}_{PSI})$ **to** P_0

on subroutine output “input received” **from** $\mathcal{F}_{PSI}$ **do**
  yield the execution **to** *Environment*

on input *“intersect”* **from** $\mathcal{F}_{PSI}$ **do**
  return “result to P_0” **to** $\mathcal{F}_{PSI}$

receive message $corr_{in}(r$ **from** $\mathcal{F}_{PSI}$ **to** $P_0)$ **do**
  $s_1^* := \{Enc_k'(s_j) | s_j \in r\}$
  draw elements s_t **uniformly at random** and add $Enc_k'(s_t)$ to s_1^* until $|s_1^*| = \omega$
  send $corr_{out}(resume(eid_1, s_1^*)$ **from** P_0 **to** $\mathcal{G}_{att})$ **to** P_0

Phase: simulate network traffic for P_1

receive message $corr_{in}((s_1, \sigma_1', bits)$ **from** $\mathcal{G}_{att}$ **to** $P_0)$ **from** P_0 **do**
  simulate delayed output while leaking $(P_1, T, e(\omega \cdot l_{hash} + l_{sig}))$ **to** $\mathcal{A}$
  $k_1' := m_1^{b'}$
  $S_1' := Enc_{k_1'}((s_1, \sigma_1'))$
  send $corr_{out}(resume(eid, S_1')$ **from** P_0 **to** $\mathcal{G}_{att})$ **to** P_0

Phase: simulate result

receive message $corr_{in}((r', \sigma_r, bits)$ **from** $\mathcal{G}_{att}$ **to** $P_0)$ **from** P_0 **do**
  pass $corr_{in}((r', \sigma_r)$ **from** T **to** $P_0)$ **to** $\mathcal{A}$

on input *“intersect 2”* **from** $\mathcal{F}_{PSI}$ **do**
  simulate delayed output while leaking $(P_1, T, l(\text{“result?”}))$ **to** $\mathcal{A}$
  simulate delayed output while leaking $(T, P_1, \omega \cdot l_{hash} + l_{sig})$ to $\mathcal{A}$
  return “result to P_1” **to** $\mathcal{F}_{PSI}$

4.6.3 Partei T ist mit oder ohne seine Enklave korrumpiert

Satz 4.3. *Sei* (Enc, Dec) *ein IND-CCA sicheres symmetrisches Verschlüsselungsverfahren. Sei* (Enc', Dec') *ein det-IND-CPA sicheres deterministisches symmetrisches Verschlüsselungsverfahren. Sei das Signaturverfahren von* $\mathcal{G}_{att}$ *EUF-CMA sicher. Sei* $h(x)$ *eine kollisionsresistente Hashfunktion. Sei* g *ein Generator einer Gruppe* $\mathcal{G}$*, in der die DDH-Annahme gilt.*

Dann gilt $protoHASH^{\mathcal{F}_{smt}, \mathcal{F}_{kex}, \mathcal{G}_{att}} \geq \mathcal{F}_{PSI}$*, wenn der Angreifer* $\mathcal{A}$ *die Partei* T *korrumpiert, aber die Enklave ehrlich bleibt.*

Beweis. Der Beweis wird durch 7 Hybride H_i dargestellt, wobei H_0 den realen Ablauf abbildet und H_6 den idealen. Es wird gezeigt, dass die Umgebung die Hybride H_i und H_{i+1} nicht von einander unterscheiden kann. Sei out_i die Ausgabe der Umgebung bei Interaktion mit Hybrid H_i.

Hybrid H_1: Sei H_1 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_1$ und Simulator $\mathrm{Sim}_1^{protoHASH}$, wobei $\mathcal{F}_1$ und $\mathrm{Sim}_1^{protoHASH}$ wie folgt definiert sind: $\mathcal{F}_1$ ist eine ideale Funktionalität, die die Eingaben aller ehrlichen Parteien an den Simulator weiterleitet. Wenn der Simulator ein Ergebnis berechnet hat, dann gibt er dieses an die ideale Funktionalität weiter, damit es weiter an die Parteien geleitet wird. $\mathrm{Sim}_1^{protoHASH}$ simuliert das gesamte Protokoll exakt wie im Realen, mit Verwendung der echten Eingaben und generiert das reale Ergebnis für die Umgebung. Die Nachrichten zwischen P_0 und P_1 werden durch zufällige Nachrichten mit der Länge l_{eid} ersetzt. Einem Angreifer fällt diese Änderung nicht auf, da die Nachrichten konstante Länge haben, über $\mathcal{F}_{smt}$ gesendet werden und der Angreifer nur die Länge lernt. Alle Enklaven werden nicht im Namen ihrer eigentlichen Besitzer anlegt, sondern im Namen der korrumpierten Partei T angelegt. Dies ist aus denselben Gründen möglich wie in H_1, im Falle das P_0 korrumpiert ist.

