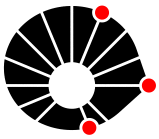

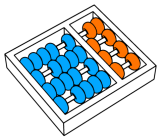

# Análise do sistema de criptografia do sistema operacional Android

*Bruno Benitez de Carvalho* *Islene Calciolari Garcia*

Relatório Técnico - IC-PFG-21-45
Projeto Final de Graduação
2021 - Dezembro

UNIVERSIDADE ESTADUAL DE CAMPINAS
INSTITUTO DE COMPUTAÇÃO

# Análise do sistema de criptografia do sistema operacional Android

Bruno Benitez de Carvalho Islene Calciolari Garcia

**Resumo**

O sigilo dos dados e operações realizadas por um computador torna-se cada vez mais relevante. Paralelamente, tornam-se mais intensos os ataques à segurança dos dados e à privacidade dos usuários. Também é fato conhecido que a miniaturização dos computadores, consolidada pelos smartphones, e recentemente por dispositivos vestíveis tornou-se amplamente comum, sendo o Android o sistema operacional mais utilizado nestes dispositivos. Tomando estas bases, este trabalho propõe-se a apresentar alguns dos princípios fundamentais para o entendimento da criptografia, em especial os algoritmos baseados em chave simétrica, e analisar as soluções de criptografia implementadas pela Google ao longo das versões de seu sistema operacional mobile e open-source, o Android.

## 1 Introdução

Proteger mensagens e informações sigilosas é uma prática que remonta aos primórdios da humanidade, principalmente em contextos de guerra e diplomacia. Com o avanço dos estudos matemáticos, e também da tecnologia disponível, novas técnicas e métodos surgiram ao longo da história, com destaque para a máquina de criptografia alemã Enigma, criada em 1918 e o sistema de criptografia de chave pública, introduzido por Diffie e Hellman em 1976. E já pode-se criar expectativas sobre o que se denomina de Criptografia Quântica.

A partir dos anos 2010, a introdução dos *smartphones*, aparelhos de celular com alta capacidade de processamento, tendo como pioneira a Apple, com seu iPhone, e seguida de perto pela Google, e seus dispositivos Android, trouxera não só a facilidade e praticidade de comunicação, mas junto com elas, uma imensidão de bytes extremamente sensíveis.

Dados do site *statcounter GlobalStats* mostram que que o Android é o sistema operacional mais popular do mundo na atualidade, tendo uma participação de mercado de 39.26%, incluindo sistemas operacionais *desktop* [11]. Sabendo da sensibilidade dos dados contidos nesses dispositivos, e da necessidade de desempenho e praticidade nos smartphones, a Google, desenvolvedora do sistema, já implementou algumas soluções para a criptografia de seu sistema operacional. O principal objetivo desta implementação é garantir que os dados armazenados localmente no dispositivo estejam seguros contra um acesso não autorizado. Para este fim, a estratégia mais adequada é o uso de algoritmos de criptografia baseados em chave simétrica. Ao longo deste trabalho, conceitos básicos de criptografia serão apresentados, além de uma breve análise das soluções implementadas pelo sistema operacional ao longo do tempo.

## 2 Uma introdução à criptografia de chave simétrica

Para entender criptografia de chave simétrica, e criptografia em geral, é necessário entender termos e conceitos básicos relacionados a essa teoria. Dentro deste contexto, a mensagem, ou conteúdo original é geralmente referida como **texto claro**, ou *plaintext*, no inglês. Já a mensagem cifrada, ou embaralhada, é chamada **texto cifrado** (*ciphertext*). **Encriptação** é o processo de converter um texto claro em texto cifrado, e o processo reverso é chamado de **decriptação**. Por fim, o esquema utilizado para encriptação é chamado **cifra**, ou **sistema criptográfico**.

O modelo de esquema criptográfico de chave simétrica se caracteriza pela existência de uma **chave secreta**, que é independente do texto claro, e junto com este é entrada do algoritmo de encriptação. As transformações e substituições aplicadas no texto claro pelo algoritmo são dependentes desta chave, e por isso, chaves diferentes produzem textos cifrados diferentes. Esta mesma chave também é utilizada pelo algoritmo de decriptação para recuperar o texto claro.

Mais formalmente, pode-se definir que $X$ é um texto claro criado por um emissor, e $K$ é uma chave secreta. Seja E um algoritmo de encriptação de chave simétrica, e $Y$ o texto cifrado que é gerado pelas entradas $X$ e $K$. Então, pode-se escrever tal operação como:

$$Y = E(K, X)$$

Analogamente, o algoritmo de decriptação D recebe o texto cifrado $Y$, e a mesma chave $K$, tendo como saída o texto original $X$. Assim:

$$X = D(K, Y)$$

A Figura 1 ilustra estes procedimentos:

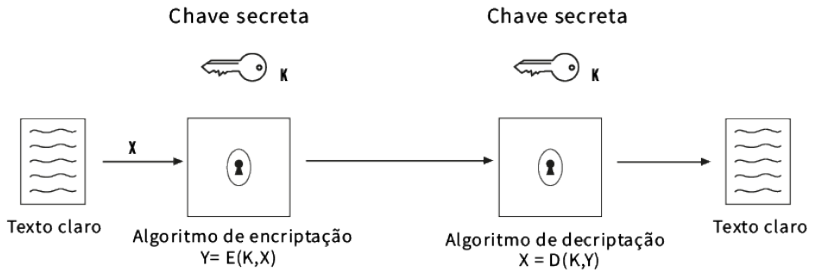


Figura 1: Esquema geral de encriptação e decriptação. Adaptado de Stallings [25].

