Blinde Turingmaschinen

Monika Brodbeck

Alpen-Adria-Universität Klagenfurt
mbrodbec@edu.aau.at

Zusammenfassung

Die Verarbeitung und Speicherung von Daten wird in zunehmendem Maße ins Internet ausgelagert, etwa durch die Nutzung von Cloud Diensten. Dabei ist es wichtig, die Sicherheit dieser Daten auch während deren Verarbeitung zu gewährleisten. Um sich vor Angriffen wie etwa einem Datendiebstahl zu schützen, werden vertrauliche Daten vor einer Auslagerung in der Regel verschlüsselt. Dadurch wird jedoch eine Analyse oder Verarbeitung der Daten schwierig. An diesem Punkt setzt die Blinde Turingmaschine an: Sie ermöglicht es, Berechnungen und Analysen auf verschlüsselten Daten durchzuführen. Nach der Betrachtung einiger dafür notwendiger Grundlagen zeigt diese Arbeit die Implementierung eines Prototyps, der verschlüsselten Assemblercode verarbeitet, aufbauend auf dem Konzept der Blinden Turingmaschinen. Außerdem werden Sicherheitsprobleme bei Implementierung sowie verschiedene Angriffsmöglichkeiten und deren Gegenmaßnahmen betrachtet.

1 Einleitung

Bei Firmen als auch bei Privatpersonen wird Cloud Computing immer beliebter. Gerade Firmen verwenden die Cloud hierbei nicht rein als Datenspeicher, sondern mieten auch Rechenkapazität und lagern ganze Berechnungen aus. Der Vorteil hierbei ist, dass Rechenkapazität nach Bedarf hinzugefügt oder entfernt werden kann und somit geringere Kosten entstehen. Die Sicherheit ist jedoch eines der grundlegenden Probleme beim Auslagern von Daten. Hierbei geht es nicht nur um das eigene Risiko, Daten zu verlieren, sondern auch um rechtliche Probleme. So gibt es z.B. Gesetze, die besagen, dass bestimmte Daten die EU nicht verlassen dürfen. Da man jedoch beim Cloud Computing über den tatsächlichen Standort der Daten meist nicht selbst entscheiden kann, hat man dies nicht mehr selbst in der Hand und kann nicht dafür garantieren, dass die Daten in der EU bleiben. Die Daten müssen daher geschützt werden.

Die naheliegendste Möglichkeit hierfür ist, die Daten zu verschlüsseln. Werden Daten nur in der Cloud gespeichert, ist dies kein Problem. Will man jedoch Berechnungen ausführen, ist eine Verschlüsselung im Allgemeinen eher hinderlich. Das Konzept der Blinden Turingmaschinen soll diesem Problem entgegenwirken: Es erlaubt, beliebige Berechnungen auf verschlüsselten Daten auszuführen. Um Berechnungen auszuführen benötigt man einen speziell dafür vorgesehenen Schlüssel, den sogenannten Evaluierungsschlüssel. Dieser kann nicht entschlüsseln, wohl aber Berechnungen auf den verschlüsselten Daten durchführen, welche wiederum ein verschlüsseltes Ergebnis liefern. Auf die Cloud bezogen bekommt nun der Cloud-Provider diesen Evaluierungsschlüssel. Somit können die Berechnungen in der Cloud ausgeführt werden, wobei die Daten zu keiner Zeit im Klartext vorliegen. Entschlüsselt werden die Ergebnisse erst lokal im eigenen System.

P. Schartner · K. Lemke-Rust · M. Ullmann (Hrsg.) · D•A•CH Security 2015 · syssec (2015) 122-131.

2 Stand der Forschung

Es gibt bereits unterschiedliche Konzepte, die sich diesem Problem widmen und Berechnungen auf vertraulichen Daten in der Cloud ermöglichen. Dies sind voll-homomorphe Verschlüsselung, Multiparty-Computation und Garbled Circuits. Die beiden letzteren bringen allerdings den Nachteil mit sich, dass eine Berechnung mehrere Instanzen und somit auch gesicherte Kommunikationskanäle zwischen diesen Instanzen benötigt. Berechnungen anhand voll-homomorpher Verschlüsselung können von einer einzelnen Instanz auf verschlüsselten Daten ausgeführt werden, sind aber für eine praktische Anwendung noch nicht ausreichend effizient. Eine direkte Simulation Blinder Turingmaschinen ist zwar auch nicht effizienter (siehe [DuMP14]), die Anwendung des Prinzips auf Assembler-Code wie nachfolgend erklärt jedoch schon. Ein entscheidender Vorteil ist hierbei auch, dass direkt auf Code gearbeitet wird und keine aufwändige Transformation (wie etwa in einem Schaltkreis) nötig ist.