Die Ausgaben der Umgebung sind somit identisch verteilt. Daher gilt:

$$Pr[out_0 = 1] - Pr[out_1 = 1] = 0$$

Hybrid H_2: Sei H_2 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_2$ und Simulator $\mathrm{Sim}_2^{protoHASH}$, wobei $\mathcal{F}_2$ und $\mathrm{Sim}_2^{protoHASH}$ wie folgt definiert sind: $\mathcal{F}_2 = \mathcal{F}_1$. $\mathrm{Sim}_2^{protoHASH}$ bricht ab, wenn P_0 und P_1 verschiedene Werte für eid erhalten und lässt P_i direkt mit der Ausgabe der Enklave eid weiter rechnen.

Wir betrachten die folgenden Ereignisse:

E_{eid} P_0 und P_1 erhalten von T verschiedene Werte für eid.

E_{output} E_{eid} tritt nicht ein und T hat die Enklave eid mit falschem Programmcode angelegt oder übermittelt die Ausgabe der Enklave nicht korrekt.

E_{input} E_{eid} und E_{output} treten nicht ein und T übermittelt die Eingabe in die Enklave nicht korrekt.

E_{correct} Die Enklave *eid* wird von T mit korrektem Programmcode angelegt, die Eingabe wird von T korrekt in die Enklave übermittelt und die Ausgabe wird von T korrekt an P_i bzw. den Simulator.

Tritt Ereignis E_{eid} ein, brechen P_0 und P_1 die Berechnung direkt nach Erhalt der *eid* von T ab, wenn sie die *eid* vergleichen.

Tritt Ereignis E_{output} ein, muss *Verify* fehlschlagen oder der Angreifer eine Signatur fälschen, denn $\mathcal{F}_{att}$ hat nie eine Nachricht mit *eid* und $\Psi_{protoHASH}[mpk]$ unterschrieben. Dieselbe Überlegung gilt dafür, wenn die Ausgabe der Enklave verändert wird. In E_{input} hat die Enklave ihr korrektes Programm und gibt ihre Eingabe unverändert aus, daher muss auch hier *Verify* fehlschlagen oder der Angreifer eine Signatur fälschen. Zur Reduktion von E_{output} und E_{input} auf die EUF-CMA Sicherheit, werden im EUF-CMA Sicherheitsspiel alle Signaturen von $\mathcal{F}_{att}$ mit dem Signaturorakel durchgeführt. Die Signatur des Angreifers, die in E_{output} oder E_{input} von *Verify* akzeptiert wird, kann dem Challenger als gefälschte Signatur gemeldet werden. Weil das Signaturverfahren EUF-CMA sicher ist, kann eine solche Fälschung höchstens mit der vernachlässigbaren Wahrscheinlichkeit $f_2(\lambda)$ auftreten.

In E_{correct} verhalten sich H_1 und H_2 gleich, weil sie mit den gleichen Werten rechnen.

$$|Pr[out_1 = 1] - Pr[out_2 = 1]| \leq f_2(\lambda)$$

H_3 und H_4 entsprechen jeweils H_4 und H_5 aus dem Beweis für Protokoll *protoRO* für den Fall, dass T korrumpiert ist, aber die Enklave ehrlich bleibt, und werden unverändert übernommen. In H_3 wird der durch den Diffie-Hellman-Schlüsselaustausch gezogenen Schlüssel durch Zufall ersetzt. In H_4 wird S_i' durch Verschlüsselung von Zufall ersetzt. Ein verändertes Chiffrat wird mit dem vom Challenger bereitgestellten Entschlüsselungsorakel entschlüsselt und die enthaltene Signatur geprüft, damit der Simulator genau dann abbricht, wenn in H_3 die Enklave auch abgebrochen hätte. Die Eingabe an die Enklave selbst für die weitere Berechnung bleibt unverändert. Damit ist sichergestellt, dass die Enklave mit korrekten Eingaben rechnet.

Hybrid H_5: Sei H_5 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_5$ und Simulator $\text{Sim}_5^{protoHASH}$, wobei $\mathcal{F}_5$ und $\text{Sim}_5^{protoHASH}$ wie folgt definiert sind: $\mathcal{F}_5 = \mathcal{F}_{int'}$. $\text{Sim}_5^{protoHASH}$ berechnet kein Ergebnis, sondern lässt dies direkt in $\mathcal{F}_{int'}$ berechnen. Der Schnitt wird nun nicht mehr auf den Hashwerten sondern direkt zwischen S_0 und S_1 gebildet.

