بسم الله الرحمن الرحیم

گزارش کد الگوریتم رمز Serpent Plus

محمد مهدی سوری

عرفان محمدی

**Main.py**

در این فایل نحوه اجرای پروژه از طریق ترمینال مشخص می شود. برای اجرای برنامه چهار Mode در نظر گرفته ایم.

در Mode اول یک متن ساده را به صورت رشته ای دریافت کرده (با استفاده از پارامتر –p) و با استفاده از توابع encrypt\_plain\_text و decrypt\_cipher\_text متن رمز شده و متن رمزگشایی شده را به دست آورده و آن را نمایش می دهیم. البته می توان کلید را هم با پارامتر –k از حالت پیشفرض تغییر داد.

در Mode دوم یک متن ساده که به صورت باینری داده شده است (با استفاده از پارامتر -b) سپس با استفاده از توابع encrypt\_plain\_text\_binary و decrypt\_cipher\_text\_binary متن رمزشده و متن رمزگشایی شده به دست آورده می شوند. مانند حالت قبل می توان با پارامتر –k کلید را نیز تغییر داد.

در Mode سوم آدرس یک فایل ورودی که هر سطر آن از 40 حرف هگزادسیمال که برابر با 160 بیت است را دریافت کرده (با استفاده از پارامتر –i) و آدرس فایل خروجی که رمز شده هر سطر فایل ورودی است را نیز دریافت کرده (با استفاده از پارامتر -o) و با استفاده از تابع encrypt\_input\_file این تبدیل را انجام می دهد. باز هم مانند Mode های قبلی با پارامتر –k می توان کلید رمزنگاری را عوض کرد.

در Mode چهارم به عنوان ورودی کلید رمزنگاری با پارامتر –k وارد می شود و همینطور آدرس فایل ورودی را نیز می توان با پارامتر –i وارد کرد. عملیات تست Avalanche با استفاده از تابع create\_round\_results و با استفاده از آدرس ورودی و کلید به ازای هر دور انجام می شود و نتیجه هر دور در فایلی با عنوان Round\_{i}.txt که به ازای i شماره هر دور است نوشته می شود.

حال به بررسی تک تک توابع می پردازیم. تابع create\_round\_results در بخش Avalanche.py بررسی می شود.

**تابع convert\_string\_to\_bitstream**

در این تابع یک رشته به عنوان ورودی به آن داده شده و به عنوان خروجی لیستی از bitstream ها که هر کدام 160 بیت هستند برگردانده می شود. اگر bitstream آخر کمتر از 160 بیت باشد Unicode حرف x که 01111000 را اضافه می کنیم.

**تابع convert\_bitstream\_to\_string**

این تابع برعکس تابع قبل است و به عنوان ورودی لیست bitstream ها را دریافت کرده و آنها را تبدیل به متن رشته ای کرده و به عنوان خروجی برمی گرداند.

**تابع encrypt\_plain\_text**

متن ساده و کلید را به عنوان ورودی می گیرد و با استفاده از تابع convert\_string\_to\_bitstream متن ساده را به لیست bitstream ها تبدیل کرده و هر کدام را توسط شیء serpent رمز کرده و ذخیره می کنیم. سپس با استفاده از تابع convert\_bitstream\_to\_string متن رمز شده را به دست آورده و در نهایت متن رمز شده و لیست bitstream رمز شده را به عنوان خروجی پس می دهیم.

**تابع decrypt\_cipher\_text**

برعکس تابع قبل است. به عنوان ورودی متن رمز شده و لیست bitsteam رمزشده و کلید را دریافت کرده و با استفاده از لیست bitstream و شیء serpent لیست متن رمزگشایی شده bitstream را به دست آورده و سپس با استفاده از تابع convert\_bitstream\_to\_string متن ساده را به دست می آوریم.

**تابع encrypt\_plain\_text\_binary**

متن ساده باینری و کلید را به عنوان ورودی گرفته و با استفاده از شیء serpent آن را رمز کرده و بر می گرداند.

