

Vorlesung Nr. 3

Kryptologie II - Datum: 08.10.2018

- Digitales Geld (Teil 2)
- Mathematics in Python (Part 1)

Altes Aufgabenblatt

- **Aufgabe 1.1**

Bilden Sie Arbeitsgruppen zu je 2-4 Teilnehmern. Denken Sie sich einen Namen (Pseudonym) aus für Ihre Gruppe.

- **Aufgabe 1.2**

Vervollständigen Sie Ihr Profil.

Optional: Verlinken Sie sich untereinander auf LinkedIn. Erstellen Sie dort ein Profil, falls Sie noch keines haben.

- **Aufgabe 1.3**

Finden und nennen Sie mindestens zehn Beispiele aus Ihrem eigenen Umfeld, wo Sie mit Cryptocurrencies bezahlen könnten.

- **Aufgabe 1.4**

Diskutieren Sie innerhalb Ihrer neu gebildeten Arbeitsgruppe, mit welchen digitalen Hilfsmitteln Sie die praktischen Aufgaben gemeinsam im Team, verteilt über das Internet, bearbeiten wollen. Beispielsweise Online-Plattformen für Task Management, Source Code Repositories, Diskussionsplattform, Python Code Bezeichnungen, etc. etc. Schreiben Sie auch auf, welche Editoren die einzelnen Gruppenmitglieder voraussichtlich verwenden werden.

Dokumentieren Sie das Ergebnis.

Buch der Woche

Titel: Introduction to Modern Cryptography (2nd Ed.)

Autor: Jonathan Katz, Yehuda Lindell

Verlag: CRC Press

Umfang: ca. 580 Seiten

Hinweise zum Inhalt:

- häufig empfohlenes Standardwerk
- sehr umfangreich, symmetrische und asymmetrische Verfahren etwa gleich gewichtet
- Darstellung teilweise sehr mathematisch und formal, die vielen Beweise sind manchmal recht knapp und für Einsteiger nicht immer sofort verständlich
- zur Begleitung der Vorlesung nur bedingt empfehlenswert, für spätere Vertiefung über die Vorlesung hinaus allerdings gut geeignet

Double Spending Problem

- Unser vorheriges Protokoll hat einen Schönheitsfehler. Angenommen, die Bank speichere alle Seriennummern bereits eingelöster Geldeinheiten. Dies erlaubt die Aufdeckung von Betrugsversuchen für den Fall, dass dieselbe Geldeinheit mehrmals zur Auszahlung bei der Bank eingereicht wird.
- Tritt dieser Fall ein, so läßt sich jedoch nicht feststellen, welcher der Einreicher der rechtmäßige Besitzer war. Wir können den Schuldigen also nicht nachträglich identifizieren. Ein solches Protokoll wäre nicht praxistauglich.
- Zum Glück gibt es verschiedene Methoden, auch dieses Problem zu lösen. Eines davon schauen wir uns im folgenden genauer an.
- Später betrachten wir dann anhand von Bitcoin einen ganz anderen Lösungsansatz, mit dessen Hilfe wir das Double Spending Problem ebenfalls lösen können.
- Im folgenden versuchen wir das genannte Problem zu beheben durch Integration eines einfachen Secret Sharing Mechanismus, in Kombination mit einem zusätzlichen Überprüfungsschritt durch den jeweiligen Händler bei Einlösung der Geldeinheit durch einen Kunden. Das komplette, derart modifizierte Protokoll sieht folgendermaßen aus.

Secret Sharing mittels XOR

Im Folgenden benötigen wir für unser Protokoll noch ein **Secret Sharing** Scheme. Beim Secret Sharing wird ein Geheimnis auf zwei oder mehr Personen aufgeteilt, so dass eine Gruppe aus mehreren Personen kooperieren muss, um das Geheimnis zu entschlüsseln.

Secret Sharing Verfahren erlauben, dass k von m Personen benötigt werden, um das Geheimnis zu entschlüsseln, wobei $2 \leq k \leq m$.