T muss S_i korrekt an die Enklave übermitteln, da die Enklave bereits eid_i kennt und nur Werte

S_i' akzeptiert, wenn die in der Verschlüsselung enthaltenen Hashwerte eine gültige Signatur der Enklave eid_i haben. Auch die Ausgabe r' ist mit einer solchen Enklavensignatur abgesichert. Damit kann die Enklave *eid* direkt mit den (gehashten und verschlüsselten) Eingaben S_i rechnen und die Ausgabe setzen. Die Wahrscheinlichkeit, dass ein das gesamte Experiment eine der Signaturen fälschen kann ist kleiner als die vernachlässigbare Funktion $f_5(\lambda)$.

Damit diese Umformung das Ergebnis ändert, muss eine Kollision zwischen zwei Elementen $a, b \in S_0 \cup S_1$ mit $Enc'_k(a) \neq Enc'_k(b) \wedge h(Enc'_k(a)) = h(Enc'_k(b))$ vorliegt. Da aber Enc' eine umkehrbare Verschlüsselung ist und h kollisionsresistent ist, ist die Wahrscheinlichkeit für eine Kollision kleiner als einer vernachlässigbare Funktion $f_5'(\lambda)$ und es gilt:

$$|Pr[out_4 = 1] - Pr[out_5 = 1]| \leq f_5(\lambda) + f_5'(\lambda)$$

H_6 entspricht H_8 aus dem Beweis für Protokoll *protoRO* für den Fall, dass T korrumpiert ist, aber die Enklave ehrlich bleibt, und wird unverändert übernommen. In H_6 wird der zufällige Schlüssel k_i wieder durch den im Diffie-Hellman-Schlüsselaustausch gezogenen Schlüssel ersetzt. Damit ist $\text{Sim}_6^{protoHASH} = \text{Sim}_{T_{\text{only}}}^{protoHASH}$. □

Satz 4.4. *Sei (Enc', Dec') ein det-IND-CPA sicheres deterministisches symmetrisches Verschlüsselungsverfahren. Sei das Signaturverfahren von $\mathcal{G}_{att}$ EUF-CMA sicher. Sei $h(x)$ eine kollisionsresistente Hashfunktion. Sei g ein Generator einer Gruppe $\mathcal{G}$, in der die DDH-Annahme gilt. $protoHASH^{\mathcal{F}_{smt}, \mathcal{F}_{kex}, \mathcal{G}_{att}} \geq \mathcal{F}_{PSI}$, wenn der Angreifer $\mathcal{A}$ die Partei T korrumpiert und die Enklave belauscht.*

Beweis. Der Beweis wird durch 7 Hybride H_i dargestellt, wobei H_0 den realen Ablauf abbildet und H_6 den idealen. Es wird gezeigt, dass die Umgebung die Hybride H_i und H_{i+1} nicht von einander unterscheiden kann. Sei out_i die Ausgabe der Umgebung bei Interaktion mit Hybrid H_i.

H_1 und H_2 werden unverändert aus dem Beweis für den Fall, dass T korrumpiert ist, aber die Enklave ehrlich bleibt, übernommen.

Hybrid H_3: Sei H_3 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_3$ und Simulator $\text{Sim}_3^{protoHASH}$, wobei $\mathcal{F}_3$ und $\text{Sim}_3^{protoHASH}$ wie folgt definiert sind: $\mathcal{F}_3 = \mathcal{F}_2$. $\text{Sim}_3^{protoHASH}$ bricht ab, wenn an die Enklave *eid* eine falsche Eingabe S_i, wie in $\text{Sim}_{T_{\text{enclave}}}^{protoHASH}$ angegeben, übermittelt wird.

Grundsätzlich unterscheiden sich die Experimente nur, wenn $\text{Sim}_3^{protoHASH}$ abbricht, obwohl die Enklave dies nicht getan hätte. Wir betrachten die 2 Varianten, in denen $\text{Sim}_3^{protoHASH}$ abbricht getrennt:

E_{none} Die Enklave *eid* akzeptiert die Signatur und T wurde bisher noch kein Wert für S_i' übermittelt.

E_{wrong} Die Enklave *eid* akzeptiert die Signatur und der Wert $\tilde{s_i}$ im Chiffrat entspricht nicht dem Wert s_i, der an T gesendet wurde.

