

Kryptologie Labor
Kompensationsarbeit

Simone Fink

28. Juni 2018

Inhaltsverzeichnis

1	Vorbereitung	2
2	Kompensationsaufgaben	3
1.	LB-KA 00	3
2.	LB-KA 01	5
3.	LB-KA 02	9

Ausgangssituation: Sie haben ein HTML-Dokument mit folgendem Inhalt vorliegen:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Signierter Vertrag</title>
  </head>
  <body>
    Ich, Alice, überweise Bob den Betrag von 100 Euro.
  </body>
</html>
```

Dieses Dokument wurde von Alice digital signiert.

1. LB-KA 00**a)**

Erstellen Sie eine HTML-Datei nach obigem Muster.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Signierter Vertrag</title>
  </head>
  <body>
    Ich, Alice, überweise Bob den Betrag von 100 Euro.
  </body>
</html>
```

Abbildung 2.1: die erstellte HTML-Datei mit dem Namen "ueberweisung.html"

b)

Berechnen Sie den SHA-256 Hash dieser Datei und verwenden Sie Ihr Programm aus LB-HVS 01 zum Signieren des Hashes. Prüfen Sie anschließend die Integrität der Datei, indem Sie die Signatur verifizieren.

Der SHA-256 Hash dieser Datei ist

5064b36096864eb0ab6df830b8b35e7ada2d20a46c95d423bfb191aa10db8ebb

```

simsca@Brynhildr:~/workspace/krypto_kompis ./test Sign '<!DOCTYPE html> <html> <head> <meta charset="UTF-8"> <title>Signierter Vertrag</title>
</head> <body> Ich, Alice, überweise Bob den Betrag von 100 Euro. </body> </html>' 0874181488339315458975731838641583974570962586427257458561922
010608119985399634661178168188823581937331818452143867349966558834909721886236834987718405827972510247129733853251563198989596286966618530727850
433963884387849875748802660101412636872879480097823613816196434996739734680932118226234906469137784601418132663399394267888404089724446225659815
475154552805144880743566884573146365250189837817826358439076768644259820289819917630819483681960079840812101026145402621981375781686664841588816
1392644947306730349722597757588806163742268316244225461142832585510328065602385652254147108751268735457864401828869091 2227170542791077233487768
1575019141239360195315023327731872785036553377327979866044243199497323528404368833049911941802832106863948559802495321282883008184358992335728090
241285890650336285197555442679460021182578675760887828774560975370517966853389142702535325065938692832137902718541776682952799038683854356840926
1830619560293845869516394413106963497522655443630284033811308247866408892760647824350116195544105226660061642046609938201365180748077411293737661
7734586371438278013241438847012572310186295364466203780870693111359678214315882481986344724840584694166335743896271129684073568128960223634580483
449510308341
8934933253917386280372183073966448026132890635337370790824596002131016883159455890298318832253170317518576095029941977394818420296387872298457374
3605864480210555519107393536189863230411132527117793034649979313367860752609344188486547962972717904771456178268972973265078146849771117984767714
2362787551376357406849818120426229874864127264611974097143817482628130239306922587194860221833212024918045396324231116850809904723075936643564492
4222917356677104610483574489466994887634452952494213768166733340571376002842629364833122981225862837078288018909487933451961016506572075286529667
59849972030886982626576913149891727
simsca@Brynhildr:~/workspace/krypto_kompis ./test Verify
Usage: ./test Verify <message> <signature> <e> <n>
simsca@Brynhildr:~/workspace/krypto_kompis ./test Verify '<!DOCTYPE html> <html> <head> <meta charset="UTF-8"> <title>Signierter Vertrag</title>
</head> <body> Ich, Alice, überweise Bob den Betrag von 100 Euro. </body> </html>' 8934933253917386280372183073966448026132890635337370790824596
002131016883159455890298318832253170317518576095029941977394818420296387872298457374360586448021055551910739353618986323041113252711779303464997
9313367860752609344188486547962972717904771456178268972973265078146849771117984767714236278755137635740684981812042622987486412726461197409714381
7482628130239306922587194860221833212024918045396324231116850809904723075936643564492422291735667710461048357448946699488763445295249421376816673
334057137600284262936483312298122586283707828801890948793345196101650657207528652966759849972030886982626576913149891727 11 222717054279107723348
7768157501914123936019531502332773187278503655337732797986604424319949732352840436883304991194180283210686394855980249532128288300818435899233572
809024128589065033628519755544267946002118257867576088782877456097537051796685338914270253532506593869283213790271854177668295279903868385435684
0926183061956029384586951639441310696349752265544363028403381130824786640889276064782435011619554410522666006164204660993820136518074807741129373
7601773458637143827801324143884701257231018629536446620378087069311135967821431588248198634472484058469416633574389627112968407356812896022363458
0483449510308341
Signature valid.
simsca@Brynhildr:~/workspace/krypto_kompis