Für unseren Bedarf genügt der Spezialfall $k = m = 2$.

Sei $S \in \{0, 1\}^n$ unser Geheimnis, kodiert als Bitvektor der Länge n .

Wir wählen nun ein (pseudo-)zufälliges $T \in \{0, 1\}^n$, dann erhalten die beiden am Protokoll beteiligten Personen A und B die folgenden Werte:

A erhält T

B erhält $T' = S \oplus T$

Beide Werte für sich genommen sind nutzlos, um das Geheimnis S zu rekonstruieren. Tun sich A und B jedoch zusammen, so erhalten sie auf einfache Weise

$$T \oplus T' = T \oplus (S \oplus T) = S$$

Anmerkung: dieses XOR-Verfahren lässt sich auch für größere Werte k und m (wobei $k = m$) in einfacher Weise verallgemeinern.

Tracebares Anonymes Cash Protokoll (1)

- Der Kunde fordert eine Einheit anonymes elektronisches Geld über einen bestimmten Geldbetrag von seiner Bank. Wieder erzeugt er k Beispiel-Einheiten; diese enthalten den Namen der auszahlenden Bank, den Geldbetrag, die zugehörige Währung etc. Jede der obigen Beispiel-Einheiten ist versehen mit einer zufällig gewählten, individuellen Seriennummer, deren Länge es praktisch nahezu ausschließt, daß jemals zwei dieser Seriennummern gleich sind; eine mögliche Länge dieser Seriennummer wäre 128 Bit.
- Sei ID ein binärer String, der die Identität des Kunden beschreibt; diese werde nun aufgeteilt in j verschiedene Paare so, daß
$$ID = ID_1 \text{ XOR } ID_1' = \dots = ID_j \text{ XOR } ID_j'.$$
- Jeder, der ein komplettes Paar bestehend aus ID_i und ID_i' besitzt, kennt die Identität des Kunden.
- Für jede der k Geldeinheiten verfährt der Kunde nun wie folgt. Er versiegelt jeden der insgesamt $2j$ ID Strings mit Hilfe einer ("Bit Commitment") Verschlüsselung, unter Verwendung eines geheimen Schlüssels $Key_{(a, b)}$: Betrachten wir beispielsweise Geldeinheit m , so enthält diese nun zusätzlich zu den bisherigen Daten noch die beiden verschlüsselten Werte
$$\text{Encrypt}_{Key(2i, m)}(ID_i \mid S), \text{Encrypt}_{Key(2i+1, m)}(ID_i' \mid S),$$
wobei S lediglich ein zur erhöhten Sicherheit angefügter Datenstring ist (siehe oben, Bit Commitment Schemes).

Tracebares Anonymes Cash Protokoll (2)

- Der Kunde versieht die obigen k Datensätze mit verschiedenen blinding factors b_i , und übergibt die so präparierten Datensätze an seine Bank.
- Die Bank wählt $k - 1$ dieser Datensätze und verlangt die zu diesen gehörigen $k - 1$ Blinding Factors, sowie die Schlüssel zum Entschlüsseln der paarweise zusammengehörigen ID-Strings.
- Der Kunde übergibt die geforderten Informationen zum kompletten Auswerten jener $k - 1$ Datensätze an seine Bank.
- Die Bank überprüft, ob alle resultierenden Datensätze die vereinbarte Form und den vom Kunden genannten Geldbetrag beinhalten. Ist dies nicht der Fall, so wird ein Verfahren wegen versuchten Betruges eingeleitet.
- Die Bank signiert die noch verbleibende, nicht aufgedeckte Geldeinheit mit ihrem Secret Key und sendet das Resultat zum Kunden.
- Der Kunde entfernt nun noch den Blinding Factor und verfügt nunmehr über eine gültig signierte elektronische Geldeinheit.

