مقدمه ای بر استاندارد رمزنگاری پیشرفته (AES)

استاندارد رمزنگاری پیشرفته یا AES، پس از آنکه ضعف استاندارد رمزنگاری دیتا (DES) به خاطر طول کلید کوتاه آن و پیشرفت های تکنولوژیکی پردازنده ها مشخص شد، برنده ی رقابتی که در ۱۹۹۷ توسط دولت آمریکا برگزار شد می باشد. پانزده کاندیدا در ۱۹۹۸ پذیرفته شدند و بر اساس نظر عمومی این تعداد در ۱۹۹۹ به پنج تا رسید که فینالیست ها را تشکیل می داد. در اکتبر ۲۰۰۰ یکی از آن پنج الگوریتم به عنوان استاندارد آینده انتخاب شد که نسخه ی تغییر یافته ی Rijndael بود.

Rijndael که نام آن از دو بنیان گذار بلژیکی آن اقتباس شده است (Joan Daemen, Vincent Rijmen) یک روش رمزنگاری قطعه ای ست، یعنی بر روی قطعات با طول ثابتی از بیت ها کار می کند که بلاک نامیده می شوند. بلاکی با طول مشخص می گیرد، معمولا۱۲۸، و بلاکی باهمان اندازه به عنوان خروجی تولید می کند. تبدیل به ورودی دیگری نیاز دارد که همان کلید مخفی ست. مهم است بدانیم که کلید مخفی از هر طولی می تواند باشد (بسته به طول متنی که باید رمز شود) و AES سه طول کلید مختلف استفاده می کند: ۱۲۸، ۱۹۲ و ۲۵۶ بیت.

برای رمز کردن متونی که طولانی تر از طول بلاک هستند یک حالت عملیات (mode of operation) انتخاب می شود که در انتها شرح داده خواهد شد.

AES فقط اندازه بلاک های ۱۲۸ و طول کلید های گفته شده را می پذیرد اما Rijndael اندازه قطعه و طول کلیدهایی را که مضربی از ۳۲، حداقل ۱۲۸ و حداکثر ۲۵۶ باشند را می پذیرد.

مطالعه بیشتر:

DES بر اساس شبکه ی فیستل بود اما AES بر اساس شبکه ی جانشینی-جایگشتی است که یک سری از عملیات ریاضی ست که از جانشینی (که S-Box هم نامیده می شود) و جایگشت (P-Box) استفاده می کند و تعریف دقیق آن ها نشان می دهد که هر خروجی به تمام بیت های ورودی وابسته است.

توضیح AES

AES یک رمز قطعه ای مرحله ای (itrated) است که اندازه بلوک آن ۱۲۸ و طول کلید آن متغیر است. تبدیلات متفاوتی بر روی نتایج میانی که حالت (state) نامیده می شوند اثر می کند. حالت، یک آرایه ی مربعی از بایت هاست که چون قطعه ۱۲۸ بیتی یا ۱۶ بایتی ست، آرایه ۴*۴ می باشد. (در نسخه ی Rijndael که سایز بلاک ها متغیر است، تعداد سطر ها ثابت و برابر ۴ و تعداد ستون ها متفاوت است. تعداد ستون ها خارج قسمت طول بلاک بر ۳۲ است و Nb را مشخص می کند). کلید رمزنگاری به طریق مشابه به صورت آرایه ای مربعی با ۴ سطر مشخص می شود. تعداد ستون های کلید رمزنگاری که Nk را تشکیل می دهد خارج قسمت طول کلید بر ۳۲ است.

```
A state:
----------------------------
| a0,0 | a0,1 | a0,2 | a0,3 |
| a1,0 | a1,1 | a1,2 | a1,3 |
| a2,0 | a2,1 | a2,2 | a2,3 |
| a3,0 | a3,1 | a3,2 | a3,3 |
----------------------------

A key:
----------------------------
| k0,0 | k0,1 | k0,2 | k0,3 |
| k1,0 | k1,1 | k1,2 | k1,3 |
| k2,0 | k2,1 | k2,2 | k2,3 |
| k3,0 | k3,1 | k3,2 | k3,3 |
----------------------------
```