```

Abbildung 2.2: Signierung und Verifizierung von der HTML- Datei

Wie in Abbildung 2.2 zu sehen ist, ist die Signatur nun gültig. Signierung und Verifizierung wurden über die Linux Bash Shell ausgeführt. Die Programme zur Signierung und Verifizierung wurden mit Hilfe von Clemens J. Zuzan erstellt.

c)

Verändern Sie den Inhalt der Datei dahingehend, dass Alice den Betrag von 200 Euro an Bob überweist. Prüfen Sie nun erneut die Integrität der Datei. Ist die Signatur noch gültig?

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Signierter Vertrag</title>
  </head>
  <body>
    Ich, Alice, überweise Bob den Betrag von 200 Euro.
  </body>
</html>

```

Abbildung 2.3: die veränderte HTML-Datei mit dem Namen "ueberweisung.html"

```

simsca@Brynhildr:~/workspace/krypto_kompis ./test Verify '<!DOCTYPE html> <html> <head> <meta charset="UTF-8"> <title>Signierter Vertrag</title>
> </head> <body> Ich, Alice, überweise Bob den Betrag von 200 Euro. </body> </html>' 8934933253917386280372183073966448026132890635337370790824596
002131016883159455890298318832253170317518576095029941977394818420296387872298457374360586448021055551910739353618986323041113252711779303464997
9313367860752609344188486547962972717904771456178268972973265078146849771117984767714236278755137635740684981812042622987486412726461197409714381
7482628130239306922587194860221833212024918045396324231116850809904723075936643564492422291735667710461048357448946699488763445295249421376816673
334057137600284262936483312298122586283707828801890948793345196101650657207528652966759849972030886982626576913149891727 11 222717054279107723348
7768157501914123936019531502332773187278503655337732797986604424319949732352840436883304991194180283210686394855980249532128288300818435899233572
809024128589065033628519755544267946002118257867576088782877456097537051796685338914270253532506593869283213790271854177668295279903868385435684
0926183061956029384586951639441310696349752265544363028403381130824786640889276064782435011619554410522666006164204660993820136518074807741129373
7601773458637143827801324143884701257231018629536446620378087069311135967821431588248198634472484058469416633574389627112968407356812896022363458
0483449510308341
Signature invalid.
simsca@Brynhildr:~/workspace/krypto_kompis

```

Abbildung 2.4: Verifizierung des Hashes der Originaldatei mit der gefälschten Datei

Wie in Abbildung 2.4 sichtbar ist, ist die Signatur nicht mehr gültig. Die Verifizierung wurde wieder über die Linux Bash Shell ausgeführt und das Programm mit Hilfe von Clemens J. Zuzan geschrieben.

2. LB-KA 01

Im Originaldokument überweist Alice an Bob den Betrag von 100 Euro. Es soll ein "gefälschtes" Dokument erzeugt werden, in welchem Alice den Betrag von 200 Euro an Bob überweist. Dieses falsche Dokument soll jedoch die exakt gleiche Signatur wie das Originaldokument aufweisen (*preimage attack*). Versuchen Sie durch kleine, aber für den Betrachter nicht sichtbare Änderungen im Markup eine Kollision der Hashwerte und somit eine idente Signatur zu erreichen. Schreiben Sie dafür ein geeignetes Programm zum Modifizieren des Markups und zum Berechnen des Hash-Wertes. Das Programm soll abschließend eine Variante des Dokuments mit identem Hash ausgeben.

Wichtig: SHA-256 hat als Ausgabe 2^{256} mögliche verschiedene Werte. Damit sind im Mittel 2^{255} Versuche notwendig, um eine Kollision zu erzeugen. Verwenden Sie für die Aufgabe daher eine stark vereinfachte (und in der Praxis unsichere) Variante, bei der nur die ersten vier Hexadezimalziffern des Hash- Wertes verwendet werden. Dies reduziert die im Mittel notwendigen Versuche.

Händisches Ausprobieren

Wir haben: die ersten 4 Hexadezimalziffern des Hash- Wertes von der ersten originalen Datei: 5064

Wir suchen: die Datei, in der steht es werden 200 Euro überwiesen, mit den selben 4 Hexadezimalziffern des Hash- Wertes.

Nachdem der Datei **2203 Leerzeichen** am Ende eingefügt wurden, entsprechen die ersten 4 Hexadezimalziffern 5064.

Programmieren

Es wird ein C++ Programm geschrieben, bei dem der gefälschten Datei hinten dran immer wieder Leerzeichen angehängt werden bis die ersten 4 Hexadezimalziffern des Hashes den ersten 4 des Originaldokuments entsprechen.