Tracebares Anonymes Cash Protokoll (3)

Um die Vorteile des Protokolls nutzen zu können, sind noch folgende Schritte erforderlich:

- Präsentiert ein Kunde beim Händler zum Einkaufen seine elektronische Geldeinheit, so erzeugt der Händler zunächst einen (pseudo-) zufälligen Bitvektor der Länge j .
- Für jedes i , $0 \leq i \leq j$, offenbart der Kunde dem Händler durch Präsentation des zugehörigen Schlüssels entweder ID_i (falls i -tes Bit = 0) oder ID_i' (falls i -tes Bit = 1). Derart verfügt der Händler nun über j verschiedene ID-Hälften, doch kein Paar zusammengehöriger Hälften, so daß der Händler keinerlei Information über die Identität seines Kunden erhält.
- Der Händler kann überprüfen, daß er tatsächlich die korrekten Schlüssel (und folglich korrekte ID-Hälften) vom Kunden bereitgestellt bekam, da diese Schlüssel mit einem Bit Commitment Scheme versiegelt waren. Erscheint stets wieder der angefügte Sicherheitsstring S im Klartext, so war die vom Kunden gelieferte Information korrekt.
- Die Gültigkeit der eigentlichen elektronischen Geldeinheit kann der Händler anhand der digitalen Signatur der Bank verifizieren.
- Der Händler reicht die erhaltene Geldeinheit bei der zugehörigen Bank ein, zusammen mit dem von ihm gewählten j -Bit String und den j korrespondierenden ID-Hälften.

Tracebares Anonymes Cash Protokoll (4)

Geschieht es nun, daß bei der Bank dieselbe elektronische Geldeinheit mehr als einmal eingereicht wird, so verfährt die Bank zur Überführung des Missetäters wie folgt.

- Hat der Kunde bei mehreren Händlern mit derselben Geldeinheit zu zahlen versucht, so werden diese nahezu mit Sicherheit unterschiedliche Zufalls-Bitvektoren der Länge j erzeugt haben; in diesem Falle ist die Bank in der Lage, zwei zueinander passende ID-Hälften ID_i und ID_i' miteinander zu kombinieren und die Identität des betrügerischen Kunden ausfindig zu machen.
- Erhält die Bank indessen zweimal denselben Bitvektor überreicht, so ist davon auszugehen, daß der einreichende Händler den Betrug begangen hat. Wie wir dem Protokoll somit entnehmen, ist eine solcherart generierte elektronische Geldeinheit nicht transferierbar von einem Besitzer auf einen oder mehrere weitere, bevor diese eingelöst wird. Dafür lassen sich Betrüger gegebenenfalls leichter identifizieren.

Mathematics in Python

Today's Schedule

- Motivation

In Cryptography, we have specific demands with respect to large integer calculations, efficient bit manipulation, and the optimization of certain time-consuming algorithms. Thus, we do not only need to know about the existence of different types like integers, floating-points etc. It's important to understand the limitations, advantages and disadvantages of different types of operations and operators in Python. Otherwise, we can get unexpected errors and/or waste computing time.

- Content

- Types of operators in Python
- Mathematical operators
- Floating-point arithmetic and rounding issues
- Large integers and decimal fixed point
- Modular arithmetic in Python

Neues Aufgabenblatt

- **Aufgabe 2.1**

Do some Google research: which are – in your opinion - the three most important cryptographic libraries for Python? Create a table and describe the most important features / functionality for each library. What are their respective advantages and disadvantages?

- **Aufgabe 2.2**

Which library provides – in your opinion – the best support for the computation of hash functions? Why? Explain! What do you miss?

- **Aufgabe 2.3**

Which library provides – in your opinion - the best support for the computation of symmetric encryption? Why? Explain! What do you miss?

- **Aufgabe 2.4**

Which library provides – in your opinion – the best support for the computation of public key protocols? Why? Explain! What do you miss?

- **Aufgabe 2.5**

The % operator also allows both operands to be floating-points. For $x, y \in \mathbb{R}$, look into Python's documentation and describe the return value of $x \% y$

Deadline für die Abgabe: Sonntag, 14.10.2018, 23:59 Uhr

Our Approach to Python and Crypto

Python includes a few cryptographic functions out of the box. In addition to that, there are several third party libraries for public key algorithms, hash functions, and symmetric encryption.

