Weihnachtsprojekt

Abgabe: 19.01.2026 **bis** 11:45 Uhr

Dieses Weihnachtsprojekt behandelt Anwendungs- und Implementierungsaspekte moderner symmetrischer Kryptographie. In den Vorlesungen 3 und 4 haben wir das Konzept der Stromchiffre kennengelernt. Hier wird ein pseudozufälliger Schlüsselstrom auf den Klartext XORiert, sodass die Sicherheit des Vefahrens an die Sicherheit des Schlüsselstroms gebunden ist. Bisher haben wir einfache LFSRs als potentielle Schlüsselstromgeneratoren eingeführt, welche jedoch durch Known-Plaintext Angriffe und mithilfe linearer Gleichungssysteme gebrochen werden können. Ziel in diesem Projekt ist, eine sichere Stromchiffre mit Praxisrelevanz kennenzulernen und zu implementieren.

Abgabemodalitäten und Bewertungskriterien

Mit den in diesem Projekt gesammelten Punkten können in den Hausaufgaben verloren gegangene Punkte ausgeglichen werden. Die Bearbeitung des Projekts ist nicht verpflichtend. Auch ohne die Bearbeitung des Projekts ist es möglich, alle Bonuspunkte zu erhalten, sofern alle Hausaufgaben vollständig und korrekt abgegeben wurden. Auch erhöht sich durch die Bearbeitung des Projekts nicht die Anzahl der maximal erreichbaren Bonuspunkte.
Das Weihnachtsprojekt wird in denselben Gruppen bearbeitet und abgegeben wie die Hausaufgaben. Für die volle Punktzahl müssen alle Aufgaben bearbeitet werden und der Code ohne Warnungen und Fehlermeldungen ausführbar sein. Für Abgaben, die diese Kriterien nicht erfüllen, werden Teilpunkte vergeben. Abschreiben führt dazu, dass das gesamte Projekte mit 0 Punkten bewertet wird. Bei der Bewertung von Programmierprojekten wie diesem setzen wir standardmäßig Software zur Plagiatsprüfung ein. Ihr könnt euch gerne gegenseitig beim Verständnis der Algorithmen helfen oder grundsätzliche Tipps zur Programmierung geben, solltet es aber dringend vermeiden, Programmcode untereinander (also außerhalb eurer Abgabegruppen) auszutauschen.
Wir stellen ein Code-Template in Python zur Bearbeitung der Programmieraufgaben bereit. Bitte nutzt für eure Abgaben *ausschließlich* dieses Template. Weitere Informationen und Hinweise zu Entwicklungsumgebungen und Python findet ihr am Ende dieser Aufgabenstellung.
Die Abgabe erfolgt in zwei Dateien (PDF mit Antworten zu den Aufgaben und bearbeitetes Python-Template).

1. Eine moderne Stromchiffre: ChaCha20 100 Punkte

Für die Realisierung von Stromchiffren haben wir bereits LFSRs kennengelernt. Eine Alternative ist die sog. Addition-Rotation-XOR-Konstruktion, die bspw. für die moderne Stromchiffre *ChaCha20* genutzt wird. ChaCha20 wurde 2008 von Daniel J. Bernstein vorgestellt[1] und ist eine besonders einfache Stromchiffre, die gut für schnelle Software-Implementierungen geeignet ist. ChaCha20 wurde im Juni 2018 von der Internet Engineering Task Force (IETF) standardisiert und ist inzwischen weit verbreitet. Insbesondere ist ChaCha20 neben AES der einzige zulässige Verschlüsselungsalgorithmus für TLS 1.3[2]. In dieser Aufgabe soll ChaCha20 anhand des IETF-Standards RFC 8439[3] in Python implementiert werden. Nutzen Sie hierzu das bereitgestellte Code-Template `chacha_template.py`. Sie können Ihren Code mit dem Skript `testbench.py` überprüfen.