```

1  /*
2   * LB_KA_01.cpp
3   *
4   * Created on: Jun 27, 2018

```

```

5  *      Author: Simone Fink (Hilfe von Clemens J. Zuzan)
6  */
7
8
9  #include <string.h>
10 #include <iomanip>
11 #include <sodium.h>
12 #include <sstream>
13
14
15 #include <iostream>
16 #include <gmp.h>
17 #include <gmpxx.h>
18 #include <fstream>
19 using namespace std;
20
21 //Erstellen eines Programms welches ein File entgegennimmt, dieses
    veraendert
22 //und dann solange Leerzeichen hintendran haengt, bis die ersten 4
    Ziffern des Hashes
23 //den ersten 4 Ziffern des Originalhashes entsprechen
24
25
26 //Mit Hilfe von Clemens J. Zuzan
27 void libsodium_to_GMP(const unsigned char (&libsodium_value) [
    crypto_hash_sha256_BYTES], mpz_class &GMP_value)
28 {
29     stringstream s;
30     s << hex;
31     for (size_t i = 0; i < sizeof libsodium_value; i++)
32         s << setw(2) << setfill('0') << (int)libsodium_value
            [i];
33     const char * const string_as_hex = s.str().c_str();
34     mpz_set_str(GMP_value.get_mpz_t(), string_as_hex , 16);
35 }
36
37 //Mit Hilfe von Clemens J. Zuzan
38 string hashwert(const string &message){
39     unsigned char hash[crypto_hash_sha256_BYTES];
40
41     const size_t message_length = message.length();
42     crypto_hash_sha256(hash, (const unsigned char*)message.c_str
        (), message_length);

```

```

43
44     mpz_class hash_mpz;
45
46     libsodium_to_GMP(hash, hash_mpz);
47     return hash_mpz.get_str(4); // es werden nur die ersten 4
        Ziffern benoetigt
48 }
49
50 int faelschen(string origs, string fakes)
51 {
52     string message_orig;
53     string message_fake;
54     char c;
55
56     ifstream orig;
57     orig.open(origs.c_str());
58     if(!orig)
59     {
60         cerr << "Fehler beim oeffnen der Originaldatei!" <<
            endl;
61         return -1;
62     }
63
64     ifstream fake;
65     fake.open(fakes.c_str());
66     if(!fake)
67     {
68         cerr << "Fehler beim oeffnen der gefaelschten Datei
            !" << endl;
69     }
70     while(orig >> c)
71     {
72         message_orig += c;
73     }
74
75     while(fake >> c)
76     {
77         message_fake += c;
78     }
79
80     //hashes der beiden dateien werden nun ausgerechnet
81
82     string hash_orig = hashwert(message_orig);

```



```

83     string hash_fake = hashwert(message_fake);
84
85     //das Inputfile wird nun nicht mehr benoetigt
86     orig.close();
87
88     //nun werden die hashes miteinander verglichen. wenn sie
89     //nicht gleich sind
90     //wird ein Leerzeichen ans outputfile angehaengt.
91
92     ofstream outputfile("Orig_Fake.html");
93     outputfile << message_fake;
94     char leer = ' ';
95     int leercnt;
96     while(hash_orig.compare(hash_fake) != 0)
97     {
98         outputfile << leer;
99         message_fake += leer;
100        leercnt++;
101        hash_fake = hashwert((const string)message_fake);
102    }
103    fake.close();
104
105    return leercnt;
106 }
107
108 int main(int argc, char** argv)
109 {
110     string orig(argv[1]);
111     string fake(argv[2]);
112     int leercnt = faelschen(orig, fake);
113     char c;
114     ifstream outputfile("Orig_Fake.html");
115     while(outputfile >> c)
116     {
117         cout << c;
118     }
119     outputfile.close();
120     cout << "\n\n leerzeichen die hinzugefuegt wurden: " <<
121         leercnt << endl;
122 }
123 }

```

Dieses Programm braucht sehr lange bis ein Ergebnis herauskommt. Ein Beispiel, wie man das Programm beschleunigen könnte, wäre die Realisierung von Threads.

3. LB-KA 02

Versuchen Sie nun zwei Dokumente zu erzeugen, ein "Originaldokument"(Alice an Bob 100 Euro) und ein "gefälschtes Dokument"(Alice an Bob 200 Euro), die beide den gleichen Hashwert haben (*second preimage attack*). Gehen Sie dabei analog zur Aufgabe LB-KA 02 vor und erzeugen Sie durch kleine, aber für den Betrachter nicht sichtbare Änderungen im Markup beider Dokumente die Kollision. Überlegen Sie zunächst, worin der Unterschied zum obigen Angriff liegt und wie sich dieser auf die Anzahl der im Mittel notwendigen Versuche auswirkt. Schreiben Sie dafür wiederum ein geeignetes Programm zum Modifizieren des Markups beider Dokumente und zum Berechnen der Hash-Werte. Das Programm soll abschließend zwei Varianten der Dokumente mit identem Hash ausgeben.

Überlegung: Die Originaldatei wird so lange verändert, bis eine Hashkollision entsteht. Danach wird die gefälschte Datei so lange verändert, bis eine Hashkollision zum kollidierenden Hash der Originaldatei entsteht.