Um dos grandes requisitos dos algoritmos de encriptação de chave simétrica é que ele precisa ser forte o bastante para que mesmo com o conhecimento do algoritmo de encriptação usado, e do texto cifrado, um atacante mal intencionado não consiga desvendar o texto claro. De fato, é desejável que o algoritmo utilizado seja de conhecimento público, pois dessa forma fabricantes podem implementar soluções diretas em hardware, como de fato é aplicado em casos como o dos processadores da linha ARM Cortex-A53 [2], e também por facilitar a análise e comprovação de sua eficácia.

Porém, o lado negativo de tal técnica é a fragilidade de se manter a chave em segredo. Por ela ser única, tanto o emissor de uma mensagem e o receptor dela precisam ter acesso à mesma chave. Sendo assim, é necessário um meio seguro, externo ao contexto da cifra, que seja capaz de transmitir a chave em sigilo. Este é um problema que foi resolvido graças à técnica de criptografia assimétrica.

## 2.1 Técnicas de criptografia

Para transformar um texto claro em um texto cifrado, um ou mais procedimentos precisam ser aplicados à entrada. Esses procedimentos envolvem a troca de letras do texto claro por outras letras, números e símbolos. E se a entrada for considerada como uma sequência de bits, o processo pode ser entendido como uma alteração nos padrões destes. A quantidade e complexidade das operações são um fator importante para determinar a eficácia de uma cifra. Duas principais técnicas de criptográfica se destacam: Substituição e Transposição. Elas serão descritas nas próximas subseções.

### 2.1.1 Técnicas de Substituição

O princípio da técnica de substituição é trocar as letras do texto claro por letras diferentes, ou símbolos e números. A escolha da forma como essa troca é feita varia bastante. Um exemplo clássico desta técnica é a Cifra de César, na qual cada letra do alfabeto é substituída por uma letra que fica três posições adiante no alfabeto. A generalização desta ideia é conhecida como **ROT-N**, já que o princípio de substituição é na verdade uma "rotação"do alfabeto em **N** letras, uma vez que o alfabeto recomeça no final. Neste caso, **N** é a chave do algoritmo. Por exemplo, o texto claro `'Bruno diz oi'` é transformado em `'Euxqr glc rl'` por ROT-3.

É importante notar que se for conhecido que a cifra usada for da família **ROT-N**, e a linguagem do texto claro é conhecida, basta que um ataque de força bruta teste as 25 alternativas possíveis para revelar o texto claro.

Muitas outras técnicas de substituição existem, e são utilizadas na prática, como a Cifra de Hill [16], e o algoritmo *Triple DES* [7]. Essas abordagens utilizam outras estratégias de substituição, em combinação com um leque de chaves muito maior, como por exemplo o *Triple DES*, que usa chaves de 168 bits (aproximadamente $3.7 \times 10^{56}$ chaves possíveis).

### 2.1.2 Técnicas de Transposição

As técnicas de transposição realizam algum tipo de permutação dos caracteres já presentes no texto claro. A cifra mais simples que utiliza esta técnica como base é conhecida como cifra **cerca de trilho**. Nela, o texto claro é escrito como uma sequência de diagonais, e depois as linhas individuais resultantes são concatenadas. A chave desse tipo de cifra é a "profundidade"das diagonais. Para ilustrar, considere a seguinte frase: `"Lucas_gosta_de_basquete"`. A mensagem pode ser visualizada da seguinte forma, escrita em diagonais de profundidade 3:

```
L   s   s   d   a   e
 u a _ o t _ e b s u t
  c   g   a   _   q   e
```

Sendo assim, após concatenar as linhas resultantes, o texto cifrado é: `"Lssdaeua_ot_ebsutcga_qe"`. É importante notar que nos exemplos as letras maiúsculas e minúsculas não foram alteradas, mas este é outro fator variável que é tratado de forma diferente por cada tipo de cifra.

Um esquema substancialmente mais complexo de transposição é escrever a mensagem em forma de um retângulo, linha por linha, permutar as colunas, e por fim, lê-la coluna por coluna. Para visualizar, considere a mensagem `"ataquem_pelo_noroeste"`. A escrita em forma de retângulo é:

```
Chave:      5 2 1 7 3 6 4
            a t a q u e m
            _ p e l o _ n
            o r o e s t e
```

A encriptação usando este esquema é feita escrevendo todos os caracteres da coluna com o rótulo 1, e sucessivamente de forma crescente. Neste exemplo, o texto cifrado é: `"aeotpruosmnea_oe_tqle"`. Desta forma, a chave é a ordem das colunas para a qual se recupera o texto claro, neste caso 5217364.

Um processo de criptoanálise simples consegue detectar uma cifra de transposição apenas pelo fato de as frequências do texto original serem exatamente as do texto cifrado. Por essa razão, as técnicas de transposição e substituição costumam ser empregadas juntas, e não raro são realizados mais de uma rodada de aplicações da cifra no mesmo texto.