3 Konstruktion Blinder Turingmaschinen

Um eine Blinde Turingmaschine zu konstruieren, benötigt man ein Verschlüsselungs-System mit besonderen Eigenschaften: Dies ist einerseits die sogenannte Gruppen-Homomorphie, andererseits eine Vergleichsoperation. Die Gruppen-Homomorphie erlaubt es, eine einzelne Operation entweder vor oder nach der Verschlüsselung auszuführen, und dasselbe Ergebnis zu erhalten. Die Vergleichsoperation erlaubt autorisierten Instanzen, die einen speziell dafür vorgesehenen Evaluierungsschlüssel besitzen, Vergleiche von verschlüsselten Klartexten durchzuführen, ohne zu diesem Zweck entschlüsseln zu müssen. Für Instanzen ohne diesen Schlüssel bleiben Chiffrate weiterhin ununterscheidbar [Tang12].

3.1 Sicherheit

Ein Angreifer, der den Evaluierungsschlüssel besitzt, kann Chiffrate vergleichen. Daher hat er mehr Informationen als ein anderer Angreifer, der beispielsweise Chiffrate auf dem Kommunikationsweg abfängt. Daher werden bei einem Verschlüsselungsverfahren mit Vergleichsoperation zwei verschiedene Angreifertypen betrachtet: Typ-1-Angreifer besitzen den Evaluierungsschlüssel, Typ-2-Angreifer kennen ihn nicht. Für diese können daher höhere Sicherheitsanforderungen gelten.

Typ-1-Angreifer: Ein solcher Angreifer kann zwei beliebige Chiffrate vergleichen. Daher kann für ihn nicht gefordert werden, dass Chiffrate ununterscheidbar sind. Ein Typ-1-Angreifer darf aber auch anhand einer Chosen-Ciphertext-Attacke keine Informationen über den Klartext gewinnen können. In der Praxis ist ein Typ-1-Angreifer beispielsweise der Cloud-Provider, in dessen Cloud eine sichere Berechnung ausgelagert wird.

Typ-2-Angreifer: Dieser Angreifer besitzt keinen Evaluierungsschlüssel und kann daher keinen Vergleich durchführen. Hier kann somit gefordert werden, dass Chiffrate ununterscheidbar sind, der Angreifer also für zwei beliebige Chiffrate nicht effizient entscheiden kann, ob sie den gleichen Klartext repräsentieren oder nicht. In der Praxis kann dies ein Angreifer sein, der die Kommunikation passiv abhört.

3.2 Konstruktion

Nun kann eine Turingmaschine zu einer Blinden Turingmaschine erweitert werden. Dazu wird der Bandinhalt Zeichen für Zeichen verschlüsselt (vergleiche Electronic Codebook Mode). Um

einen Übergang zu bestimmen, muss daher ein Vergleich mittels Evaluierungsschlüssel durchgeführt werden, da nur so entschieden werden kann, welches der passende Übergang ist. Um