a) Lesen Sie die Abschnitte 2.1 bis 2.4 aus der RFC 8439. Der Kern von ChaCha ist die sog. *Viertelrunde*. Diese simple Operation ist in Abschnitt 2.1. beschrieben. Zeichnen Sie ein Blockschaltbild, welches die Operation darstellt. (5 Punkte)

b) Die Viertelrunde benötigt eine zyklische Linksverschiebung der 32-Bit Eingabewerte. Implementieren Sie dies zunächst in der Funktion `cyclic_shift`. (5 Punkte)

c) Die Viertelrunde operiert jeweils auf vier von insgesamt 16 32-Bit Ganzzahlen des internen ChaCha-Zustands. Implementieren Sie die Viertelrunde in der Funktion `quarterround`. Diese soll die entsprechenden Positionen der Zustandsmatrix entgegennehmen. Sie können Ihren Code mit den Testwerten aus Abschnitt 2.1.1. der RFC 8439 überprüfen.
Hinweis: Beachten Sie, dass die benötigten Additionen modulo 2^{32} gerechnet werden müssen (Hinweis am Ende von Abschnitt 2.3 der RFC 8439). Dies lässt sich bspw. durch eine einfache Maskierung des Additionsergebnisses mit `0xffffffff` erreichen. (10 Punkte)

d) Die Eingabe des ChaCha-Algorithmus (gem. RFC 8439) besteht aus einem 256-Bit Schlüssel, einer 96-Bit Nonce (number used only once) sowie einem 32-Bit Zähler, welche den ChaCha-State initialisieren. Implementieren Sie die Funktion `init_chacha_state` gemäß RFC 8439, Abschnitt 2.3. Überprüfen Sie Ihre Implementierung mit den Testwörtern aus Abschnitt 2.3.2.
Hinweis: Beachten Sie die Little-Endian-Byte-Reihenfolge[4]. Sie können die Funktionen `struct.unpack()` und `to_bytes()` verwenden. (10 Punkte)

e) ChaCha20 verwendet 80 Viertelrunden (d.h. 20 „ganze“ Runden), wobei abwechselnd vier Spalten-Runden und vier Diagonal-Runden ausgeführt werden. Implementieren Sie die ChaCha-Block-Funktion in `generate_chacha_keystream` gemäß RFC 8439, Abschnitt 2.3.1. Überprüfen Sie Ihre Implementierung mit den Testwörtern aus Abschnitt 2.3.2. und 2.4.2.
(20 Punkte)

f) Komplettieren Sie unter Verwendung des vorherigen Teilergebnisses die Funktion `process_file`. Die Funktion soll eine verschlüsselte Quelldatei einlesen, entschlüsseln und in eine Zieldatei

[1] `https://cr.yp.to/chacha/chacha-20080128.pdf`, Daniel J. Bernstein hat in der Vergangenheit auch an der RUB gearbeitet.

[2] `https://datatracker.ietf.org/doc/html/rfc8446#appendix-B.4`

[3] `https://datatracker.ietf.org/doc/html/rfc8439`

[4] `https://en.wikipedia.org/wiki/Endianness#Example`

schreiben. Entschlüsseln Sie die Datei `encrypted.zip`. Folgende Parameter wurden zur Verschlüsselung genutzt.
Schlüssel:
`0xaf3d5cf9133ed833bd390ee187bb14f0da5aa23d4864c291a011b7bb031ac96d`
Nonce: `0x081e13f87bb610c2044c1665`
Initialer Blockzähler: `0`
Beschreiben Sie in einem Satz die Inhalte des verschlüsselten Archivs und geben Sie die Anzahl der enthaltenen Dateien an. (15 Punkte)

g) Komplettieren Sie die Funktion `bruteforce`, um die Schlüssel zu den mit ChaCha20 verschlüsselten Dateien im Verzeichnis `bruteforce_files` zu ermitteln. Insbesondere gibt der jeweilige Name der Dateien die Größe des verwendeten Schlüsselraums (in Bit) an, aus dem ein zufälliger Schlüssel zur Verschlüsselung gezogen wurde. Ein n-Bit Schlüssel besteht hier also aus $(256 - n)$ führenden Nullen, gefolgt von n zufälligen Schlüsselbits. Lesen Sie die Dateien ein und führen Sie den Brute-Force Angriff durch. Sie wissen, dass die Chiffrate jeweils mit der Zeichenkette `Weihnachtsprojekt WS2526` beginnen. Weiterhin ist bekannt, dass als Nonce `0x4853e7373f57a43af1c4f89d` verwendet wurde und der Blockzähler bei der Verschlüsselung mit `0` begonnen hat. Geben Sie für mindestens 20 Dateien ihrer Wahl den jeweils verwendeten Schlüssel an. (20 Punkte)