## 3 A criptografia no Android

No sistema operacional Android, a encriptação de dados é baseada nas técnicas de chave simétrica. Se um usuário escolhe por utilizá-la, todos os dados criados por ele serão encriptados antes de serem escritos no disco, e todas as leituras também precisam executar a decriptação.

Desde a versão 5 até a versão 9 do Android esteve disponível a Encriptação de Disco Completo (*Full-Disk Encryption*, no inglês). Este método de encriptação é baseado no `dm-crypt`, que é um subsistema de criptografia do Kernel Linux, e utiliza uma única chave, que ainda é protegida pela senha do usuário para o dispositivo e protege toda a partição de usuário do sistema. O algoritimo utilizado por esse método é o 128 Advanced Encryption Standard (AES) com encadeamento de blocos de criptografia, cipher-block chaining (CBC), no inglês, como modo de operação e ESSIV:SHA256 como método de inicialização de vetores. A chave master é encriptada com AES 128 bits via chamadas à biblioteca OpenSSL. Por adotar a estratégia de encriptar a partição em sua totalidade, o método se mostra muito eficiente na questão de segurança, mas isso significa que a maioria das funcionalidades principais do sistema - como ligações, despertador, etc. - não estariam disponíveis imediatamente após o boot, uma vez que é necessário introduzir a chave para descriptografar a partição.

Por conta desta limitação, a versão 7 do sistema Android introduziu o Sistema de Criptografia Baseada em Arquivos, ou *File-Based Encryption* (FBE). Na abordagem deste sistema de encriptação, diferentes pastas e arquivos são encriptados e descriptografados com chaves diferentes, derivadas da chave principal. Junto com o FBE, foi introduzido o conceito de *Direct Boot*, ou Inicialização Direta. Essa funcionalidade permite que alguns aplicativos possam operar em um contexto limitado, possibilitando a sua execução em um cenário em que o usuário ligou o smartphone, mas não o desbloqueou.

Para suportar o *Direct Boot*, uma mudança importante teve que ser implementada no paradigma de armazenamento para aplicativos cientes de criptografia. Trata-se da criação de dois espaços separados para armazenamento: Armazenamento Criptografado por Credencial (Credential Encrypted Storage, CE) e Armazenamento Criptografado por Dispositivo (Device Encrypted Storage, DE). O CE é o local de armazenamento padrão, e o seu conteúdo só está disponível após o usuário desbloquear o aparelho. Já no DE, os dados estão disponíveis durante o Direct Boot, antes do usuário desbloquear o smartphone (e depois também), e é onde informações importantes para o funcionamento básico do dispositivo devem ser armazenadas, como informações de alarmes e acessibilidade, por exemplo.

Um dos problemas iniciais introduzidos pelo FBE, foi a falta de encriptação para os metadados. Os metadados são informações referentes ao tamanho de arquivos, estruturas de diretórios, informações de permissão, e data de modificação. A versão 9 do sistema operacional introduziu o suporte à criptografia de metadados, que corrigiu este problema. Com apenas uma única chave, todo o conteúdo que não é criptografado diretamente pelo FBE é encriptado.

Uma parte essencial da criptografia é a **chave**. Como já abordado, no sistema de criptografia baseado em chave simétrica, essa chave é única, e é usada tanto para criptografar como descriptografar. No Android, as chaves são manipuladas por outros serviços e subsistemas do sistema operacional - Gatekeeper, Keymaster e Keystore - que fogem ao escopo deste estudo. Porém, é importante mencionar que as chaves tem tamanho de 512 bits, e também são encriptadas por uma chave de 256 bits, usando o algoritmo AES-GCM.

Uma observação final que pode ser feita na linha do tempo da criptografia no Android é que desde a versão 10 do sistema, todos os dispositivos já vem com Criptografia Baseada em Arquivos ativada por padrão. Isso significa mais conforto, pois os usuários não precisam se preocupar em ativar uma funcionalidade que potencialmente não conhecem, e não precisam formatar seu dispositivo para usufruir dos benefícios da criptografia. Outra observação importante é o fato de que a funcionalidade ser ativada por padrão indica que o impacto de desempenho é suficientemente pequeno de modo a ser entregue a todos usuários diretamente da fábrica. Em partes, isso é possível graças ao desenvolvimento de algoritmos de criptografia mais eficientes, em especial, o algoritmo denominado Adiantum, desenvolvido pela própria Google e focado em dispositivos com baixa capacidade de processamento.

## 4 Algoritmos baseados em chave simétrica

Nas seções anteriores foram citadas várias vezes nomes de algoritmos clássicos de criptografia baseados em chave simétrica. Esta seção explicará o funcionamento destes algoritmos

e suas variações, e também introduzirá um algoritmo relativamente novo, porém que usa componentes que são algoritmos já conhecidos, o Adiantum. Conceitos base para o entendimento destes algoritmos também serão explicados.

## 4.1 DES

Embora este algoritmo nunca tenha sido de fato utilizado pelo sistema operacional Android, ele é o predecessor do AES, que foi a primeira solução de criptografia oferecida pela Google para o seu sistema operacional móvel. DES é a sigla em inglês para *Data Encryption Standard*, ou padrão de encriptação de dados. Ele foi adotado em 1977 pelo NIST, National Institute of Standards and Technology, e passou a ser o algoritmo de encriptação simétrica dominante, uma vez que estava sendo utilizado pelo governo norte americano. Em 2001, o próprio NIST publicou o Advanced Encryption Standard (AES) para ser o novo padrão de criptografia baseada em chave simétrica, e desde então ele vem substituindo seu antecessor até os dias atuais.