We could use these libraries, and we probably will do so later. First of all, however, we are interested in learning more about Python's operators and data types. This way, we can better understand the inner workings and we can learn how to write our own functions and libraries as we need them.

One of the golden rules of software development is:

“Don't build your own crypto!”. Instead, just use well-known and tested libraries.

However, somebody needs to be able to write such crypto libraries in the first place. And someone has to be able to check those libraries for efficiency and correctness. Clever experts like you who studied IT security, for example 😊

Python Crypto Libraries

Question:

What Python operators / functions / number types / libraries did you use so far?

Operators in Python

Types of Operators in Python

- Arithmetic/Mathematical Operators
- Logical Operators
- Bit Manipulating Operators
- Comparison (Relational) Operators
- Assignment Operators
- Membership Operators
- Identity Operators

➔ We will focus on Mathematics-related operators today!

Order of Precedence (1)

Suppose we have a longer arithmetic expression involving several different operators. Then it's important to know the exact order of the resulting evaluation/calculation of subexpressions.

Of course we can always use parentheses around subexpressions to define a certain order of precedence. If we don't use parentheses, the following default behaviour applies:

Precedence

Highest



Lowest

| Operator | Operation(s) |
|-------------|---|
| ** | Exponentiation |
| - | Negation |
| *, /, %, // | Multiplication, Division, Remainder, Integer Division |
| +, - | Addition, Subtraction |

Operators of equal precedence are applied left to right. Only exponentiation is applied right to left.

Order of Precedence (2)

Here, we include additional operators, in order of precedence (from high to low).

| Operator |
|-------------------------|
| ** |
| ~ + - |
| * / % // |
| + - |
| << >> |
| & |
| ^ |
| <= < > >= |
| == <> != |
| = %= /= //=-= += *= **= |
| is is not |
| in not in |
| not or and |

Order of Precedence: Examples

Please check the manual and answer for yourself:

How does Python interpret...

```
>>> 2**3**4
```

is this 8^{**4} or is it 2^{**81} ???

How about...

```
>>> 2/3*4/5
```

How about...

```
>>> 2**-2/2
```

Quick Task for volunteers: invent a tricky example involving three or four operators and post it to this week's ILIAS forum. Try to solve and answer challenges posted by others.

Mathematical Operators in Python

Floating-Point Arithmetic

- Normally, if we do public key cryptography, we deal with large integers.
- Some algorithms, however, use floating-point arithmetic.
- Besides that, floating-point calculations can be much faster in some environments.
- Moreover, with some operations in Python, you may inadvertently get return values in floating-point type instead of integer.
- Therefore, we cannot completely ignore floating-point arithmetic and related issues.
- Nevertheless we will primarily work with integers.

Rounding Errors (1)

First, let's have a look at some examples...

Example:

```
>>>> 2 / 3
0.6666666666666666
>>>> 5 / 3
1.6666666666666667
>>>> 1 + 2 / 3
1.6666666666666665
```

In the first example, the correct (exact) result would be $\frac{2}{3}$, and in the second and third example, the result would be $1\frac{2}{3}$. In any case, Python should print an infinite amount of 66666666... instead of only 16 digits for an exact representation, but floating-point calculations can only use finite precision and we have to accept rounding errors.

But even if we accept the fact that we only compute with finite precision, - why is there a difference in the 16th decimal place in the representation of $\frac{2}{3}$ versus $1\frac{2}{3}$ versus $\frac{5}{3}$?

Why do we get a 6 and a 7 and a 5 in the 16th decimal place ???

Rounding Errors (2)

As soon as we are dealing with floating-point calculations, we have to be aware of rounding errors. This is not Python-specific, even though individual programming languages behave differently with respect to floating-point arithmetic, precision and rounding.

Since all numbers are represented as binary strings, rational numbers like $\frac{1}{3}$ can only be approximated. On the other side, numbers like $\frac{1}{4}$ can exactly be represented as 0.25 .