In Ereignis E_{none} wurde bisher keine Signatur auf ein Enklavenergebnis von Enklave eid_i erstellt. Damit stellt die Signatur, die T an die Enklave *eid* übermittelt eine Signatur auf eine neue Nachricht da, mit der man das EUF-CMA Sicherheitsspiel gewinnen kann.

In Ereignis E_{wrong} akzeptiert *Verify* die Signatur für eine neue Nachricht, für die bisher keine Signatur erzeugt wurde. Auch in diesem Fall kann man damit das EUF-CMA Sicherheitsspiel gewinnen.

Da Signaturverfahren EUF-CMA sicher ist, ist damit die Wahrscheinlichkeit, dass die Ereignisse E_{none} oder E_{wrong} eintreten kleiner als eine vernachlässigbare Funktion $f_3(\lambda)$. Damit gilt:

$$|Pr[out_2 = 1] - Pr[out_3 = 1]| \leq f_3(\lambda)$$

Hybrid H_4: Wird von H_5 aus dem Beweis für den Fall, dass T korrumpiert ist, aber die Enklave ehrlich bleibt, übernommen. Damit muss in $\mathcal{F}_4$ die Ausgabe der Parteien P_i nicht vom Simulator gesetzt werden.

Hybrid H_5: Sei H_5 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_5$ und Simulator $\text{Sim}_5^{protoHASH}$, wobei $\mathcal{F}_5$ und $\text{Sim}_5^{protoHASH}$ wie folgt definiert sind: $\mathcal{F}_5 = \mathcal{F}_4$. $\text{Sim}_5^{protoHASH}$ verschlüsselt nicht die echten Eingaben S_i mit Enc_k sondern zieht zufällige Werte. Wie in $\text{Sim}_{T_{\text{enclave}}}^{protoHASH}$ angegeben, wird dabei die von $\mathcal{F}_{int'}$ übermittelte Schnittgröße $|r|$ genutzt, um für gleiche Werte in den Eingabemengen S_i auch gleiche Zufallswerte zu verschlüsseln.

Der Schlüssel k wird vom Simulator zufällig gezogen und zu Beginn nur für Enc_k benutzt. Eine Reduktion auf die det-IND-CPA Eigenschaft von Enc_k ist wie folgt möglich: Der Simulator zieht keinen Schlüssel k und gibt immer, wenn er $Enc_k(x)$ berechnen müsste das Tupel aus dem Wert aus S_i und dem Zufallswert an den det-IND-CPA Challenger. Abhängig von seinem Bit b verschlüsselt dieser entweder die erste oder die zweite Eingabe. Ist $b = 0$, so verläuft das Experiment genau wie in H_2, ist $b = 1$ so verläuft das Experiment wie in H_3. Wird mehrfach für dieselbe Nachricht ein Chiffrat benötigt, so ist dabei auch der Zufallswert gleich, und es kann direkt die Antwort aus einer vorherigen Anfrage übernommen werden. Dadurch wird in dieser Reduktion nie zweimal für die selbe Nachricht ein Chiffrat angefragt. Erkennt die Umgebung, dass das Experiment H_2 durchgeführt wurde, wird dem det-IND-CPA Challenger

$b = 0$ gemeldet. Ansonsten wird $b = 1$ gemeldet. Da aber Enc_k det-IND-CPA sicher ist, ist diese Wahrscheinlichkeit, dass zwischen $b = 0$ und $b = 1$ unterschieden werden kann, kleiner als eine vernachlässigbare Funktion $f_5(\lambda)$. Daher gilt:

$$|Pr[out_4 = 1] - Pr[out_5 = 1]| \leq f_5(\lambda)$$

Hybrid H_6: Sei H_6 das Experiment zwischen der Umgebung, dem idealen Protokoll mit idealer Funktionalität $\mathcal{F}_6$ und Simulator $\mathrm{Sim}_6^{protoHASH}$, wobei $\mathcal{F}_6$ und $\mathrm{Sim}_6^{protoHASH}$ wie folgt definiert sind: $\mathcal{F}_6 = \mathcal{F}_{int'}$, $\mathrm{Sim}_6^{protoHASH} = \mathrm{Sim}_{T_{\text{enclave}}}^{protoHASH}$. Weil die Eingaben S_i in H_5 durch Zufall ersetzt wurden, werden sie nicht benötigt und $\mathcal{F}_6$ muss sie dem Simulator nicht senden. Es gilt:

$$Pr[out_5 = 1] - Pr[out_6 = 1] = 0$$

□

Programm 29 : $\mathrm{Sim}^{protoHASH}_{T_{\text{type}}}$, type $\in$ {only, enclave}