**تابع decrypt\_cipher\_text\_binary**

برعکس تابع قبل است. متن رمزشده باینری را به همراه کلید دریافت کرده و با استفاده از شیء serpent متن ساده باینری را به دست می آورد.

**تابع encrypt\_input\_file**

در این تابع آدرس فایل های ورودی و خروجی به همراه کلید به عنوان ورودی دریافت شده و در نهایت فایل خروجی ساخته می شود. در این تابع از دو تابع bin\_to\_hexa و hexa\_to\_bin استفاده شده است. این دو تابع برای تبدیل رشته باینری به رشته هگزادسیمال و برعکس استفاده می شود. فایل ورودی را باز کرده و هر خط آن را می خوانیم سپس بعد از تبدیل هر خط با استفاده از hexa\_to\_bin به شیء serpent می دهیم که آن را رمز کند سپس نتیجه را پس از اعمال bin\_to\_hexa در فایل خروجی می نویسیم.

**Avalanche.py**

**توابع bin\_to\_hexa و hexa\_to\_bin**

برای تبدیل رشته باینری به هگزادسیمال و برعکس استفاده می شوند.

**تابع create\_random\_number**

به عنوان ورودی تعداد ورودی های تصادفی را دریافت کرده (160) و به صورت رندوم رشته باینری 160 بیتی را می سازد. سپس این رشته را با استفاده از bin\_to\_hexa به هگزادسیمال تبدیل می کنیم.

این تابع مستقیم در فرآیند Avalanche تاثیر ندارد و برای ساخت input.txt استفاده می شود.

**تابع create\_hex\_random\_file**

اسم فایل و همینطور تعداد بیت های تصادفی (160) را به عنوان ورودی دریافت کرده و سپس با استفاده از تابع create\_random\_number پانصد خط خروجی هگزادسیمال را در فایل ایجاد می کنیم.

این تابع مستقیم در فرآیند Avalanche تاثیر ندارد و برای ساخت input.txt استفاده می شود.

**تابع create\_binary\_random\_file**

این تابع دو اسم فایل را به عنوان ورودی می گیرد که ورودی اول آدرس فایل هگزادسیمال تصادفی است که این فایل را می خوانیم و آن را تبدیل به باینری کرده و در فایل دوم ذخیره می کنیم.

این تابع مستقیم در فرآیند Avalanche تاثیر ندارد و برای داشتن رشته های باینری تصادفی برای مثال کلید استفاده می شود.

**تابع change\_one\_bit**

این تابع یک رشته باینری و یک ایندکس را دریافت کرده و بیت ایندکس را برعکس کرده و به عنوان خروجی پس می دهد.

**تابع get\_collection\_of\_inputs**

یک رشته باینری را دریافت کرده و به اندازه تعداد بیت های رشته باینری change\_one\_bit را با ایندکس مناسب صدا می کنیم و لیست نتایج را به عنوان خروجی پس می دهیم.

**تابع calculate\_list\_of\_distinct**

در این تابع دو رشته باینری به عنوان ورودی داده می شود و ایندکس هایی که این دو با یکدیگر تفاوت دارند در یک لیست به عنوان خروجی پس داده می شود.

**تابع calculate\_result\_output**

ورودی اول لیستی است که با استفاده از رشته باینری ورودی و تابع get\_collection\_of\_inputs به دست آمده است. ورودی دوم خود رشته باینتری ورودی است. ورودی سوم شیء serpent است. ورودی چهارم ایندکس شماره دور الگوریتم serpent است. ورودی پنجم کلید رمزنگاری است.

این تابع دارای دو خروجی است. خروجی اول لیست رشته باینری دور اندیس از لیست ورودی اول است و خروجی دوم رشته باینری دور اندیس از ورودی دوم است.

**تابع calculate\_matrix\_one\_row**

ورودی اول ماتریس یک دور است. ورودی دوم رشته بیتی است که بقیه رشته بیت ها با آن مقایسه می شوند. ورودی سوم رشته بیتی است که با هر بار صدا زدن عوض می شود. ورودی چهارم اندیس شماره سطر است.