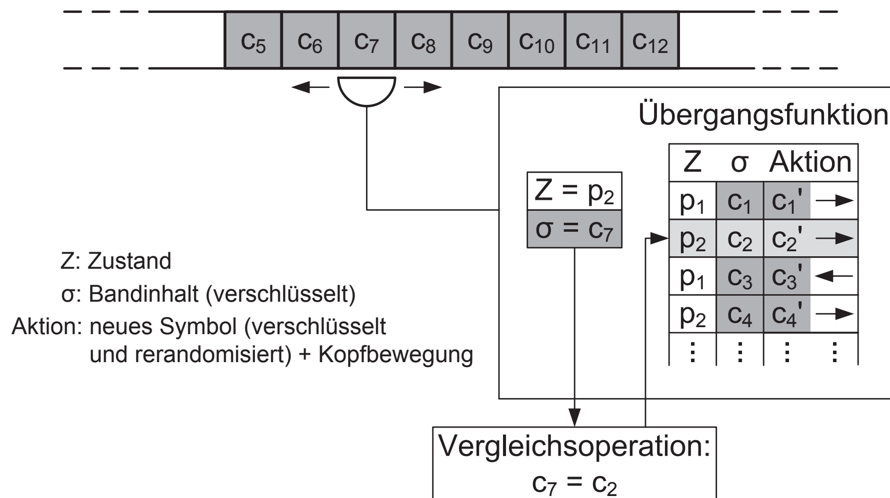

Abb. 1: Übersicht über die Arbeitsweise einer blinden Turingmaschine

die Ununterscheidbarkeit der Chiffrate auch nach Berechnungen zu erhalten, muss außerdem eine Re-Randomisierung durchgeführt werden: Das neue Zeichen wird nicht einfach auf das Band geschrieben, stattdessen liefert die Überführungsfunktion die Differenz des neuen Werts zum gelesenen Zeichen. Das neue Zeichen wird dann durch Multiplikation der Chiffrate unter Ausnützung der Homomorphie-Eigenschaften der Verschlüsselungsfunktion bestimmt. Dadurch fließt in jedes geschriebene Chiffrat die Zufallskomponente des bereits vorhandenen Chiffrats ein. Somit entstehen auch bei mehrmaliger Ausführung desselben Übergangs verschiedene, insbesondere ununterscheidbare Chiffrate [Rass13]. Eine Übersicht über den Aufbau und Ablauf einer Blinden Turingmaschine gibt Abbildung 1. Die hier verschlüsselten Bandsymbole sind dunkelgrau markiert. Aufgrund der ununterscheidbaren Chiffrate kann nur mit Hilfe der Vergleichsoperation der benötigte Übergang (hellgrau) bestimmt werden.

4 Verschlüsselter Assembler-Code

Dieses Prinzip kann nun auf Assembler-Code angewendet werden. Die Register-Inhalte sowie Konstanten im Code werden mit einer Erweiterung der ElGamal-Verschlüsselung verschlüsselt. Statt der Überführungsfunktion gibt es für jede Operation eine sogenannte Lookup-Table. Diese benötigt als Eingabe ein Commitment der beiden Operanden. (Dieses kann anhand der Vergleichsoperation gewonnen werden). Sie liefert dann die verschlüsselte Differenz aus Ergebnis und erstem Operanden. Damit kann mittels Re-Randomisierung ein Chiffrat des Ergebnisses berechnet werden.

Diese Lookup-Table benötigen jedoch bei n-Bit langen Zahlen $(2^n)^2 = 2^{2n}$ Einträge. Bei 32- oder 64-Bit Wörtern würden sie daher sehr groß werden. Um die Anzahl möglicher Werte und somit die Lookup-Tables kleiner zu halten, werden diese Lookup-Tables nur auf 4-Bit-Basis erstellt. Operationen auf größeren Werten werden emuliert. Dazu werden die Werte in 4-Bit Blöcke aufgeteilt und blockweise verschlüsselt. Die Tabelle muss dann Ergebnis und Übertrag liefern. Damit kann, abhängig von der Operation, der entstehende Übertrag berücksichtigt und

mit nachfolgenden Chiffratteilen verknüpft werden. Tabelle 1 zeigt ein Beispiel für eine solche Lookup-Table. Der erste Wert ist jeweils das Ergebnis (die zwei niedrigstwertigsten Bits), der zweite Wert der Übertrag (die höherwertigen Bits). Betrachten wir als Beispiel die Multiplikation $2 \cdot 3$. Die entsprechenden Commitments bekommen wir mit dem Evaluierungsschlüssel und wissen somit, dass das Ergebnis in der dritten Zeile und der zweiten Spalte zu finden ist. Vom Ergebnis muss der erste Operand, in diesem Fall $Enc(2, k)$, abgezogen werden. Aufgrund der Homomorphie-Eigenschaft erhalten wir damit $Enc(0, k) - Enc(2, k) = Enc(2, k)$ (es wird Modulo 4 gerechnet, da es sich um 2-Bit-Werte handelt). Mit dem Übertrag verfährt man genauso und erhält $Enc(1, k)$. Konkateniert man die beiden Werte erhält man den korrekten Ergebniswert $(0110)_2 = (6)_{10}$. Da später ein Operand aus mehreren Blöcken zusammengesetzt ist, müssen Ergebnis und Übertrag getrennt zurückgegeben werden, da der Übertrag erst noch mit höherwertigen Blöcken verknüpft werden muss.