Phase: setup

$\mathrm{Sim}^{protoHASH}_{\square}(T, T_{\text{type}})$ Angreifer initialisieren und Nachrichten weiterleiten wie in Programm 18
on subroutine output "input received" **from** $\mathcal{F}_{PSI}$ **do**
  simulate delayed output while leaking ("key exchange is carried out", P_0, P_1) **from** $\mathcal{F}_{kex}$ **to** $\mathcal{A}$
  send $corr_{out}(install(\Psi_{hash})$ **from** T **to** $\mathcal{G}_{att})$ **to** T

Phase: first key exchange and input for enclave

receive message $corr_{in}(eid_0$ **from** $\mathcal{G}_{att}$ **to** $T)$ **from** T **do**
  draw $a' \in \{0, \ldots, |\mathcal{G}| - 1\}$ **uniformly at random**
  $m'_0 := g^{a'}$
  pass $corr_{in}((m'_0, eid_0)$ **from** P_0 **to** $T)$ **to** $\mathcal{A}$

on $corr_{out}((m_0, \sigma_0, eid)$ **from** T **to** $P_0)$ **from** $\mathcal{A}$ **do**
  breche ab, wenn Enklave eid nicht den Programmcode $\Psi_{protoHASH}[mpk]$ hat, nicht m'_0, eid_0 als Eingabe bekommen hat oder nicht m'_0, m_0, eid_0 ausgegeben hat
  bei Eintreffen der nächsten Eingaben S'_i in die Enklave eid, entschlüssle $\tilde{s_i} := Dec_{k'_i}(S'_i)$ und wenn S'_i noch nicht an T gesendet wurde oder wenn $\tilde{s_i} \neq s_i$, breche ab.
  assert $Verify(mpk, (eid, \Psi_{protoHASH}[mpk], (m'_0, m_0, eid_0)), \sigma_0) = 1$
  simulate delayed output while leaking (P_0, P_1, l_{eid}) **to** $\mathcal{A}$
  simulate delayed output while leaking "receiver is fetching key" **from** $\mathcal{F}_{kex}$ **to** $\mathcal{A}$
  send $corr_{out}(install(\Psi_{hash})$ **from** T **to** $\mathcal{G}_{att})$ **to** T

Phase: second key exchange and input for enclave

receive message $corr_{in}(eid_1$ **from** $\mathcal{G}_{att}$ **to** $T)$ **from** T **do**
  draw $b' \in \{0, \ldots, |\mathcal{G}| - 1\}$ **uniformly at random**
  $m'_1 := g^{b'}$
  pass $corr_{in}((m'_1, eid_1)$ **from** P_1 **to** $T)$ **to** $\mathcal{A}$

on $corr_{out}((m_1, \sigma_1, eid)$ **from** T **to** $P_1)$ **from** $\mathcal{A}$ **do**
  assert $Verify(mpk, (eid, \Psi_{protoHASH}[mpk], (m'_1, m_1, eid_1)), \sigma_1) = 1$
  simulate delayed output while leaking (P_1, P_0, l_{eid}) **to** $\mathcal{A}$
  draw k **uniformly at random**
  draw elements s_t **uniformly at random** and add $Enc'_k(s_t)$ to s^*_0 until $|s^*_0| = \omega$
  send $corr_{out}(resume(eid_0, s^*_0)$ **from** T **to** $\mathcal{G}_{att})$ **to** T

receive message $corr_{in}((s_0, \sigma'_0, bits)$ **from** $\mathcal{G}_{att}$ **to** $T)$ **from** T **do**
  $k'_0 := m_0^{a'}$
  $S'_0 := Enc_{k'_0}((s_0, \sigma'_0))$
  pass $corr_{in}(S'_0$ **from** P_0 **to** $T)$ **to** $\mathcal{A}$

Programm 29 : Fortsetzung von $\text{Sim}^{protoHASH}_{T_{\text{type}}}$

```
33 if type = enclave then
34     receive message corr_in(|r| from F_PSI to T) do
35         send corr_out("size received" from T to F_PSI)
36 end
37
38 on input "intersect" from F_PSI do
39     s_1^* := ∅
40     if type = enclave then
41         draw t := |r| distinct elements out of s_0^* uniformly at random
42         s_1^* := t
43     draw elements s_t uniformly at random and add Enc'_k(s_t) to s_1^* until |s_1^*| = ω
44     send corr_out(resume(eid_1, s_1^*) from T to G_att) to T
45
46 receive message corr_in((s_1, σ'_1, bits) from G_att to T) from T do
47     k'_1 := m_1^{b'}
48     S'_1 := Enc_{k'_1}((s_1, σ'_1))
49     pass corr_in(S'_1 from P_1 to T) to A
                                        Phase: send result
50 on corr_out((r', σ_r) from T to P_0) from A do
51     assert Verify(mpk, (eid, Ψ_protoHASH[mpk], r'), σ_r) = 1
52     return "result to P_0" to F_PSI
53
54 on input "intersect 2" from F_PSI do
55     pass corr_in("result?" from P_1 to T)
56
57 on corr_out((r', σ_r) from T to P_1) from A do
58     assert Verify(mpk, (eid, Ψ_protoHASH[mpk], r'), σ_r) = 1
59     return "result to P_1" to F_PSI
```