با استفاده از تابع calculate\_list\_of\_distinct لیست اندیس های تفاوت ورودی دوم و سوم را به دست می آوریم سپس به آن شماره سطر مورد نظر از ماتریس دور، اندیس های تفاوت را به اضافه یک می کنیم. به عنوان خروجی چیزی را بر نمی گردانیم ولی ماتریس دور را تغییر داده ایم.

**تابع calculate\_matrix\_all\_rows**

ورودی اول ماتریس است. ورودی دوم رشته ورودی باینری است. ورودی سوم شیء serpent است و ورودی چهارم اندیس دور است و ورودی پنجم کلید رمزنگاری است.

با استفاده از get\_collection\_of\_inputs لیست باینری را از رشته باینری اولیه به دست می آوریم سپس با استفاده از calculate\_result\_output لیست باینری خروجی و باینری خروجی را به دست می آوریم. حال به ازای هر یک لیست باینری خروجی تابع calculate\_matrix\_one\_row را صدا می زنیم. این تابع نیز مانند تابع قبلی خروجی ندارد اما ماتریس را تغییر می دهد.

**تابع calculate\_inputs**

از آدرس ورودی استفاده کرده و لیست باینری را پس می دهد.

**تابع create\_round\_results**

ابتدا شیء serpent را می سازیم سپس با استفاده از calculate\_inputs مجموعه باینری های ورودی را داریم. حال به ازای هر دور الگوریتم serpent یک ماتریس خالی می سازیم سپس به ازای هر رشته باینری ورودی تابع calculate\_matrix\_all\_rows را صدا می زنیم. پس از آنکه همه رشته های ورودی را امتحان کردیم با استفاده از np.savetxt ماتریس را در یک فایل ذخیره می کنیم.

**SerpentPlus.py**

**تابع create\_final\_permutation**

این تابع ورودی ای نمی گیرد اما initial permutation در آن تعریف شده و از روی آن final permutation محاسبه می شود. چون جدول final permutation به طور مستقیم در الگوریتم استفاده می شود لذا این تابع در الگوریتم تاثیری ندارد.

**کلاس SerpentPlus**

زمانی که از این کلاس شیء ای ساخته می شود تابع \_\_init\_\_ آن صدا زده می شود که در آن جدول جایگشت اولیه (IPTable)، جدول جایگشت نهایی (IFTable)، S-Box ها به صورت اعدادی از صفر تا 16 در 8 لیست، تعداد کلید، تعداد دور، لیست R-Con برای KeyScheduling، اعداد باینری پی و نپر و رادیکال دو و نسبت طلایی مقدار دهی اولیه می شوند.

از روی S-Box ها که به صورت اعداد ده دهی هستند دو لیست SBoxBitstring و SBoxBitstringInverse به طوری که درون هر کدام 8 دیکشنری وجود دارد که کلید و مقدار آن به صورت باینری است (از متد bitstring استفاده شده) و نشاندهنده تبدیل S-Box است ساخته می شود.

**متد bitstring**

در این تابع یک ورودی عدد دهدهی و یک عدد دیگری به عنوان حداقل طول را می گیرد و این عدد را تبدیل به رشته باینری با حداقل طول داده شده می کند. از این متد فقط در \_\_init\_\_ استفاده می کنیم.

**متد change\_rcon**

این متد ورودی ای ندارد ولی درون آن R-Con به صورت هگزادسیمال تعریف شده است. این تابع این لیست هگزادسیمال را به باینری تبدیل می کند. چون به صورت مستقیم از R-Con باینری در \_\_init\_\_ استفاده کرده ایم، این متد تاثیری در روند الگوریتم ندارد.

**متد applyPermutation**

این متد جدول جایگشت را به همراه ورودی رشته باینری 160 بیتی تحویل می گیرد و در خروجی جدول جایگشت را بر ورودی اعمال کرده و بر می گرداند.