h) Ermitteln Sie mit Hilfe von `time.time()` die Zeit, die die Schlüsselsuche zu einer bestimmten Datei benötigt. Erweitern Sie die zuvor vervollständigte Funktion `bruteforce` wie folgt: Legen Sie zwei Listen an und speichern Sie in jeweils einer Liste die aktuelle Schlüsselraumgröße in Bit sowie die Dauer, um den Schlüssel zu finden. Nutzen Sie anschließend die Funktionen `plot_results` und `plot_results_log`, um die Dauer der Schlüsseluche als Funktion der Schlüsselraumgröße darzustellen. Fügen Sie beide Plots (`plot.png` und `plot_log.png`) Ihrer Abgabe hinzu. Beantworten Sie die folgenden Fragen:

Wieso werden hin und wieder Schlüssel trotz wachsender Schlüsselraumgröße schneller gefunden?

Welchen Vorteil hat die logarithmische Darstellung?

Geben Sie anhand Ihrerer Ergebnisse begründete Schätzungen ab, wie lange die Schlüsselsuche auf Ihrem System für die 32, 64, 128 und 256 Bit Verschlüsselung benötigen würde.
(15 Punkte)

Entwicklungsumgebung

Als Programmiersprache für die Implementierung soll ausschließlich `Python3` verwendet werden. Die Verwendung der entsprechend bereitgestellten Vorlagen ist verpflichtend, da die Korrektur bei der erwarteten Anzahl an Abgaben für uns ansonsten nicht zu bewerkstelligen ist. Die Vorlagen enthalten bereits die grobe Struktur der resultierenden Programme und geben (hoffentlich) hilfreiche Hinweise zur Implementierung der einzelnen Komponenten. Außerdem ist für die Programmierung in `Python3` (mindestens) die Version `3.7` vorgeschrieben. Beachten Sie bitte, das wir keinen Support zu verwendeten Entwicklungsumgebungen geben können. Wir empfehlen die Nutzung von *Visual Studio Code*[5] oder *PyCharm*[6], es kann allerdings auch jeder beliebige andere Texteditor verwendet werden.
Fehlende Python-Pakete können in der Python Konsole mit `pip` installiert werden. Für das Code-Template benötigen Sie das Paket `matplotlib`, welches wie folgt installiert wird:
`pip install matplotlib`

Ausführen des Codes

Navigieren Sie in den Ordner, in dem das von Ihnen bearbeitete Python Skript `chacha_template.py` liegt. Dort müssen auch alle benötigten zugehörigen Dateien liegen und Verzeichnisse: Die Datei `encrypted.zip` und das Verzeichnis `bruteforce_files`. Führen Sie anschließend das Skript in Ihrer Entwicklungsumgebung aus. Sollten Sie nur mit einem Texteditor arbeiten, navigieren Sie mit der Konsole in das Verzeichnis. Dafür steht der Konsolenbefehl `cd` zur Verfügung. Mit `cd [Ordnername]` wechseln Sie in den mit „Ordnername“ spezifizierten Ordner. Alle Inhalte im aktuellen Verzeichnis können mit `ls` angezeigt werden. Der Befehl `pwd` zeigt den aktuellen Ordnerpfad an. Mit `cd ..` wechseln Sie in das vorherige Verzeichnis. Im Verzeichnis, welches `chacha_template.py` enthält, können Sie das Skript durch folgenden Befehl ausführen: `python chacha_template.py`.
Zusätzlich zum Code-Template stellen wir ein spezielles Test-Skript bereit (`testbench.py`), das genutzt werden kann, um die korrekte Funktion Ihrer ChaCha20-Implementierung zu überprüfen.

[5] `https://code.visualstudio.com`
[6] Kostenlose Version oder Bildungslizenz, `https://www.jetbrains.com/community/education/#students`