مهم است توجه کنیم که بایت ها ی ورودی بر بایت ها ی حالت نگاشته می شوند که به این ترتیب است: a0,0,a1,0,a2,0,a3,0,a0,1,a1,1,a2,1,a3,1و ... و بایت های کلید رمزنگاری به آرایه ای با این ترتیب نگاشته می شوند: k0,0,k1,0,k2,0,k3,0,k0,1,k1,1,k2,1,k3,1و ...در انتهای عملیات رمزنگاری خروجی رمزنگاری، از حالت، با گرفتن بایت های

حالت به همان ترتیب حاصل می شود. AES از تعداد دور های متفاوتی استفاده می کند که ثابت هستند: کلیدی به اندازه ی ۱۲۸، ۱۰ دور دارد، کلید ۱۹۲ بیتی ۱۲ دور دارد و کلید ۲۵۶بیتی ۱۴ دور. در هر دور عملیات زیر بر روی حالت اعمال می شوند:

۱- SubBytes: هر بایتی در حالت با استفاده از S-Box مربوط به Rijndael با یکی دیگر جایگزین می شود؛
۲- ShiftRow: هر سطر در آرایه ی ۴*۴ به میزان مشخصی به چپ شیفت داده می شود؛
۳- MixColumns: یک تبدیل خطی بر روی ستون های حالت؛
۴- AddRoundKey: هر باتی از حالت با کلید دوره ترکیب می شود که برای هر دور متفاوت است و از برنامه ی Rijndael مشتق شده است.

در دور نهایی عملیات MixColumns انجام نمی شود. الگوریتم شبیه شبه کد زیر است:

```
AES(state, CipherKey)
{
KeyExpansion(CipherKey, ExpandedKey);
AddRoundKey(state, ExpandedKey);
for (i = 1; i < Nr; i++)
{
Round(state, ExpandedKey + Nb*i);
}
FinalRound(state, ExpandedKey + Nb * Nr);
}
```

ملاحظات

- کلید رمزنگاری به یک کلید بزرگ توسعه داده می شود که بعدا برای عملیات واقعی استفاده می شود.
- کلید دوره قبل از شروع حلقه ی while به حالت اضافه می شود.
- FinalRound() همان Round() است که عملیات MixColumns از آن حذف شده است.
- در طی هر دور قسمت دیگری از ExpandedKey برای عملیات استفاده می شود.
- ExpandedKey باید همیشه از کلید رمزنگاری مشتق شود و هیچ وقت مستقیما تخصیص داده نشود.