O DES utiliza uma cifra de bloco. Isto significa que ele trata um bloco de texto claro como um todo e gera um bloco de texto cifrado do mesmo tamanho. Nestes algoritmos, os blocos têm um tamanho fixado em 64 bits, e a chave 56 bits (que na verdade têm 64 bits, mas somente 56 destes são utilizados pelo algoritmo). Este algoritmo é fortemente baseado na Cifra de Feistel, com a adição de permutação inicial e final, que não estão presentes no algoritmo original. A Cifra de Feistel foi criada com o objetivo de implementar os conceitos de difusão e confusão, introduzidos por Claude Shannon [24]. **Difusão** refere-se à dissipação de estruturas estatísticas do texto cifrado. Isso significa que cada bit do texto claro afeta muitos bits o texto cifrado. **Confusão** significa o grau de complexidade entre as relações estatísticas do texto cifrado e a chave.

O algoritmo realiza transformações em $n$ rodadas, tantas quanto desejadas, e no geral, mais rodadas significa mais segurança, mas também menos performance. O número de rodadas escolhida pelos projetistas do DES é 16. O texto claro de entrada é dividido em duas partes, $L_0$ e $R_0$, e cada rodada recebe os $L_i$ e $R_i$ - onde $L$ refere-se à metade esquerda, $R$ à metade direita, e $i$ o índice da rodada - derivados do bloco anterior, além da chave $K_i$, também derivada da chave original $K$.

Cada rodada realiza uma série de procedimentos sobre a entrada. Primeiramente, na parte esquerda do texto é realizada uma substituição, através de uma função F. O design desta função pode variar de acordo com a implementação. A função F tem como entrada a parte direita dos dados, e à sua saída aplica-se a operação de ou-exclusivo (XOR) com relação à parte esquerda dos dados. Feito isso, é realizada uma permutação, onde as metades direita e esquerda são trocadas. Assim, pode-se equacionar o conteúdo dos blocos na i-ésima iteração como:

$$L_i = R_i\text{–}1$$

$$R_i = L_i\text{–}1 \oplus F(R_i\text{–}1, K_i)$$

A Figura 2 esquematiza um trecho da execução da Cifra de Feistel durante as rodadas 1 e 2.

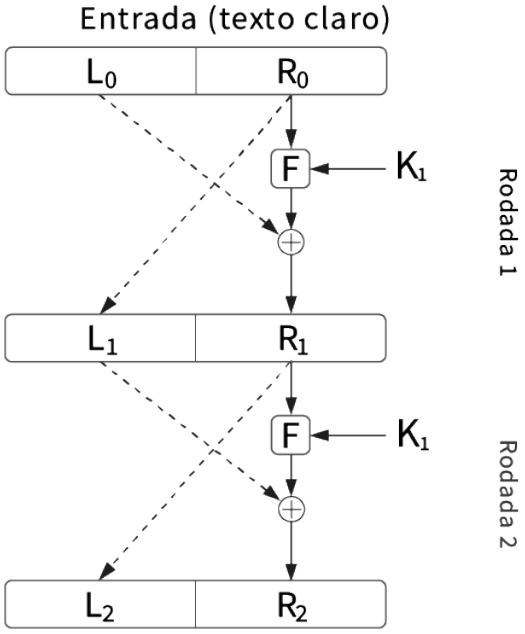


Figura 2: Rodadas 1 e 2 da Cifra de Feistel. Adaptado de Stallings [25].

Este algoritmo possui uma característica muito interessante, de que o processo de decriptação é feito utilizando-se a mesma Cifra de Feistel, com o texto cifrado como entrada, porém com as sub-chaves $K_i$ inseridas em ordem reversa. Uma prova deste comportamento pode ser encontrada na referência [24].

No DES o processamento do texto claro acontece em três etapas, e a chave é processada paralelamente para servir de entrada em cada uma das 16 rodadas dos blocos de processamento baseados na Cifra de Feistel. A primeira etapa do processamento do bloco de texto claro de 64 bits é chamada de IP, ou processamento inicial. Nela, é aplicada uma função de permutação em que os bits são reorganizados. A segunda etapa é a rede de 16 etapas, baseada na Cifra de Feistel, e ao final dela, os bits das metades da esquerda e direita são trocados para produzir a pré-saída. Com a pré-saída em mãos, é realizada a terceira etapa do algoritmo, que é aplicar a função inversa de permutação, também chamada IP-1, e a sua saída é o texto cifrado final.

O processamento da chave de 56 bits e sub chaves se dá pela seguinte lógica: (1) uma permutação inicial é aplicada na chave; e (2) a cada uma das 16 rodadas uma sub chave $K_i$ é gerada por se deslocar circularmente para a esquerda a chave proveniente da rodada anterior, adicionada de uma permutação que é a mesma em todas as rodadas. Assim, as sub chaves são utilizadas como entrada da função F nas rodadas apropriadas. A Figura 3 representa o funcionamento geral do DES.

O mesmo princípio que se aplica à decriptação da Cifra de Feistel também se aplica ao DES, sendo que a prova se encontra na mesma referência.