Tab. 1: Beispiel für eine Lookup-Table: Multiplikation von 2-Bit-Werten

·	Enc(0,k)	Enc(1,k)	Enc(2,k)	Enc(3,k)
Enc(0,k)	(Enc(0,k),Enc(0,k))	(Enc(0,k),Enc(0,k))	(Enc(0,k),Enc(0,k))	(Enc(0,k),Enc(0,k))
Enc(1,k)	(Enc(3,k),Enc(3,k))	(Enc(0,k),Enc(3,k))	(Enc(1,k),Enc(3,k))	(Enc(2,k),Enc(3,k))
Enc(2,k)	(Enc(2,k),Enc(2,k))	(Enc(0,k),Enc(2,k))	(Enc(2,k),Enc(3,k))	(Enc(0,k),Enc(3,k))
Enc(3,k)	(Enc(1,k),Enc(1,k))	(Enc(0,k),Enc(1,k))	(Enc(3,k),Enc(2,k))	(Enc(2,k),Enc(3,k))

Als Beispiel hierfür werden wir die Implementierung der Addition betrachten. Die Addition beginnt bei den niedrigstwertigsten Blöcken. Der entstandene Übertrag muss zum Ergebnis der nachfolgenden Addition addiert werden. Hierbei kann allerdings wieder ein Übertrag entstehen. Dies ist in Abbildung 2 dargestellt. Um zu verhindern, dass in jedem Schritt mehr Überträge hinzukommen, werden immer zuerst die beiden Überträge addiert. Ein Übertrag kann bei der Addition maximal Eins betragen. Bei einer Addition von zwei Überträgen ist der Übertrag damit sicher Null und kann daher verworfen werden. Das Ergebnis dieser Addition wird danach zum Ergebnis der Addition der beiden Operandenblöcke addiert. Hierbei kann wiederum ein Übertrag entstehen, welcher als $carry2$ in der Abbildung dargestellt ist.

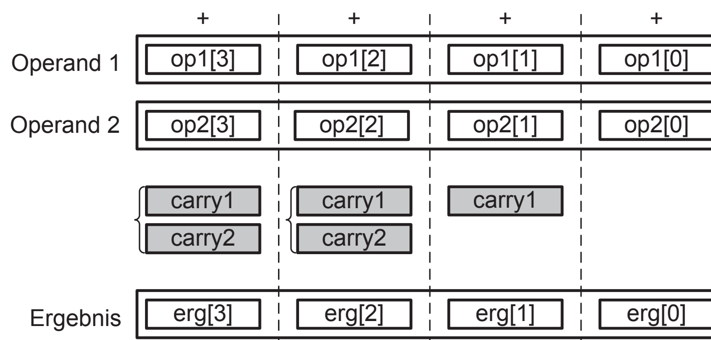

Abb. 2: Die Implementierung der Addition und die dabei enstehenden Überträge

Bei 4-Bit großen Blöcken gibt es allerdings nur 16 mögliche Werte, womit der Klartextraum sehr klein ist. Dies führt bei vorhandener Vergleichsoperation zu folgendem Problem: Bei einer asymmetrischen Verschlüsselung kann sich ein Angreifer problemlos ein Chiffrat jedes

Klartextwertes erzeugen und damit jedes Chiffrat anhand von Vergleichen entschlüsseln. Damit solch ein Angriff nicht möglich ist, muss verhindert werden, dass ein Angreifer, welcher im Besitz des Evaluierungsschlüssels ist, Chiffrate mit bekanntem Wert erzeugen kann. Eine Erzeugung über Verschlüsselung kann verhindert werden, indem die Werte vor der Verschlüsselung codiert (d.h. durch zufällig gewählte Repräsentanten ersetzt) werden und diese Codierung geheim gehalten wird.

5 Verwendetes Verschlüsselungssystem

In der Implementierung des Prototyps für die Blinden Turingmaschinen wurde die von Damgård erweiterte ElGamal-Verschlüsselung verwendet. Hier gibt es eine zusätzliche dritte Komponente um Chosen-Ciphertext-Attacken entgegenzuwirken [Damg92]. Zusätzlich wird dann noch die Vergleichsoperation benötigt. Dazu wird ein zweiter öffentlicher Schlüssel (pk_2) eingeführt. Der dazugehörige private Schlüssel sk_2 entspricht dem Evaluierungsschlüssel. Zusätzlich müssen zwei erzeugende Elemente der Gruppe C der Chiffrate – g und h – bestimmt werden. In die Verschlüsselung wird nun zusätzlich das für die Lookup-Tables benötigte Commitment eingebettet:

$$Enc(m, (pk_1, pk_2)) = (c_1, c_2, c_3) = (Enc(m, pk_1), g^m h^r, Enc(h^r, pk_2))$$

Nur mit dem Evaluierungsschlüssel (pk_1) kann das zur Durchführung von Berechnungen benötigte Commitment g^m bestimmt werden. Der zum öffentlichen Schlüssel passende private Schlüssel sk_1 dient zur Entschlüsselung der Nachricht und muss geheim gehalten werden. [Rass13]

Ein Chiffrat im Prototyp besteht damit immer aus einem Commitment sowie zwei Damgård ElGamal-Verschlüsselungen, welche jeweils aus drei Komponenten bestehen. Insgesamt besteht ein Chiffrat somit aus sieben Komponenten.