عملیات AES: SubBytes، ShiftRow، MixColumns و AddRoundKey

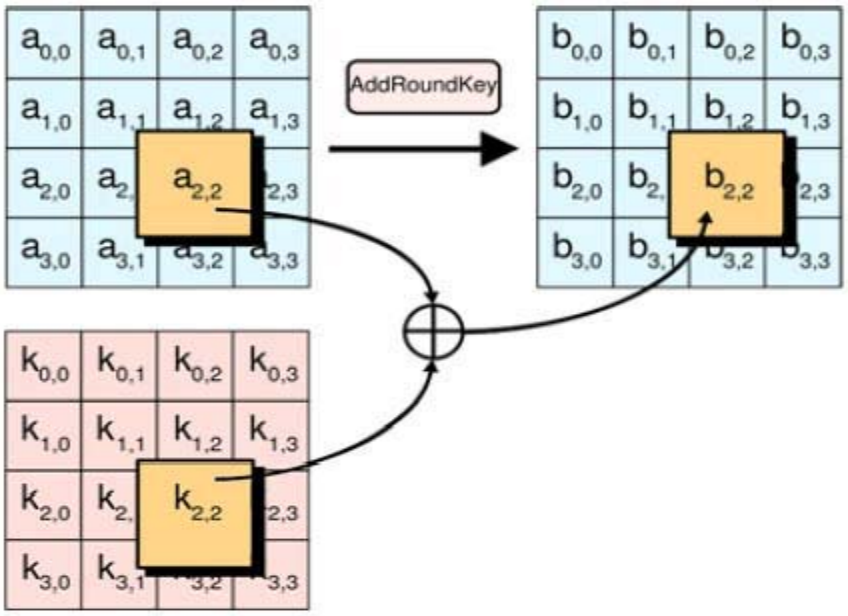
عملیات AddRoundKey

در این عملیات یک کلید دوره با یک XOR ساده بر روی حالت اعمال می شود. کلید دوره از روی کلید رمزنگاری با استفاده از یک زمانبند کلید مشتق می شود. طول کلید دوره برابر با طول بلاک کلید است (=۱۶ بایت).

```
-----------------------------     -----------------------------   -----------------------------
| a0,0 | a0,1 | a0,2 | a0,3 |     | k0,0 | k0,1 | k0,2 | k0,3 |   | b0,0 | b0,1 | b0,2 | b0,3 |
| a1,0 | a1,1 | a1,2 | a1,3 | XOR | k2,0 | k2,1 | k2,2 | k2,3 | = | b2,0 | b2,1 | b2,2 | b2,3 |
| a2,0 | a2,1 | a2,2 | a2,3 |     | k1,0 | k1,1 | k1,2 | k1,3 |   | b1,0 | b1,1 | b1,2 | b1,3 |
| a3,0 | a3,1 | a3,2 | a3,3 |     | k3,0 | k3,1 | k3,2 | k3,3 |   | b3,0 | b3,1 | b3,2 | b3,3 |
-----------------------------     -----------------------------   -----------------------------

where: b(i,j) = a(i,j) XOR k(i,j)
```

توضیح گرافیکی عملیات به شکل زیر است:



عملیات ShiftRow

در این عملیات هر سطر حالت شیفت حقه ای به چپ داده می شود که وابسته به اندیس سطر است.

اولین سطر به محل۰ در چپ منتقل می شود؛

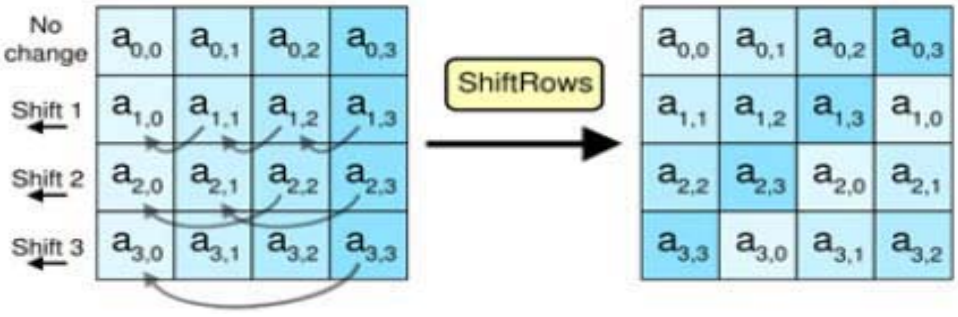
دومین سطر به محل۱ در چپ منتقل می شود؛

سومین سطر به محل۲ در چپ منتقل می شود؛

چهارمین سطر به محل۳ در چپ منتقل می شود؛

```
---------------------------     ---------------------------
| a0,0 | a0,1 | a0,2 | a0,3 |     | a0,0 | a0,1 | a0,2 | a0,3 |
| a1,0 | a1,1 | a1,2 | a1,3 | ->  | a1,1 | a0,2 | a1,3 | a1,0 |
| a2,0 | a2,1 | a2,2 | a2,3 |     | a2,2 | a2,3 | a2,0 | a2,1 |
| a3,0 | a3,1 | a3,2 | a3,3 |     | a3,3 | a3,0 | a3,1 | a3,2 |
---------------------------     ---------------------------
```

توضیح گرافیکی عملیات به شکل زیر است:



عملیات SubBytes

عملیات SubBytes یک عملیات جانشینی غیر خطی ست که بر روی هر بایت حالت مستقلا عمل می کند. جدول جانشینی (S-Box) معکوس پذیر است و از تکیب دو تبدیل ساخته شده است:

۱- گرفتن وارون ضربی در میدان متناهی Rijndael؛

۲- اعمال یک تبدیل affine که در مستندات Rijndael آمده است.