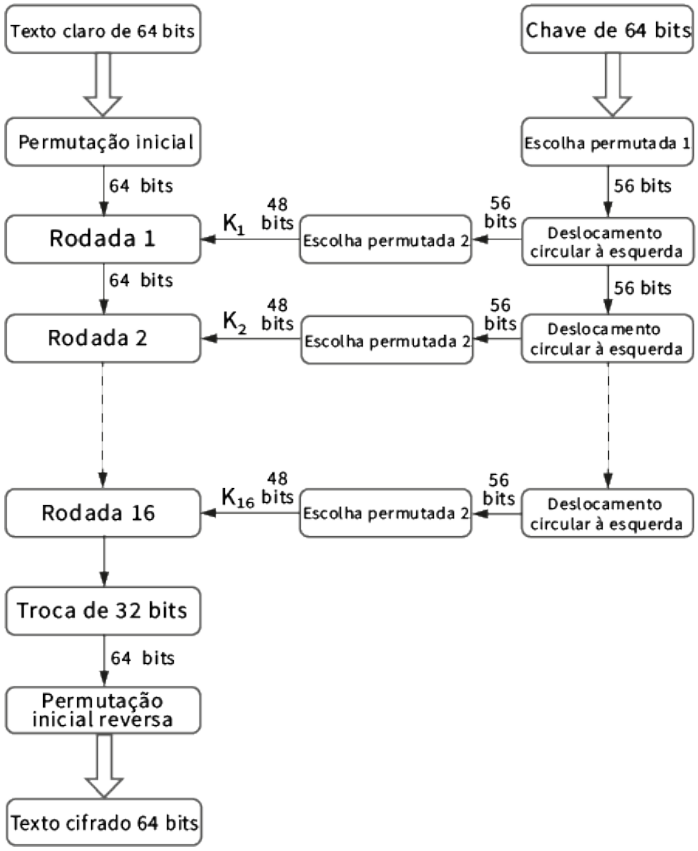


Figura 3: Visão geral do DES. Adaptado de Stallings [25].

## 4.2 AES

A necessidade de um novo padrão para criptografia surge com a evolução da capacidade de processamento dos computadores. Como visto na seção anterior, o DES usa uma chave de 56 bits, o que significa que existem $2 \times 10^{56}$ chaves diferentes que podem ser usadas, um número próximo de $7.2 \times 10^{16}$. Assumindo que uma máquina possa realizar uma operação de encriptação a cada microssegundo, e baseando-se na análise estatística de que em média metade das chaves precisam ser buscadas para se encontrar a desejada, chega-se ao resultado de que uma máquina levaria aproximadamente mil anos para ser bem sucedida em um ataque de força bruta ao DES. Porém, em 1977, ano em que o DES foi proposto, os próprios criadores, Diffie e Hellman já postularam que naquela época era possível criar uma máquina de processamento paralelo com um milhão de dispositivos realizando uma operação de decriptação por microssegundo, reduzindo o tempo de busca a 10 horas. A estimativa dos autores era de que esta máquina custaria 22 milhões de dólares na época [10].

Os processadores atuais também já ameaçam a segurança do DES, dada a existência de processadores multicores e taxas de clock próximas de 5GHz. A Seagate, em um estudo [23] sugere que é razoável assumir que os processadores multicore atuais podem atingir a

marca de $10^9$ chaves processadas por segundo. E um estudo mais recente [3] revela que esse número já é $10^{13}$. Dada essa necessidade, o AES foi criado, com variantes de 128, 192 e 256 bits de tamanho de chave. A Tabela 1 mostra um comparativo entre as diferentes cifras, o tamanho de chave e o tempo estimado de quebra por força bruta, assumindo uma velocidades de $10^9$ e $10^{13}$ decriptações por segundo.

| Tamanho da chave (bits) | Cifra | Quantidade de chaves diferentes | Tempo de quebra ($10^9$ dec./seg.) | Tempo de quebra ($10^{13}$dec./seg.) |
|---|---|---|---|---|
| 56 | DES | $2 \times 10^{56} \approx 7.2 \times 10^{16}$ | 1.125 ano | 1 hora |
| 128 | AES | $2 \times 10^{128} \approx 3.4 \times 10^{38}$ | $5.3 \times 10^{21}$ anos | $5.3 \times 10^{17}$ anos |
| 168 | Triple DES | $2 \times 10^{168} \approx 3.7 \times 10^{50}$ | $5.8 \times 10^{33}$ anos | $5.8 \times 10^{29}$ anos |
| 192 | AES | $2 \times 10^{192} \approx 6.3 \times 10^{57}$ | $9.8 \times 10^{40}$ anos | $9.8 \times 10^{36}$ anos |
| 256 | AES | $2 \times 10^{256} \approx 1.2 \times 10^{77}$ | $1.8 \times 10^{60}$ anos | $1.8 \times 10^{56}$ anos |

Tabela 1: Cifras e tempo de quebra estimado por força bruta. Adaptado de Stallings[25].

Observando-se a documentação do Android, pode-se verificar que um dos algoritmos disponíveis para criptografia baseada em arquivos é o AES-256-XTS, sendo o outro Adiantum [8]. Os princípios de funcionamento do AES são descritos a seguir, acompanhados de uma explicação sobre o modo de operação XTS.