6 Angriffsszenarien und Gegenmaßnahmen

Der Evaluierungsschlüssel bietet jedoch auch die Möglichkeit, Berechnungen auszuführen. Daher ist es möglich, Chiffrate mit bekanntem Wert mittels Berechnungen zu erzeugen. Eine einfache Möglichkeit ist, einen Wert von sich selbst abzuziehen – damit erhält man die Null – oder einen von Null verschiedenen Wert (durch Anwendung der Vergleichsoperation erkennbar) durch sich selbst zu dividieren – womit die Eins erzeugt wird. Schon allein mit diesen beiden Werten ist es möglich, ein Chiffrat zu jedem Klartext-Werte zu erzeugen. Eine einfache Möglichkeit, dem entgegenzuwirken wäre, die Division eines Chiffrates durch sich selbst zu verbieten. Die Null lässt sich damit zwar weiterhin erzeugen, die Eins jedoch nicht mehr.

6.1 Angriffe auf die implementierten Algorithmen

Auch manche der implementierten Algorithmen bieten Angriffsmöglichkeiten. Hier werden Probleme, die sich bei der Implementierung der Multiplikation und der Division ergeben haben, vorgestellt.

6.1.1 Division

Für die Division wird eine Schleife benötigt, wobei in jedem Durchlauf der Wert des berechneten Ergebnisblockes um Eins inkrementiert wird. Hierbei ergibt sich das Problem, dass ein

Inkrementieren auf verschlüsselten Werten nicht ohne Weiteres möglich ist. Die Eins zu verschlüsseln ist aufgrund der unbekannten Codierung nicht möglich. Daher wird dieser Schritt anhand einer eigenen Inkrementations-Lookup-Table durchgeführt. Diese arbeitet auf einem einzelnen Operanden und liefert jeweils den Wert inkrementiert um Eins. Das Problem mit einer solchen Tabelle ist, dass wiederum sehr einfach alle Zahlen erzeugt werden können, sobald man wenigstens eine Zahl kennt. Da man die Null anhand der Subtraktion und Vergleichen erkennt, ist dies ein Problem.

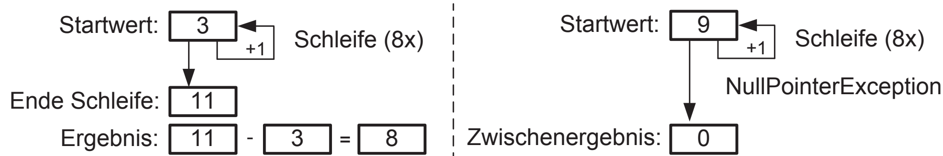

Abb. 3: Inkrementieren Divison: Zwischenergebnis Null führt zu `NullPointerException`

Daher wird die Tabelle lediglich für Werte ungleich Null erstellt. Ein Aufruf des Wertes Null auf dieser Tabelle würde damit jedoch zu einer `NullPointerException` führen. Der Ergebnisblock wird daher mit einem Zufallswert initialisiert. Nun kann wiederholt inkrementiert werden. Der Zufallswert wird am Ende wieder abgezogen, um das korrekte Ergebnis zu erhalten. Dies ist in Abbildung 3 auf der linken Seite dargestellt. Allerdings kann es bei diesem Vorgehen passieren, dass eines der Zwischenergebnisse Null beträgt, wie auf der rechten Seite dargestellt. Dann würde es zu einer `NullPointerException` kommen, wenn versucht wird, weiter zu inkrementieren. Diese Exception wird daher abgefangen, das Zwischenergebnis, welches in diesem Fall Null beträgt, wird durch einen neuen Zufallswert ersetzt. Nun muss am Ende nur jeder dieser Zufallswerte abgezogen werden. Da Modulo 16 gerechnet wird, erhält man dann am Ende wieder das korrekte Ergebnis, was auch in Abbildung 4 nochmals dargestellt ist.