از آنجا که S-Box به هیچ ورودی ای وابسته نیست، از فرم های از پیش محاسبه شده ای استفاده می شود اگر حافظه ی کافی (۲۵۶ بایت برای یک S-Box) در دسترس باشد. سپس هر بایت حالت با مقداری موجود در S-Box که اندیس آن به مقدار موجود در حالت مربوط است جانشین می شود:

```
A(I,j)=SBox[a(I,j)]
```

توجه کنید که معکوس SubBytes همان عملیات است که با S-Box معکوس که از قبل محاسبه شده، انجام می شود.

عملیات MixColumns

این عملیات مشتمل بر محاسبات بسیار پیشرفته ای در میدان متناهی Rijndael است به همین خاطر این بخش بسیار خلاصه آورده می شود. به طور خلاصه این عملیات با ضرب ماتریسی با این ماتریس در ارتباط است:

```
2 3 1 1
1 2 3 1
1 1 2 3
3 1 1 2
```

و عملیات ضرب و جمع کمی با حالت عادی فرق دارند:

```
Addition and Substraction:
Addition and subtraction are performed by the exclusive or operation. The two
operations are the same;
there is no difference between addition and subtraction.
Multiplication in Rijndael's galois field is a little more complicated. The
procedure is as follows:
Take two eight-bit numbers, a and b, and an eight-bit product p
Set the product to zero.
Make a copy of a and b, which we will simply call a and b in the rest of this
algorithm
Run the following loop eight times:
1. If the low bit of b is set, exclusive or the product p by the value of a
2. Keep track of whether the high (eighth from left) bit of a is set to one
Rotate a one bit to the left, discarding the high bit, and making the low bit have
a value
of zero
3.
If a's hi bit had a value of one prior to this rotation, exclusive or a with the
hexadecimal
number 0x1b
4.
Rotate b one bit to the right, discarding the low bit, and making the high (eighth
from
left) bit have a value of zero.
5.
The product p now has the product of a and b
Thanks to Sam Trenholme for writing this explanation.
```

زمانبندی کلید Rijndael

یک زمانبند کلید باید یک کلید کوتاه را به کلیدی بزرگ تبدیل کند که اجزای آن در طی گام های (iteration) مختلف به کار می روند.

هر اندازه ای از کلید به اندازه ای متفاوت گسترده می شود:

کلید ۱۲۸ بیتی به ۱۷۶ بیت؛

کلید ۱۹۲ بیتی به ۲۰۸ بیت؛

کلید ۱۵۶ بیتی به ۲۴۰ بیت توسعه می یابد.

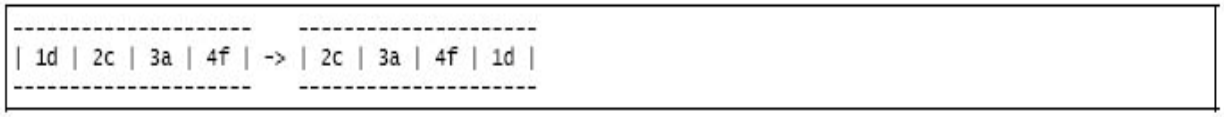
ارتباطی بین کلید رمزنگاری و تعداد دورها و اندازه ی ExpandedKey وجود دارد.برای کلید ۱۲۸بیتی یک عملیات اولیه ی AddRoundKey به علاوه ی ۱۰ دور که هر دور یک کلید ۱۶ بایتی جدید نیاز دارد، وجود دارد. بنابراین به ۱۰+۱ RoundKey ۱۶ بایتی نیاز داریم که برابر با ۱۷۶بایت است. منطق مشابهی در مورد سایر اندازه های کلید وجود دارد.

```
ExpandedKeySize =  (nbrRounds + 1) * BlockSize
```