5 Zusammenfassung

In dieser Arbeit wurden zwei Protokolle, *protoRO* und *protoHASH*, zur privaten Schnittmengenberechnung mit Hilfe einer dritten Partei vorgestellt. Beide Protokolle wurden unter unterschiedlichen Annahmen im UC-Framework als sicher bewiesen.

Es hat sich gezeigt, dass Enklaven ein nützliches Werkzeug sind, um private Schnittmengenberechnung effizient und UC-sicher zu realisieren. Die Modellierung im UC-Framework offenbart viele subtile Fehler, die bei der Entwicklung von einem Protokoll auftreten können. Die natürliche Weise die gewünschte Funktionalität des Protokolls durch eine ideale Funktionalität zu beschreiben erlaubt recht einfach eine starke Sicherheitsdefinition anzugeben. Bei der Führung des Sicherheitsbeweises fallen auch kleine Abweichungen von der idealen Funktionalität auf. Man kann sich entscheiden, ob man die Schwachstelle toleriert und in der idealen Funktionalität abbildet oder das Protokoll verbessert.

Beim Entwickeln von UC-sicheren Protokollen mit Enklaven ist es grundsätzlich ein Problem, wenn Parteien, die selbst keine Enklaven erzeugen können, durch das Protokoll Zugriff auf gültige Enklavensignaturen bekommen. Um dieses Problem zu lösen, wird in der Praxis auf Gruppensignaturen zurückgegriffen. Diese erlauben auf der einen Seite, dass Signaturen anonym unter allen Teilnehmern eines Protokolls sind, aber trotzdem zwingend von einem der Teilnehmer stammen müssen. Diese zusätzliche Komplexität wird durch die Verwendung der idealen Funktionalität für einen Attested Execution Processor abgefangen, sodass man sich bei der Protokollentwicklung nicht direkt damit beschäftigen muss.

Bei UC-sicheren Protokollen kann es zu Problemen kommen, wenn es gewünscht ist, dass eine Partei auf ihre Eingabe eine Hashfunktion anwendet. Dann muss ein Simulator für den Sicherheitsbeweis dazu in der Lage sein, aus korrumpierten Parteien ihre Eingaben zu extrahieren. Enklaven können bei der Extraktion behilflich sein, weil der Simulator die Eingabe in die Enklave beobachten kann.

Weiterführende Arbeiten könnten die vorgestellten Protokolle auf n Parteien erweitern und die genauen Auswirkungen auf die Protokollkomplexität untersuchen.

Außerdem könnten sich weitere Arbeiten damit beschäftigen, ein Protokoll zu entwerfen, dass ohne dritte Partei auskommt. Nimmt man beispielsweise bei *protoHASH* sowohl T als auch eine der Parteien P_i gleichzeitig korrumpiert werden, kann man *protoHASH* als zwei

Parteien Protokoll betrachten, indem man T mit einer der Parteien P_i vereinigt. Dabei muss genau beobachtet werden, welche Auswirkungen dies auf die Sicherheit des Protokolls hat. Diese intuitive Adaption von *protoHASH* realisiert nicht mehr $\mathcal{F}_{PSI}$. Daher ist interessant zu überlegen, ob und wie ein ähnliches Sicherheitsniveau erreicht werden kann.

Literatur

[Aft14] Parry Aftab. *Findings under the Personal Information Protection and Electronic Documents Act(PIPEDA)*. https://parryaftab.blogspot.com/2014/03/what-does-whatsapp-collect-that.html. 2014.

[Bah+17] Raad Bahmani u. a. „Secure multiparty computation from SGX“. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2017, S. 477–497.

[Bar+16] Manuel Barbosa u. a. „Foundations of hardware-based attested computation and application to SGX“. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE. 2016, S. 245–260.

[Blu83] Manuel Blum. „Coin flipping by telephone a protocol for solving impossible problems“. In: *ACM SIGACT News* 15.1 (1983), S. 23–27.