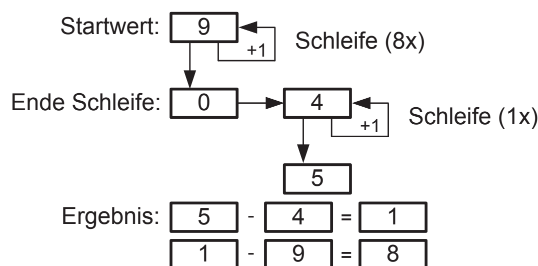

Abb. 4: Inkrementieren mit mehrmaligem Addieren einer Zufallszahl

6.1.2 Multiplikation

Beim Multiplikations-Algorithmus ergibt sich ein Problem aufgrund des verwendeten Zweierkomplements. Werden zwei n-Bit lange Zahlen multipliziert, so erhält man ein $2n$-Bit langes Ergebnis. Im Zweierkomplement ergibt sich dabei aber das Problem, dass bei negativen Zahlen alle Bits nach der n-ten Stelle falsch berechnet werden, da hier die bei negativen Zahlen vorne stehenden Einsen nicht mehr berücksichtigt werden. Um dieses Problem zu umgehen, müssen diese vor der Multiplikation ergänzt werden (die Zahl auf doppelte Länge erweitert werden).

Damit ein Angreifer hier nicht sofort erkennen kann, ob eine Zahl positiv oder negativ ist, muss eine solche Erweiterung auch für positive Zahlen gemacht werden, auch wenn hier lediglich mit Nullen aufgefüllt wird. Abbildung 5 zeigt die Erweiterung eines negativen Operanden auf doppelte Länge. Kennt ein Typ-1-Angreifer hier bereits die Null (hellgraue Blöcke), kann er die so hinzugefügten Blöcke auf Null vergleichen (in der Abbildung als *com* dargestellt). Findet er einen Block ungleich Null (dunkelgrau) weiß er, dass es sich um einen Block aus 4 Einsern handelt, also den Wert 15. Mit $0 - 15 \equiv 1 \mod 16$ kann er sich dann die Eins berechnen.

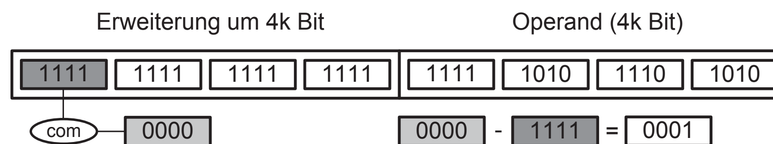

Abb. 5: Erweiterung eines (negativen) Operanden auf doppelte Länge

Diesen Angriff könnte man einerseits umgehen, indem man die Erweiterung der Operanden streicht, womit das Ergebnis automatisch auf die Länge der Operanden gekürzt würde. Bei großen Operanden kann dies allerdings zu fehlerhaften Ergebnissen führen. Das Problem der Operandenerweiterung könnte vollständig umgangen werden, wenn statt des Zweierkomplements das Einerkomplement verwendet wird. Dies bringt allerdings eine vorzeichenbehaftete Null mit sich. Die anderen Algorithmen sowie die Vergleichsoperation müssten dann entsprechend angepasst werden.

6.2 Angriff über Manipulation der Operanden

Solange es möglich ist, die Null zu erzeugen, ist es aufgrund der obligatorischen Re-Randomisierung immer noch möglich, die Eins durch Division zu erzeugen. Ein Beispiel dafür zeigt der folgende Code:

```
lw $s0 Enc(offset_1)($zero)
sub $s1, $s0, $s0        #$s1 = E(0)
add $s2, $s0, $s1        #D($s0) = D($s2)
div $s2, $s0
mflo $s3                 #$s3 = E(1)
```

Die Addition von Null in der dritten Zeile re-randomisiert den Wert aus Register `$s2`. Der Wert ändert sich bei einer Addition mit Null nicht, wohl aber das Chiffrat. Damit hat man zwei Chiffrate, die für denselben Wert stehen. Nun ist eine Division im HSM möglich. Der resultierende Wert, der am Ende in Register `$s3` steht, ist Eins.

Selbst ohne die Null ist es möglich, ein Chiffrat für die Eins – und damit auch für alle anderen Werte – zu berechnen. Durch Addition einer Zahl zu sich selbst und darauffolgender Division lässt sich zum Beispiel die Zwei erzeugen:

$$A := x + x = 2x$$
$$B := A/x = 2x/x = 2$$

Durch zusätzliche Additionen ($A := x+x+x = 3x$) können weitere Werte erzeugt werden. Sobald man die Drei berechnet hat, kann man wie auch bei der Eins durch wiederholte Additionen für jeden Wert ein Chiffrat erzeugen.