**متد IP**

ورودی را دریافت کرده و متد applyPermutation را با IPTable و ورودی صدا می زند و خروجی را بر می گرداند.

**متد FP**

ورودی را دریافت کرده و متد applyPermutation را با FPTable و ورودی صدا می زند و خروجی را بر می گرداند.

**متد IPInverse**

ورودی خود را دریافت کرده و چون IPInverse همان FP است متد FP را صدا می زند.

**متد FPInverse**

ورودی خود را دریافت کرده و چون FPInverse همان IP است متد IP را صدا می زند.

**متد S**

شماره S-Box را دریافت کرده (یا همان شماره دور) و ورودی رشته باینری 4 بیتی را دریافت کرده و با استفاده از SBoxBitstring عمل تبدیل را انجام می دهد و خروجی را بر می گرداند.

**متد SInverse**

مانند متد S است با این تفاوت که از SBoxBitstringInverse استفاده می کند.

**متد SHat**

یک دور کامل S-Box را با استفاده از متد S اجرا می کند به این صورت که شماره S-Box (یا همان شماره دور) را به همراه ورودی رشته باینری 160 بیتی دریافت کرده و 4 بیت 4 بیت جدا کرده و به متد S می دهد و نتیجه نهایی را باز می گرداند.

**متد SHatInverse**

مانند متد قبل است با این تفاوت که از متد SInverse استفاده می کند.

**متد binaryXor**

دو رشته هم اندازه باینری را دریافت کرده و XOR آن ها را به صورت رشته باینری پس می دهد. این متد در KeyScheduling و linear transformation و تابع دور(R و RInverse) استفاده می شود.

**متد bitstring2matrix**

یک ورودی 160 بیتی را می گیرد و به ماتریس تبدیل می کند. از این متد در KeyScheduling استفاده می کنیم.

**متد get\_n\_column**

ماتریس و شماره ستون را به عنوان ورودی گرفته و ستون مورد نظر را پس می دهد. از این متد در KeyScheduling استفاده می کنیم.

**متد put\_column\_to\_matrix**

ماتریس و ستون را به عنوان ورودی دریافت کرده و ستون را به ماتریس اضافه می کنیم. سپس ماتریس نهایی را پس می دهیم. از این متد در KeyScheduling استفاده می کنیم.

**متد sbox\_key\_scheduling**

ورودی 8 بیتی رشته باینری به همراه شماره S-Box (یا شماره دور) به متد داده می شود و این متد با استفاده از SBoxBitstring و تقسیم کردن ورودی به دو 4 بیت تبدیل را انجام داده و خروجی را بر می گرداند.

**متد get\_key\_from\_key\_columns**

ماتریس کامل شده در KeyScheduling را به همراه شماره کلید مورد نظر دریافت کرده و کلید را به صورت رشته باینری 160 بیتی می سازد و نتیجه را بر می گرداند.

**متد key\_scheduling**

این متد کلید اصلی 160 بیتی را به صورت رشته باینری تعریف کرده و از متدهای بالا که برای این متد ساخته شده اند استفاده کرده و دقیقا مانند KeyScheduling الگوریتم Rijndael همه کلید ها را ساخته و در یک لیست قرار می دهد.

ابتدا ورودی 160 بیتی را به ماتریس تبدیل می کند (با استفاده از متد bitstring2matrix) سپس به اندازه تعداد کلید های مد نظر ضربدر چهار حلقه ای را تکرار می کند. در این حلقه ابتدا ستون آخر ماتریس دریافت می شود (با استفاده از متد get\_n\_column) سپس اگر این ستون، ستون ضریب چهار از ماتریس باشد یک سری دستورات اجرا می شود و اگر نباشد دستورات دیگری اجرا می شود.

اگر ستون ضریب چهار باشد ابتدا یک Rotate انجام می شود سپس S-Box روی ستون آخر اعمال می شود (با استفاده از متد sbox\_key\_scheduling) سپس R-Con بر روی سطر اول از ستون آخر اعمال می شود. سپس ستون چهار تا مانده به آخر را به دست آورده و با ستون آخر XOR می کنیم. سپس ستون جدید را به ماتریس اضافه کرده و شمارنده را یکی زیاد می کنیم.