[BS17] Dan Boneh und Victor Shoup. *A Graduate Course in Applied Cryptography*. 2017.

[Can+07] Ran Canetti u. a. „Universally composable security with global setup“. In: *Theory of Cryptography Conference*. Springer. 2007, S. 61–85.

[Can01] Ran Canetti. „Universally composable security: A new paradigm for cryptographic protocols“. In: *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symposium on*. IEEE. 2001, S. 136–145.

[CD16] Victor Costan und Srinivas Devadas. *Intel SGX Explained*. Techn. Ber. https://eprint.iacr.org/2016/086. 2016.

[CK02] Ran Canetti und Hugo Krawczyk. *Universally Composable Notions of Key Exchange and Secure Channels*. Cryptology ePrint Archive, Report 2002/059. https://eprint.iacr.org/2002/059. 2002.

[DT10] Emiliano De Cristofaro und Gene Tsudik. „Practical private set intersection protocols with linear complexity“. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2010, S. 143–159.

[FS90] Uriel Feige und Adi Shamir. „Witness indistinguishable and witness hiding protocols“. In: *Proceedings of the twenty-second annual ACM symposium on Theory of computing*. ACM. 1990, S. 416–426.

[GN17] Satrajit Ghosh und Tobias Nilges. *An Algebraic Approach to Maliciously Secure Private Set Intersection*. Cryptology ePrint Archive, Report 2017/1064. `https://eprint.iacr.org/2017/1064`. 2017.

[HEK12] Yan Huang, David Evans und Jonathan Katz. „Private set intersection: Are garbled circuits better than custom protocols?“ In: *NDSS*. 2012.

[Kam+14] Seny Kamara u. a. „Scaling private set intersection to billion-element sets“. In: *International Conference on Financial Cryptography and Data Security*. Springer. 2014, S. 195–215.

[KL14] Jonathan Katz und Yehuda Lindell. *Introduction to modern cryptography*. Chapman und Hall/CRC, 2014.

[Mar17] Moxie Marlinspike. *Private contact discovery for Signal*. `https://signal.org/blog/private-contact-discovery/`. 2017.

[Nie02] Jesper Buus Nielsen. „Separating Random Oracle Proofs from Complexity Theoretic Proofs: The Non-committing Encryption Case“. In: *Advances in Cryptology — CRYPTO 2002*. Hrsg. von Moti Yung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, S. 111–126.

[Pin+18] Benny Pinkas u. a. *Efficient Circuit-based PSI via Cuckoo Hashing*. Cryptology ePrint Archive, Report 2018/120. `https://eprint.iacr.org/2018/120`. 2018.

[PST17] Rafael Pass, Elaine Shi und Florian Tramer. „Formal abstractions for attested execution secure processors“. In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2017, S. 260–289.

[Van+18] Jo Van Bulck u. a. „Foreshadow: Extracting the Keys to the Intel {SGX} Kingdom with Transient Out-of-Order Execution“. In: *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 2018, S. 991–1008.

[Wei+18] Ofir Weisse u. a. *Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution*. Techn. Ber. 2018.

[YY08] Tian Yuan und Wang Ying. *GUC-Secure Set-Intersection Computation*. Cryptology ePrint Archive, Report 2008/392. `https://eprint.iacr.org/2008/392`. 2008.

Anhang

1 Simulator für korrumpierte Partei P_1 in Verfahren *protoRO*

In Programm 30 wird der Vollständigkeit halber der Simulator für *protoRO* angegeben für den Fall, dass P_1 korrumpiert ist. Er ist sehr ähnlich zu dem Simulator für den Fall, dass P_0 korrumpiert ist.

Programm 30 : $\text{Sim}_{P_1}^{protoRO}$

Phase: setup

$\text{Sim}_{\square}^{protoRO}(P_1, P_1)$ Angreifer initialisieren und Nachrichten weiterleiten wie in Programm 18
on subroutine output "input received" **from** $\mathcal{F}_{PSI}^{weak}$ **do**
  pass ("key exchange is carried out", P_0, P_1) **from** $\mathcal{F}_{kex}$ **to** $\mathcal{A}$