```

1  /*
2   * LB_KA_02.cpp
3   *
4   * Created on: Jun 27, 2018
5   * Author: Simone Fink (Hilfe von Clemens J. Zuzan)
6   */
7
8
9  #include <string.h>
10 #include <iomanip>
11 #include <sodium.h>
12 #include <sstream>
13
14
15 #include <iostream>
16 #include <gmp.h>
17 #include <gmpxx.h>
18 #include <fstream>
19 using namespace std;
20
21 // Erstellen eines Programms welches zwei Files entgegennimmt
22 // und dann bei beiden solange Leerzeichen hintendran haengt, bis die
   ersten 4 Ziffern des Hashes
23 // den ersten 4 Ziffern des Originalhashes entsprechen

```

```

24
25
26 //Mit Hilfe von Clemens J. Zuzan
27 void libsodium_to_GMP(const unsigned char (&libsodium_value) [
    crypto_hash_sha256_BYTES], mpz_class &GMP_value)
28 {
29     stringstream s;
30     s << hex;
31     for (size_t i = 0; i < sizeof libsodium_value; i++)
32         s << setw(2) << setfill('0') << (int)libsodium_value
            [i];
33     const char * const string_as_hex = s.str().c_str();
34     mpz_set_str(GMP_value.get_mpz_t(), string_as_hex, 16);
35 }
36
37 //Mit Hilfe von Clemens J. Zuzan
38 string hashwert(const string &message){
39     unsigned char hash[crypto_hash_sha256_BYTES];
40
41     const size_t message_length = message.length();
42     crypto_hash_sha256(hash, (const unsigned char*)message.c_str
        (), message_length);
43
44     mpz_class hash_mpz;
45
46     libsodium_to_GMP(hash, hash_mpz);
47     return hash_mpz.get_str(4); //es werden nur die ersten 4
        Ziffern benoetigt
48 }
49
50 int faelschen(string origs, string fakes)
51 {
52     string message_orig;
53     string message_fake;
54     char c;
55
56     ifstream orig;
57     orig.open(origs.c_str());
58     if(!orig)
59     {
60         cerr << "Fehler beim oeffnen der Originaldatei!" <<
            endl;
61         return -1;

```

```

62     }
63
64     ifstream fake;
65     fake.open(fakes.c_str());
66     if (!fake)
67     {
68         cerr << "Fehler beim oeffnen der gefaelschten Datei
69             !" << endl;
70     }
71     while(orig >> c)
72     {
73         message_orig += c;
74     }
75     while(fake >> c)
76     {
77         message_fake += c;
78     }
79
80     //hashes der beiden dateien werden nun ausgerechnet
81
82     string hash_orig = hashwert(message_orig);
83     string hash_fake = hashwert(message_fake);
84
85     //nun werden die hashes miteinander verglichen. wenn sie
86     //nicht gleich sind
87     //wird ein Leerzeichen ans outputfile angehaengt.
88
89     char leer = ' ';
90     int leercnt1 = 0;
91     string hash_orig_neu = hashwert(message_orig);
92     do{
93         orig << leer;
94         message_orig += leer;
95         leercnt1++;
96         hash_orig_neu = hashwert(message_orig);
97     }
98     while(hash_orig.compare(hash_orig_neu) != 0);
99
100     int leercnt = leercnt1;
101
102     int leercnt2 = 0;
103     while(hash_orig.compare(hash_fake) != 0)

```

```

103     {
104         fake << leer;
105         message_fake += leer;
106         leercnt2++;
107         hash_fake = hashwert(message_fake);
108     }
109
110     leercnt += leercnt2;
111
112     fake.close();
113     orig.close();
114
115     return leercnt;
116
117 }
118
119
120
121 int main(int argc, char** argv)
122 {
123     string orig(argv[1]);
124     string fake(argv[2]);
125     int leercnt = faelschen(orig, fake);
126     cout << "\n\n leerzeichen die insgesamt in beiden Files
        hinzugefuegt wurden: " << leercnt << endl;
127 }

```

Hierbei stößt man auf das Geburtstagsparadoxon. Es ist um einiges schwerer eine Hash-kollision von der Originaldatei zu erlangen, als bei der gefälschten Datei. Deswegen wird die Ausführung dieses Programmes noch länger dauern als bei LB_KA_01.