Besonders der letzte Angriff zeigt, dass sich die Erzeugung von Chiffraten für jeden möglichen Wert kaum verhindern lässt. Damit kann ein Typ-1-Angreifer nun jedoch ein Probe-Chiffrat für jeden möglichen Wert erzeugen und somit auch unbekannte verschlüsselte Klartexte durch Vergleichen mit diesen Probe-Chiffraten aufdecken. Die Möglichkeit für einen solchen Vergleich ist Voraussetzung für die Konstruktion Blinder Turingmaschinen.

6.3 Hardware-Security-Modul

Um einen Angriff in den beschriebenen Formen zu verhindern, haben wir ein Hardware-Security-Modul (HSM) eingeführt (das System wurde also konzeptuell in eine Two-Party Computation abgeändert, wobei die zweite Instanz jedoch lokal und mit geringeren Ressourcen als bei alternativen Ansätzen wie Multiparty-Computation oder Garbled Circuits implementiert werden kann). Evaluierungsschlüssel, also Vergleichsoperation sowie die Lookup-Tables, werden in dieses HSM verlagert. Nun bekommt auch die Ausführungsumgebung (beispielsweise die Cloud) keinen Evaluierungsschlüssel. Berechnungen wie oben dargestellt können zwar weiterhin ausgeführt werden, jedoch kann mit den erzeugten Chiffraten kein Vergleich mehr durchgeführt werden. Einzige Möglichkeit wäre nun, große Mengen an Chiffraten zu erzeugen und zu sammeln, in der Hoffnung, genau ein solches bereits bekanntes Chiffrat zu erhalten. Gibt es jedoch ausreichend viele Möglicheiten, ein und denselben Klartext zu verschlüsseln, so ist die Wahrscheinlichkeit hierfür gering genug, um vernachlässigbar zu sein. Der Cloud-Provider kann trotz des HSMs den Code und die Werte manipulieren und damit gezielt Operationen auf von ihm gewünschten Werten (mit eventuell bekanntem Klartext) ausführen. Wichtig ist jedoch, dass der Evaluierungsschlüssel das HSM nie verlässt, sodass der Cloud-Provider keine Möglichkeit hat, zwei Chiffrate auf Gleichheit zu prüfen.

6.4 Angriff durch Ausnützung bedingter Sprünge

Solange ein Angreifer beliebigen Code ans HSM übergeben kann, hat er aber eventuell doch die Möglichkeit, einen Vergleich auszuführen. Dies geht anhand der bedingten Sprünge. Der im Rahmen dieser Arbeit implementierte Prototyp unterstützt vier dieser bedingten Sprünge: `beq` (branch if equal), `bne` (branch if not equal), `blt` (branch if lower than) und `bgt` (branch if greater than). Diese werden im verschlüsselten Code zwar verschleiert, sodass die vier Sprünge für einen Angreifer zunächst nicht unterscheidbar sind, ein Angriff ist aber trotzdem möglich. Abbildung 6 zeigt für welche Wertebereiche jeweils ein Sprung ausgeführt wird. Hier sieht man

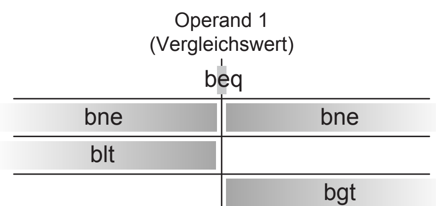

Abb. 6: Übersicht über die vier bedingten Sprünge

auch, dass `beq` die einzige Anweisung ist, welche den Sprung bei Gleichheit ausführt. Somit kann man über den Vergleich eines Chiffrats mit sich selbst diesen Befehl bestimmen, womit man wieder eine Vergleichsoperation hat und somit das System kompromittieren kann. Auch solch ein Angriff kann jedoch verhindert werden, indem man die Sprünge um einen else-Zweig erweitert. Ein Angreifer kann diesen nicht vom if-Zweig unterscheiden und somit nicht feststellen, wann ein Sprung ausgeführt wird. Ein verschleierter, bedingter Sprung bekommt also

immer zwei Sprungziele und es wird immer ein Sprung zu einer dieser Adressen durchgeführt. Wohin gesprungen wird, hängt von der Bedingung ab.

6.5 Seitenkanal-Attacken

Es gibt noch weitere Angriffsmöglichkeiten, die durch die Aufteilung der Operanden in Blöcke entstanden sind. So wird zum Beispiel ein Ergebnis-Block bei der Division über wiederholtes Subtrahieren berechnet. Hier wäre demnach eine Seitenkanal-Attacke möglich, indem gezählt wird, wie oft die Schleife durchlaufen wird. Mit der Anzahl der Schleifen-Durchläufe kennt man den Wert des Ergebnis-Blocks und damit ein Zwischenergebnis. Da auch die Zwischenergebnisse sensible Daten beinhalten können, muss ein solcher Angriff verhindert werden. Dies ist anhand von Dummy-Operationen möglich. Die Schleife wird dann immer 15-mal durchlaufen. (Der maximale Wert eines einzelnen Blockes ist 15.) Für einige Operationen werden außerdem if-Bedingungen benötigt, welche abhängig von den Daten ausgeführt werden. Auch hier ist jedes Mal ein Dummy-else-Zweig notwendig, damit keine Information über die Daten gewonnen werden kann.