Phase: simulate network traffic of P_0

on *"ok P_0"* **to** $\mathcal{F}_{kex}$ **from** $\mathcal{A}$ **do**
  simulate delayed output while leaking (P_0, T, l_{dh}) **to** $\mathcal{A}$
  simulate delayed output while leaking $(T, P_0, l_{dh} + l_{sig} + l_{eid}))$ **to** $\mathcal{A}$
  simulate delayed output while leaking $(P_0, T, e((\omega + 1) \cdot l_{hash})$ **to** $\mathcal{A}$ yield the execution **to** *Environment*

on $corr_{out}$("get key" **from** P_1 **to** $\mathcal{F}_{kex}$) **from** $\mathcal{A}$ **do**
  pass "receiver is fetching key" **from** $\mathcal{F}_{kex}$ **to** $\mathcal{A}$

on *"ok P_1"* **to** $\mathcal{F}_{kex}$ **from** $\mathcal{A}$ **do**
  draw k **uniformly at random**
  pass $corr_{in}(k$ **from** $\mathcal{F}_{kex}$ **to** $P_1)$ **to** $\mathcal{A}$

on $corr_{out}(m_1'$ **from** P_1 **to** $T)$ **from** $\mathcal{A}$ **do**
  send $corr_{out}$("crs" **from** P_1 **to** $\mathcal{G}_{acrs})$

receive message $corr_{in}((epk, \_, crs)$ **from** $\mathcal{G}_{acrs}$ **to** $P_1)$ **do**
  send $corr_{out}$("idk" **from** P_1 **to** $\mathcal{G}_{acrs})$

Phase: second key exchange

receive message $corr_{in}(idk[P_1]$ **from** $\mathcal{G}_{acrs}$ **to** $P_1)$ **do**
  draw eid **uniformly at random**
  draw $b \in \{0, \ldots, |\mathcal{G}| - 1\}$ **uniformly at random**
  $m_1 := g^b$
  draw σ_1, ξ **uniformly at random**
  $C_1 := Enc_{epk}((\sigma_1, idk[P_1]), \xi)$
  $\pi_1 := NIWI.Prove(crs, (P_1, (eid, \Psi_{protoRO}, (m_1', m_1)), C_1), (\xi, \sigma_1, idk[P_1]))$
  pass $corr_{in}((m_1, C_1, \pi_1, eid)$ **from** T **to** $P_1)$ **to** $\mathcal{A}$

Programm 30 : Fortsetzung von $\text{Sim}_{P_1}^{protoRO}$

Phase: extract input from P_1

on $corr_{out}(S_1'$ **from** P_1 **to** $T)$ **from** $\mathcal{A}$ **do**
  $k_1 := (m_1')^b$
  $s_1 := Dec(k_1, S_1')$
  if *first element of* $s_1 \neq h(\perp \| k)$ **then**
    abort
  $s_1' :=$ drop first element of s_1
  $S_1 := \{x | h(x\|k) \in s_1'\}$ by looking every element of s_1' in the List of Random Oracle queries
  send $corr_{out}(S_1$ **from** P_1 **to** $\mathcal{F}_{PSI}^{weak})$ **to** P_1

on input *"intersect"* **from** $\mathcal{F}_{PSI}^{weak}$ **do**
  simulate delayed output while leaking $(T, A, (\omega + 1) \cdot l_{hash} + l_{sig})$ **to** $\mathcal{A}$
  return "result to P_0" **to** $\mathcal{F}_{PSI}^{weak}$

when the Environment resumes execution do
  pass execution to $\mathcal{A}$

on $corr_{out}$("result?" **from** P_1 **to** $T)$ **from** $\mathcal{A}$ **do**
  send $corr_{out}$("execution resumed" **from** P_1 **to** $\mathcal{F}_{PSI}^{weak})$ **to** $\mathcal{F}_{PSI}^{weak}$

on input *"intersect 2"* **from** $\mathcal{F}_{PSI}^{weak}$ **do**
  return "result to P_1" **to** $\mathcal{F}_{PSI}^{weak}$

Phase: simulate result

receive message $corr_{in}(r$ **from** $\mathcal{F}_{PSI}^{weak}$ **to** $P_1)$ **do**
  calculate $h(x)$ by asking $\mathcal{F}_{RO}$ via the corrupted P_1:
  $r' := \{h(\gamma\|k) | \gamma \in r\}$
  pad r' with random elements
  sort r'
  add $h(\perp||k)$ to r' as first element
  draw σ_r, ξ **uniformly at random**
  $C_{r1} := Enc_{epk}((\sigma_R, idk[P_1]), \xi)$
  $\pi_{r1} := NIWI.Prove(crs, (P_1, (eid, \Psi_{protoRO}, r'), C_{r1}), (\xi, \sigma_r, idk[P_1]))$
  pass $corr_{in}((r', C_{r1}, \pi_{r1})$ **from** T **to** $P_1)$ **to** $\mathcal{A}$