زمانبند کلید تشکیل شده از گام های هسته ی زمانبند کلید است که بر روی کلمات (word) ۴ بایتی کار می کند. هسته از تعداد عملیات مشخصی استفاده می کند که در ادامه توضیح داده شده است:

چرخش (Rotate):

کلمه ی ۴بایتی یک بایت به چپ شیفت حلقوی داده می شود:



RCon:

این بخش هم محاسبات پیچیده ای دارد که ذکر نخواهد شد. فقط توجه کنید که مقادیر Rcon را می توان از قبل محاسبه کرد که منتهی به یک جانشینی ساده می شود (یک Table lookup) در یک جدول Rcon ثابت (اگر حافظه کافی در دسترس نباشد Rcon را می توان هر بار محاسبه کرد).

S-Box:

زمانبند کلید از همان S-Box جانشینی بدنه ی الگوریتم اصلی استفاده می کند.

هسته ی زمانبند کلید:

حال که می دانیم عملیات چه هستند شبه کد زمانبند کلید را می بینیم:

```
keyScheduleCore(word)
{
	Rotate(word);
	SBoxSubstitution(word);
	word[0] = word[0] XOR RCON[i];
}
```

در کد بالا اندازه ی کلمه ۴ بایت است و i شمارنده ی گام از زمانبند کلید است.

گسترش کلید:

ابتدا اجازه دهید تابع گسترش کلید را ببینیم که در مستندات Rijndael آمده است (دو نسخه از آن وجود دارد، یکی برای طول کلیدهای ۱۹۲ و ۱۲۸ و دیگری برای ۲۵۶):

```
KeyExpansion(byte Key[4*Nk] word W[Nb*(Nr+1)])
{
	for(i = 0; i < Nk; i++)
		W[i] = (Key[4*i],Key[4*i+1],Key[4*i+2],Key[4*i+3]);
	for(i = Nk; i < Nb * (Nr + 1); i++)
	{
		temp = W[i - 1];
		if (i % Nk == 0)
		temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
		W[i] = W[i - Nk] ^ temp;
	}
}
```

- Nk تعداد ستون ها در کلید رمزنگاری ست(۱۲۸ بیت - >۴، ۱۹۲بیت - >۵ و ۲۵۶ بیت - > ۶)
- نوع W "کلمه" است که ۴ بایت است

به زبان ساده:

- اولین n بایت کلید توسعه یافته همان کلید رمزنگاری ست (n = اندازه ی کلید رمزنگاری)؛
- مقدارi در rcon به ۱ مقدار دهی می شود؛

- تا وقتی که به تعداد بایت های کافی کلید گسترده شده داشته باشیم، کار های زیر را برای تولید n بایت کلید توسعه یافته ی بیشتر انجام می دهیم. توجه کنید که n به نسبت طول کلید متغیر است.

۱- برای تولید ۴ بایت این کارها را انجام می دهیم:
- از یک کلمه ی ۴ بایتی موقت به اسم t استفاده می کنیم
- ۴بایت قبلی را به t نسبت می دهیم.
- هسته ی زمانبند کلید را بر t اعمال می کنیم که i مقدار rcon آن باشد
- i را یکی زیاد می کنیم
- t را با کلمه ی ۴ بایتی ای که n بایت قبل در expandedKey هست XOR می کنیم (که n یکی از این مقادیر است: ۱۶، ۲۴ یا ۳۲ بایت)

۲- موارد زیر را x بار برای تولید x*4 بایت بعدی expandedKey انجام می دهیم(برای n=16,32، x=3، و برای n=24 ، x=5)
- کلمه ی۴ بایتی قبلی را به t نسبت می دهیم
- t را با کلمه ی ۴ بایتی ای که n بایت قبل از expandedKey است XOR می کنیم (که n n یکی از این مقادیر است: ۱۶، ۲۴ یا ۳۲ بایت)

۳- اگر n=32 (و فقط در این صورت) کارهای زیر را برای تولید ۴ بایت بیشتر انجام می دهیم:
- کلمه ی۴ بایتی قبلی را به t نسبت می دهیم
- هر ۴ بایت در t را به وسیله ی S-Box مربوط به Rijndael اجرا (run) می کنیم
- t را با کلمه ی ۴ بایتی ای که ۳۲ بایت قبل از expandedKey است XOR می کنیم

۴- اگر n=32 (و فقط در این صورت) کارهای زیر را سه بار برای تولید ۱۲ بایت بیشتر انجام می دهیم:
- کلمه ی۴ بایتی قبلی را به t نسبت می دهیم
- t را با کلمه ی ۴ بایتی ای که ۳۲ بایت قبل از expandedKey است XOR می کنیم

حالا ما کلید توسعه یافته را ایجاد کرده ایم.