Antes de iniciar a descrição do algoritmo, é importante notar que as operações no AES são feitas em 8 bits, e além disso, todas operações aritméticas são realizadas em cima do corpo GF($2^8$), um conjunto de $n-1$ elementos que contém todos os polinômios de grau menor ou igual à $n-1$ e com coeficientes binários. A descrição deste corpo e a justificativa de sua escolha podem ser analisadas em [9]. A grande importância do conhecimento desta escolha é a alteração no significado das operações, como se segue:

- A adição de dois bytes é definida como uma operação XOR bit a bit;
- A multiplicação $A \times B$ é equivalente a fazer um *shift* à esquerda em A num total de $B-1$ vezes. Se o polinômio equivalente que o produto representa tiver um grau maior que $n-1$ é necessário definir um polinômio irredutível $m(x)$ (pertencente à GF($2^8$)) para que seja feito um XOR bit à bit com ele. No AES o polinômio escolhido é $m(x) = x^8 + x^4 + x^3 + x + 1$ .

No AES, o bloco de texto de entrada sempre tem 128 bits. Geralmente, este bloco é referenciado como uma matriz de bytes 4 x 4, e para que as operações sejam realizadas nele, antes é preciso que ele seja copiado para um array denominado **estado**. Similarmente, a chave também é tratada como uma matriz quadrada, e ela passará por um processo de expansão para palavras chave. Uma chave de 128 bits, por exemplo, é expandida para 44 palavras de 4 bytes. O algoritmo também é baseado em rodadas, e o número $N$ delas depende do tamanho da chave. A Tabela 2 apresenta estes valores.

Quatro funções de transformação podem ser consideradas o núcleo do AES: `AddRoundKey`, `SubBytes`, `MixColumns` e `ShiftRows`. Essas funções são muito importantes e serão analisadas uma a uma. O procedimento inicial é referenciado como rodada 0. Nele, apenas uma

| Tamanho da chave (words/bytes/bits) | Tamanho do bloco (words/bytes/bits) | Quantidade de rodadas | Tamanho da chave da rodada (words/bytes/bits) | Tamanho da chave expandida (words/bytes) |
|---|---|---|---|---|
| 4/16/128 | 4/16/128 | 10 | 4/16/128 | 44/176 |
| 6/24/192 | 4/16/128 | 12 | 4/16/128 | 52/208 |
| 8/32/256 | 4/16/128 | 14 | 4/16/128 | 60/240 |

Tabela 2: Parâmetros de implementação do AES. Adaptado de Stallings[25].

aplicação de `AddRoundKey` é feita. A seguir, em cada uma das $N-1$ rodadas, são aplicadas as quatro transformações supracitadas, e na rodada final, apenas três transformações são feitas. Cada transformação tem como entrada uma matriz 4 x 4, sendo que apenas `AddRoundKey` faz uso da chave. A chave que é fornecida e é expandida para um array `w[i]`, com words de 32 bits, tendo tamanho definido pela Tabela 2. Quatro words distintas são usadas como chave em cada rodada. Uma visão geral do AES é apresentada na Figura 4.

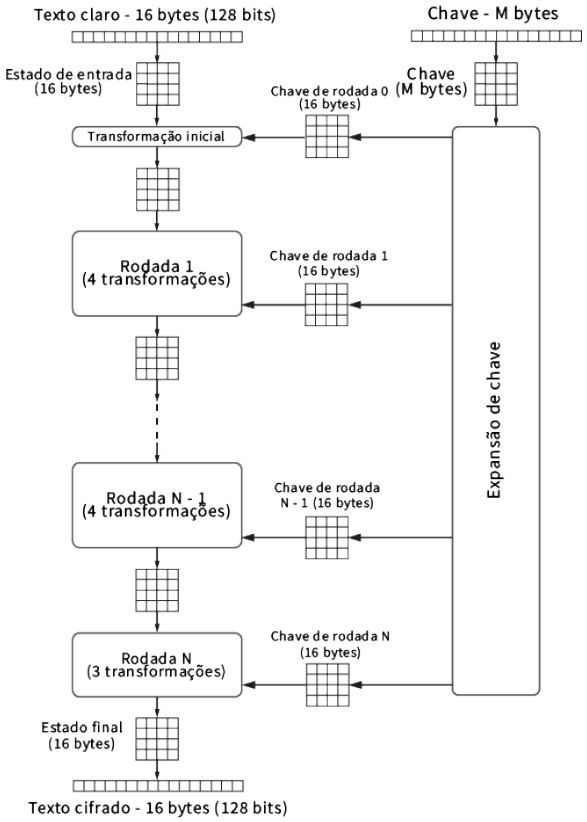


Figura 4: Visão geral do AES. Adaptado de Stallings [25].

A transformação `SubBytes` pode ser chamada como transformação direta de substituição de byte, e nela os bytes são mapeados para valores contidos em uma tabela de tamanho

16x16 definida pelo AES, chamada S-Box, que contém todas as 256 possíveis permutações de valores de 8 bits. Para o mapeamento, os 4 bits mais à esquerda do byte são usados para definir a linha, e os 4 bits mais à direita para definir a coluna. A transformação inversa de substituição de byte, chamada `InvSubBytes` utiliza o mesmo princípio de `SubBytes`, porém utiliza uma S-Box invertida. A descrição da construção de tais tabelas pode ser encontrada na referência [25]. A ideia por trás da criação das S-Box é diminuir ao máximo a correlação entre os bits de entrada e saída. Outra característica importante é de que a saída não pode ser descrita como uma função matemática simples da entrada, principalmente por conta do uso do inverso multiplicativo definido para GF($2^8$). Algumas propriedades chave para as S-Box são:

- $(\nexists a)(S\text{-}box(a) = a)$
- $(\nexists a)(S\text{-}box(a) = -a)$
- $IS\text{-}box[S\text{-}box(a)] = a$
- $(\nexists a)(S\text{-}box(a) = IS\text{-}box(a))$

Onde $-a$ é o complemento bit a bit de $a$.