7 Effizienz

Die Effizienz der einzelnen Operationen kann als Anzahl der benötigten Table-Lookups betrachtet werden. Ein Table-Lookup ist immer gleich aufwändig, egal welche Operation durchgeführt wird. Am effizientesten sind die Logischen Bitoperationen, da hier keine Überträge berücksichtigt werden müssen. Aber auch Addition und Subtraktion liegen in $O(k)$ (k ist die Anzahl der Blöcke). Die Addition benötigt hierbei etwa $3k$ Operationen (es müssen pro Block zwei Überträge berücksichtigt werden). Die Subtraktion wäre theoretisch mit derselben Anzahl Lookups möglich. Beim Prototyp wurde aber eine etwas aufwändigere Variante implementiert, bei der die Subtraktion als Addition der negativen Zahl durchgeführt wird. Dadurch kommen ein paar mehr Lookups zustande, die für die Negierung der Zahl benötigt werden. Dies spart dafür Platz beim Evaluierungsschlüssel, da keine Subtraktionstabelle benötigt wird.

Die Multiplikation und die Division liegen in $O(k^2)$. Wird ein ausreichend langer Schlüssel gewählt, um Sicherheit zu gewährleisten kann die Berechnung vieler Multiplikationen und Divisionen daher doch längere Zeit in Anspruch nehmen. Hier sind aber sicher noch Effizienzoptimierungen einerseits beim Algorithmus, andererseits durch teilweise Implementierung direkt in Hardware innerhalb des HSM möglich.

8 Schlussfolgerung und Ausblick

Wie sich während des Erstellens der Arbeit herausgestellt hat, gab es zunächst eine Vielzahl an Angriffsmöglichkeiten, um Informationen über die Daten während einer Berechnung auf einer Blinden Turingmaschine zu gewinnen. Ein Angreifer mit Evaluierungsschlüssel kann anhand gezielter Manipulation von Code und Daten ein Chiffrat für jeden möglichen Wert erstellen. Gemeinsam mit der Vergleichsoperation könnte er somit jeden Wert "entschlüsseln" und damit das System kompromittieren. Um dem entgegenzuwirken, wurde ein Hardware-Security-Modul eingeführt, welches Vergleiche sowie sicherheitskritische Operationen durchführt. Daher bekommt auch der Cloud-Provider (oder eine andere Ausführungsumgebung) keinen Evaluierungsschlüssel. Mit Hilfe des HSMs kann er zwar immer noch ein Chiffrat aller Werte erzeugen, ohne Vergleichsoperation bringt ihm dies aber nicht viel. Einen Informationsgewinn hätte er nur für den unwahrscheinlichen Fall, dass er genau dasselbe Chiffrat nochmal erhält.

Bisher ist die Blinde Turingmaschine nur als Prototyp implementiert. Weitere Optimierungen, sowohl die Sicherheit als auch die Effizienz betreffend, sind möglich. Führt man weitere Sicherheitsanalysen durch, kann man eventuell manche Algorithmen wieder aus dem HSM auslagern. Die Vergleichsoperation selber jedoch muss im HSM verbleiben, da Vergleiche zu viel Angriffsfläche bieten.

Literatur

[Damg92] I. Damgård: Towards Practical Public Key Systems Secure Against Chosen Ciphertext Attacks. *In: Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, Springer-Verlag, London, UK, UK (1992), 445–456, .

[DuMP14] R. Dutra, B. Mehne, J. Patel: BlindTM–A Turing Machine System for Secure Function Evaluation. https://www.cs.berkeley.edu/~kubitron/courses/cs262a-F14/projects/reports/project9_report_ver2.pdf (2014).

[Rass13] S. Rass: Blind Turing-Machines: Arbitrary Private Computations from Group Homomorphic Encryption. In: *CoRR*, abs/1312.3146 (2013).

[Tang12] Q. Tang: Public Key Encryption Supporting Plaintext Equality Test and User-specified Authorization. In: *Sec. and Commun. Netw.*, 5, 12 (2012), 1351–1362.