پیاده سازی زمانبند به دشواری آن چیزی که تا کنون توضیح داده شد نیست. باید دقت کنید که:

- قسمتی که قرمز است فقط مخصوص کلید رمزنگاری اندازه ۳۲ است؛
- برای n=16 به تعداد ۴*۳ + ۴ بایت = ۱۶ بایت در هر گام تولید می کنیم
- برای n=24 به تعداد ۴*۵ + ۴ بایت = ۲۴ بایت در هر گام تولید می کنیم
- برای n=32 به تعداد ۴*۳ +۴ + ۴*۳ + ۴ بایت = ۳۲ بایت در هر گام تولید می کنیم

پیاده سازی زمانبند کلید بسیار واضح و روشن است، اما از آنجا که کد تکراری زیادی وجود دارد، می توان حلقه را کمی بهینه کرد و هنگامی که عملیات اضافه تری باید انجام شود از عملگر پیمانه ای برای چک کردن استفاده کرد.

پیاده سازی: زمانبند کلید

پیاده سازی AES را با گسترش کلید رمزنگاری آغاز می کنیم. چنان که در قسمت تئوریک بالا دیدید می خواهیم اندازه ی کلید رمزنگاری را که اندازه ی آن بین ۱۲۸ تا ۲۵۶ است، بیشتر کنیم و از آن کلید های دوره را به ادست آوریم.

بهتر است توابع کمکی مثل Rotat,Rcon یا S-Box پیاده سازی شوند و آزمایش شوند و بعد حلقه های بزرگتر تشکیل گردد. اگر از رویکرد پایین به بالا خوشتان نمی آید می توانید از کمی بعدتر از این قسمت شروع کنید و ادامه دهید.

پیاده سازی: توضیحات کلی

با اینکه برخی ممکن است فکر کنند اعداد صحیح بهترین انتخاب برای کار کردن است چون اندازه ی ۳۲ بیتی آن ها بسیار با یک کلمه تناسب دارد، قویا این را پیشنهاد نمی کنم. شما ممکن است به اشتباه فرض کنید اعداد صحیح یا به صورت خاص int همیشه ۴ بایتی هستند. در حالی که برد مورد نیاز برای int علامت دار و بی علامت کاملا مشابه برد نوع داده ی short است. در کامپایلر های با ۸ و ۱۶ بیت پردازنده (مثل پردازنده ی Intel x86 که در مود ۱۶ بیتی اجرا می شود، مثلا تحت Ms-DOS) عدد صحیح معمولا ۱۶ بیت است و نمادی درست مثل short دارد. در کامپایلر های ۳۲ بیت و بالاتر (مثل پردازنده ی Intel x86 که در مود ۳۲ بیتی اجرا می شود، مثلا Win32 یا Linux) یک int معمولا ۳۲ بیت طول دارد و همان نمایش نوع داده ی long را دارد.

به همین خاطر از char بدون علامت استفاده می کنیم چون اندازه ی کاراکتر (که CHAR_BIT نامیده می شود و در limits.h تعریف شده است) باید حداقل ۸ باشد. جک کلین می گوید: " تقریبا همه ی کامپیوتر های مدرن امروزی از بایت های ۸ بیتی استفاده می کنند اما هنوز کامپیوتر هایی وجود دارد که از اندازه های دیگری مثل ۹ بیت استفاده می کنند. همچنین برخی پردازنده ها (مخصوصا Digital Signal Processors) نمی توانند به حافظه در قطعاتی کمتر از اندازه ی کلمه ی پردازنده دسترسی داشته باشند. حد اقل یک DSP وجود دارد که من با آن کار کرده ام که CHAR_BIT آن ۳۲ بیت بوده است. نوع داده ی char، Short و long همه ۳۲ بیت هستند."