اگر ستون ضریب چهار نباشد ستون چهار تا مانده به آخر را پیدا کرده و با ستون آخر XOR می کنیم سپس ستون جدید را به ماتریس اضافه می کنیم.

سپس پس از آنکه حلقه تمام شد به اندازه تعداد کلید ها در یک لیست کلید های 160 بیتی را بر می گردانیم (با استفاده از متد get\_key\_from\_key\_columns)

**متد leftRotate**

رشته باینری را به همراه تعداد Rotate می گیرد و عمل Rotate به چپ را انجام می دهد. از این متد در linear transformation استفاده شده است.

**متد rightRotate**

رشته باینری را به همراه تعداد Rotate می گیرد و عمل Rotate به راست را انجام می دهد. از این متد در linear transformation استفاده شده است.

**متد znotfill**

در عملیات ریاضی linear transformation ممکن است که تعداد بیت ها بیش از 40 شود لذا از این متد استفاده کرده و بیت های اضافی را دور می ریزیم.

**متد linear\_transformation**

ورودی 160 بیتی را دریافت کرده و به چهار بخش 40 بیتی A, B, C, D تقسیم می کنیم. مراحل مانند encrypt یک دور از الگوریتم RC6 اجرا می شود با این تفاوت که به جای کلید از اعداد ثابت پی، نپر، رادیکال دو و نسبت طلایی استفاده شده است و از پیمانه 2 به توان 40 استفاده شده است.

**متد linear\_transformation\_inverse**

ورودی 160 بیتی را دریافت کرده و به چهار بخش 40 بیتی A, B, C, D تقسیم می کنیم. مراحل مانند decrypt یک دور از الگوریتم RC6 اجرا می شود با این تفاوت که به جای کلید از اعداد ثابت پی، نپر، رادیکال دو و نسبت طلایی استفاده شده است و از پیمانه 2 به توان 40 استفاده شده است.

**متد R**

این متد یک دور از الگوریتم را پیاده سازی می کند به این صورت که شماره دور و خروجی دور قبل و لیست کلید را دریافت کرده و ابتدا خروجی دور قبل را با کلید مربوطه XOR کرده و سپس S-Box را روی آن اعمال می کند سپس اگر دور آخر نباشد linear transformation را اعمال کرده و اگر دور آخر باشد با کلید نهایی XOR می کند و نتیجه را بر می گرداند.

**متد RInverse**

مانند متد R است فقط به این صورت است که مراحل به صورت برعکس است یعنی اول linear transformation سپس S-Box و سپس XOR با کلید اعمال می شود.

**متد encrypt**

این متد رشته 160 بیتی متن ساده را به همراه رشته کلید 160 بیتی دریافت کرده و ابتدا عملیات KeyScheduling را انجام می دهد سپس متن ساده را تحت متد IP اعمال می کند و سپس به اندازه تعداد دور ها متد R را اعمال می کند و در نهایت نیز از متد FP استفاده کرده و نتیجه را باز می کرداند.

**متد encrypt\_full\_round**

این متد مانند متد قبل است با این تفاوت که این متد برای استفاده در Avalanche نوشته شده است و تفاوت آن با متد قبل این است که خروجی های هر دور به همراه خروجی نهایی در یک لیست ذخیره می شوند و لیست به عنوان خروجی برگردانده می شود.

**متد decrypt**

این متد رشته 160 بیتی متن رمز شده را به همراه رشته کلید 160 بیتی دریافت کرده و ابتدا عملیات KeyScheduling را انجام می دهد سپس متن ساده را تحت متد FPInverse اعمال می کند و سپس به اندازه تعداد دور ها متد RInverse را اعمال می کند و در نهایت نیز از متد IPInverse استفاده کرده و نتیجه را باز می کرداند.