Rounding errors can increase dramatically during some calculations and, if nobody is aware of this, could have devastating effects.

If, for example, we add very large and very small numbers, the effect could be the following:

```
>>> 10000000000 + 0.00000000001
10000000000.0
```

It seems that the addition never took place! In some scenarios, like banking, for example, this can result in lost money or worse.

As a rule of thumb, don't add numbers of very different size. Moreover, add them from smallest to largest to reduce rounding errors. If you need to make sure that your errors won't exceed a certain limit, then numerical analysis has answers for you.

Rounding Errors (3)

Rounding Issues cannot only influence the precision of calculations. If we try to compare two sides of an equation, we must also be aware of potential problems. Let's have a look at the following

Example:

```
>>> 0.1 + 0.1 == 0.2
```

```
True
```

```
>>> 0.1 + 0.1 + 0.1 == 0.3
```

```
False
```

```
>>> round(0.1 + 0.1 + 0.1, 1) == round(0.3, 1)
```

```
True
```

```
>>> round(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 1)
```

```
False
```

In this case, pre-rounding doesn't solve the problem, but post-rounding does.

Decimal Fixed Point

Python provides an alternative to “regular” floating-point numbers: the so-called ***Decimal Fixed Point***. This is better suited for high precision requirements like accounting applications, for example. The idea behind decimal fixed point is to avoid unexpected (at least for people who won’t read the manual...) rounding errors. We could try to use higher precision floating-point operations, but this doesn’t solve the underlying problem.

For example, the numbers 0.1 and 0.2 don’t have an exact binary floating-point representation, which yields the following:

Example:

```
>>> 0.1 + 0.2
0.30000000000000004
```

If we use decimal instead, we get the desired result:



```
>>> from decimal import *
>>> getcontext().prec = 6
>>> Decimal(0.1) + Decimal(0.2)
Decimal('0.300000')
>>>
```






More details can be found in the documentation of Python’s decimal module.

Remark: don’t confuse “Decimal Fixed Point” with “Decimal Floating Point”.

The SciPy Numerical Python Package


https://scipy.org








[Install](#)[Getting Started](#)[Documentation](#)[Report Bugs](#)[Blogs](#)


SciPy (pronounced "Sigh Pie") is a Python-based ecosystem of open-source software for mathematics, science, and engineering. In particular, these are some of the core packages:


**NumPy**
Base N-dimensional array package

**SciPy library**
Fundamental library for scientific computing

**Matplotlib**
Comprehensive 2D Plotting

**IPython**
Enhanced Interactive Console

**Sympy**
Symbolic mathematics

**pandas**
Data structures & analysis

[More information...](#)

News

NumPy 1.15.0 released
(2018-07-23)

[See Obtaining NumPy & SciPy libraries.](#)

SciPy 1.1.0 released
(2018-05-05)

[See Obtaining NumPy & SciPy libraries.](#)

SciPy 1.0.0 released
(2017-10-25)

[See Obtaining NumPy & SciPy libraries.](#)

EuroSciPy 2017
(2017-08-28)

The EuroSciPy meeting is a cross-disciplinary gathering focused on the use and development of the Python language in scientific research. The 2017 edition will take place in Erlangen, Germany, Aug 28-Sep 1.

SciPy 2017
(2017-07-10)

SciPy, the 16th annual Scientific Computing with Python conference, will be held July 10-16, 2017 in Austin, Texas.

SciPy 0.19.1 released
(2017-06-21)

[See Obtaining NumPy & SciPy libraries.](#)

SciPy 0.19.0 released
(2017-03-09)

[See Obtaining NumPy & SciPy libraries.](#)

[Past news...](#)

About SciPy

Getting Started

Documentation

Install

Bug Reports

Codes of Conduct

SciPy Conferences

Topical Software

Citing

Cookbook

Blogs

NumFOCUS

CORE PACKAGES:

Numpy

SciPy library

Matplotlib

IPython

Sympy

Pandas

Search

Go