از آنجا که می خواهیم کد ما تا حد ممکن portable باشد و نیز چون به کامپایلر بستگی دارد که تعیین کند نوع داده ی پیش فرض char, علامت دار یا بی علامت است، ما char بی علامت را در سراسر کد استفاده می کنیم.

پیاده سازی: S-Box

مقادیر S-Box هم می تواند هر بار محاسبه شود و هم در حافظه ذخیره گردد. از آن جایی که فرض می کنیم هر ماشینی که کد ما بر روی آن اجرا می شود حداقل 2x 256byte دارد (دوتا S-Box داریم، یکی برای رمزنگاری و یکی برای رمزگشایی) مقادیر را در یک آرایه ذخیره می کنیم. به علاوه به جای اینکه به مقادیر بلافاصله از برنامه ی خودمان دسترسی داشته باشم یک تابع wrap اجاد خواهیم کرد که کد را خواناتر می کند و موجب می شد کد بیشتر را بعدا اضافه کنیم. البت این کار سلیقه ای ست و می توان مستقیما هم به آرایه دسترسی داشت.

این کدی برای دو S-Box است. این فقط یک table lookup است که مقداری از آرایه را بر می گرداند که اندیس آن توسط پارامترهای تابع مشخص شده است:

```
unsigned char sbox[256] =   {
//0     1    2     3     4    5     6     7     8    9     A     B    C     D     E     F
0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
```

```
0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16 }; //F

unsigned char rsbox[256] =
{ 0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb
, 0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb
, 0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e
, 0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25
, 0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92
, 0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84
, 0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06
, 0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b
, 0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73
, 0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e
, 0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b
, 0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4
, 0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f
, 0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef
, 0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61
, 0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d };

unsigned char getSBoxValue(unsigned char num)
{
    return sbox[num];
}

unsigned char getSBoxInvert(unsigned char num)
{
    return rsbox[num];
}
```

پیاده سازی: Rotate

از قسمت تئوریک به خاطر دارید که Rotate یک کلمه می گیرد (آرایه ی ۴ بایتی) و آن را ۸ بیت به چپ چرخش می دهد. چون ۸ بیت همان یک بایت است و نوع داده ی ما کاراکتر است، ۸ بیت چرخش به چپ به معنی شیفت چرخشی مقادیر آرایه به اندازه ی یک واحد به چپ است.

این کد تابع Rotate است:

```
/* Rijndael's key schedule rotate operation
* rotate the word eight bits to the left
*
* rotate(1d2c3a4f) = 2c3a4f1d
*
* word is an char array of size 4 (32 bit)
*/
void rotate(unsigned char *word)
{
      unsigned char c;
      int i;
      c = word[0];
      for (i = 0; i < 3; i++)
            word[i] = word[i+1];
      word[3] = c;
}
```

پیاده سازی: Rcon

درست مثل S-Box مقادیر Rcon هم می تواند هر بار محاسبه شود و هم در حافظه ذخیره گردد. من ترجیح می دهم آن را ذخیره کنم چون فقط به ۲۵۵ بایت نیاز دارد. برای مشابهت با پیاده سازی S-Box یک تابع دسترسی نوشته شده است:

```
unsigned char Rcon[255] = {

0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8,
0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3,
0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,
0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d,
0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab,
0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d,
0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25,
0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01,
0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,
0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa,
0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a,
0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02,
0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a,
0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,
0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,
0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04,
0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f,
0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5,
0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33,
0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb};

unsigned char getRconValue(unsigned char num)
{
    return Rcon[num];
}
```