A transformação `ShiftRows` também pode ser chamada de transformação direta de deslocamento de linhas. Nela, as linhas da matriz Estado sofrem um deslocamento circular à esquerda por 0, 1, 2 e 3 bytes na primeira, segunda, terceira e quarta linhas respectivamente. A função inversa desta operação, `InvShiftRows`, realiza os mesmos deslocamentos, porém à direita. Esta transformação tem um efeito considerável, já que a entrada é disposta em Estado por meio das colunas. Assim, um único deslocamento move um byte de uma coluna para outra, garantindo uma distância múltipla de 4 bytes sempre.

A próxima transformação é `MixColumns`, e pode ser chamada de transformação direta de embaralhamento de colunas. Ela pode ser definida como uma multiplicação de matrizes, utilizando as operações definidas para GF($2^8$), conforme segue:

$$\begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \tag{1}$$

Por exemplo, os elementos da primeira coluna são calculados pelas equações:

$$s'_{0,j} = (2 \cdot s_{0,j}) \oplus (3 \cdot s_{1,j}) \oplus s_{2,j} \oplus s_{3,j}$$

$$s'_{1,j} = s_{0,j} \oplus (2 \cdot s_{1,j}) \oplus (3 \cdot s_{2,j}) \oplus s_{3,j}$$

$$s'_{2,j} = s_{0,j} \oplus s_{1,j} \oplus (2 \cdot s_{2,j}) \oplus (3 \cdot s_{3,j})$$

$$s'_{3,j} = (3 \cdot s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 \cdot s_{3,j})$$

E a transformação inversa de embaralhamento de colunas, `InvMixColumns` é definida pela seguinte multiplicação de matrizes, semelhante à primeira:

$$\begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix} = \begin{bmatrix} 0B & 0E & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \quad (2)$$

O objetivo desta etapa é garantir um bom embaralhamento entre os bytes de cada coluna. Para isso, a escolha dos coeficientes da equação matricial de `MixColumns` foi bem estudado. `MixColumns` combinado com `ShiftRows` garantem que após algumas rodadas, todos os bits de saída dependam de todos os bits da entrada [9].

Por fim, `AddRoundKey`, também chamado de transformação direta de adição de chave da rodada, realiza uma operação de XOR bit a bit dos 128 bits da matriz Estado com a chave da rodada. É importante que tanto Estado quanto a chave são dispostos como matrizes de bytes 4 x 4 orientados por coluna. Não existe uma função de inversão separada para esta transformação, uma vez e a operação XOR é o seu próprio inverso, por definição. Esta é uma transformação relativamente simples, mas a principal fonte de segurança deste procedimento vem da expansão da chave. Também é garantido que cada bit de Estado é afetado.

### 4.2.1 Expansão da chave

Para explicar o algoritmo de expansão de chave será usada como exemplo uma chave de 128 bits (4 words de 4 bytes cada), mas o raciocínio é análogo para as variantes de 192 e 256 bits. A ideia principal do algoritmo é gerar uma sub chave para o processo inicial de `AddRoundKey` e alimentar este mesmo procedimento em cada uma das rodadas. Para a variante de 128 bits de chave do algoritmo, são necessárias 4 + 40 = 44 words, uma vez que é realizado o `AddRoundKey` inicial (rodada 0) e mais 10 rodadas completas. Esta saída é referenciada pelo array de words `w[44]`.

As primeiras quatro posições de `w` são preenchidas com a própria chave. As posições subsequentes são preenchidas da seguinte maneira:

- Se a word `w[i]` não está em uma posição múltipla de 4, o conteúdo de `w[i-1]` é copiado, e apenas uma operação XOR é realizada com a word em quatro posições anteriores.

- Caso a posição de `w[i]` seja múltipla de 4, primeiramente são realizadas as operações de deslocamento circular de um byte à esquerda, uma substituição utilizando a mesma S-Box utilizada em SubBytes, e após é realizada uma operação de XOR com a constante da rodada do AES. A constante da rodada foi construída de forma que a operação de XOR afete apenas o byte mais à esquerda da word. Sua definição pode ser encontrada em [9]. Finalmente, uma operação de XOR com `w[i-4]` também é feita.

Os desenvolvedores do algoritmo o criaram de forma a dificultar a reconstrução da chave com base em conhecimento prévio de partes dela. Isso quer dizer que quanto menos bits

conhecidos, mais difícil se torna reconstruir o restante. A introdução de constantes também foi pensada de forma a eliminar simetria ou similaridade na forma que as chaves são geradas em cada uma das rodadas do AES. Ele também foi feito para que a expansão seja reversível, isto é, seja possível recuperar a chave original com o conhecimento de *todos* os bits da chave expandida.