پیاده سازی: هسته ی زمانبند کلید

پیاده سازی هسته ی زمانبند کلید با زبان شبه C بسیار ساده است. تمام کاری که کد انجام می دهد اعمال عملیات، یکی پس از دیگری بر روی کلمه ی ۴بایتی ست. پارامتر ها، کلمه ی ۴ بایتی و شمارنده ی گام است که Rcon به آن وابسته است.

```
void core(unsigned char *word, int iteration)
{
	int i;
	/* rotate the 32-bit word 8 bits to the left */
	rotate(word);
	/* apply S-Box substitution on all 4 parts of the 32-bit word */
	for (i = 0; i < 4; ++i)
	{
		word[i] = getSBoxValue(word[i]);
	}
	/* XOR the output of the rcon operation with i to the first part
	(leftmost) only */
	word[0] = word[0]^getRconValue(iteration);
}
```

پیاده سازی: گسترش کلید

پیاده سازی جایی ست که همه ی اینها با هم استفاده می شوند. همان طور که در لیست بزرگی در بخش تئوری در مورد گسترش کلید Rijndael دیدید، ما باید چند عمل را به تعداد دفعاتی انجام دهیم که به اندازه ی کلید وابسته است. چون اندازه ی کلید مقادیر محدودی را می گیرد، من تصمیم گرفتم آن را به صورت یک نوع داده ی شمارشی تعریف کنم. نه فقط این باعث می شود که اندازه کلید به سه مقدار محدود شود، بلکه باعث خوانایی کد می گردد.

```
enum  keySize{
	SIZE_16 = 16
	SIZE_24 = 24
	SIZE_32 = 32
};
```

تابع گسترش کلید ما به صورت اولیه به دو چیز نیاز دارد:

- کلید رمزنگاری ورودی

- کلید گسترش یافته ی خروجی

از آنجا که در C نمی توان اندازه ی یک آرایه را که به صورت اشاره گر به یک تابع پاس داده می شود دانست، اندازه ی کلید رمزنگاری را به صورت نوع داده ی شمارشی keySize و اندازه ی کلید گسترش یافته را به صورت نوع داده ی size_t به لیست پارامتر های تابع اضافه می کنیم. پروتوتایپ به صورت زیر خواهد شد:

```
void expandKey(unsigned char *expandedKey, unsigned char *key, enum
keySize, size_t expandedKeySize);
```

ضمن پیاده سازی تابع سعی می کنم تا حد امکان جزئیات تئوریک را دنبال کنم. همان طور که گفتم چون برخی اجزای کد تکرار می شوند سعی می کنم از تکرار کد پرهیز کنم. به جای نوشتن:

```
while (expanded_key_size < required_key_size)
{
      key_schedule_core(word);
      for (i=0; i<4; i++)
            some_operation();
}
```

از ساختار متفاوتی استفاده می کنم:

```
while (expanded_key_size < required_key_size)
{
      if (expanded_key_size % key_size == 0)
            key_schedule_core(word);
      some_operation();
}
```

این ساختار همان کار را می کند اما موقع استفاده از کلید ۲۵۶ بیتی که آن گام های اضافی را دارد، کار را راحت تر می کند. حال تابع گسترش کلید را ببینیم و بعد توضیحات در ادامه خواهد آمد.

```
void expandKey(unsigned char *expandedKey,
                  unsigned char *key,
                  enum keySize size,
                  size_t expandedKeySize)
{
      /* current expanded keySize, in bytes */
      int currentSize = 0;
      int rconIteration = 1;
      int i;
      unsigned char t[4] = {0}; // temporary 4-byte variable
      /* set the 16,24,32 bytes of the expanded key to the input key */
      for (i = 0; i < size; i++)
            expandedKey[i] = key[i];
      currentSize += size;
      while (currentSize < expandedKeySize)
      {
            /* assign the previous 4 bytes to the temporary value t */
            for (i = 0; i < 4; i++)
            {
                  t[i] = expandedKey[(currentSize - 4) + i];
            }
            /* every 16,24,32 bytes we apply the core schedule to t
            * and increment rconIteration afterwards
            */
            if(currentSize % size == 0)
            {
                  core(t, rconIteration++);
            }
            /* For 256-bit keys, we add an extra sbox to the calculation */
            if(size == SIZE_32 && ((currentSize % size) == 16)) {
```