### 4.2.2 Modo XTS

Como observado na documentação [14], a variante do AES utilizada pelo Android para a criptografia baseada em arquivos é o AES-256-XTS. Os modos de operação surgem devido à possível brecha gerada por uma cifra ao ser aplicada repetidamente em vários blocos. Até aqui, foram apresentados algoritmos que operam sobre um bloco de tamanho definido de bits. Se o texto claro a ser encriptado for maior do que esse tamanho, é preciso encriptar vários blocos com a mesma chave, e isso é potencialmente perigoso.

O modo XTS para o AES foi aprovado pelo NIST em 2010 e tem sido amplamente usado desde então, como pelo próprio Android, e em dispositivos de armazenamento da Kingston Technology, por exemplo [17]. Ele é um método de encriptação pensado para dispositivos de armazenamento com base em setor, no qual se assume que um atacante pode ter acesso à parte do texto cifrado. Ele é uma implementação na prática do conceito de cifra de bloco ajustável [19].

As cifras de bloco ajustável têm como entrada, além da chave e do texto claro, um ajuste. Este valor de ajuste, geralmente referenciado pela letra T (abreviação para "tweak"), não precisa ser mantido em segredo, e sua introdução se deve à produzir variabilidade do texto cifrado, dados uma mesma chave e texto claro. O funcionamento do modo XTS é demonstrado na Figura 5.

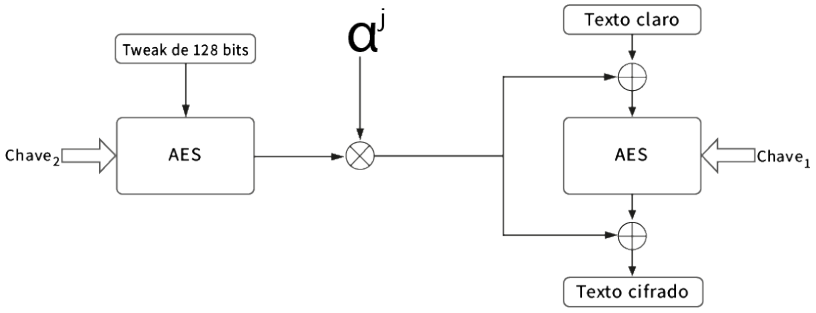


Figura 5: Operação do modo XTS. Adaptado de Kingston [17].

É importante notar que duas instâncias do AES são utilizadas no processo. Para cada uma delas, são usadas as chaves $K_1$ e $K_2$, de 256 bits cada uma (ou 128, dependendo da implementação).

Primeiramente, um valor de ajuste $i$ é definido para cada setor do sistema de armazenamento. Este valor tem 128 bits e passa pela encriptação do AES. A saída desta etapa passa por uma multiplicação modular com um polinômio $\alpha$ pertencente à GF($2^{128}$), sendo o

polinômio irredutível $m(x)$ escolhido para esse corpo como $m(x) = x^{128} + x^7 + x^2 + x + 1$. O valor $j$ é o número sequencial do bloco de 128 bits dentro do setor de armazenamento de dados. Este valor é usado para definir quantas vezes $\alpha$ será multiplicado por si mesmo antes de passar pela multiplicação com a o ajuste encriptado $i$. O resultado dessa operação é o ajuste que efetivamente será usado, e é referido por $T$. O bloco de texto claro, referido por $P_j$, passa então por uma operação de XOR com $T$. Este resultado passa então pelo AES, usando a segunda parte da chave como uma das entradas, e por fim, é realizado novamente uma operação de XOR entre $T$ e a saída da segunda chamada do AES.

A operação do AES em modo XTS pode ser equacionada da seguinte maneira:

$$T = AES(K_2, i) \oplus \alpha^j$$

$$C = AES(K_1, P \oplus T) \oplus T$$

Como se observa, são necessários diversos procedimentos complexos para a execução de um simples bloco de 128 bits. E levando em conta o fato de que nos sistemas de arquivo suportados pelo Android (Ext4 e F2FS) os setores têm 4096 bytes de tamanho, é natural que as operações de criptografia podem levar um tempo incomodo ao usuário final, especialmente em dispositivos não dotados de hardware dedicado para criptografia. Levando isso em consideração, e com a ideia de prover uma solução criptográfica para dispositivos chamados "de entrada" ou *low tier*, no jargão inglês, a Google empenhou esforços em projetar o algoritmo conhecido como Adiantum.

## 4.3 Adiantum

### 4.3.1 Visão geral

Adiantum é um modo de criptografia criado pela Google para prover criptografia em disco e arquivos, focado principalmente no desempenho em dispositivos sem um hardware específico para criptografia [15]. A partir do ARMv8, aceleração da criptografia por hardware foi introduzida, com suporte à AES e SHA, via ARMv8 Cryptography Extensions, que fornece instruções que tornam a multiplicação em GF($2^{128}$) muito mais rápidas. Porém, dispositivos de custo mais baixo, em geral, não possuem tal componente, por exemplo aqueles que usam um processador baseado em ARM Cortex-A7. Este modo foi criado visando fornecer soluções de segurança via criptografia para toda a gama de dispositivos que rodam Android, e usa como componentes chave quatro algoritmos já conhecidos pela comunidade: a cifra `ChaCha`, em sua variante `XChaCha12`, as funções de hash `NH` e `Poly1305`, e o próprio AES.

Uma das grandes diferenças entre AES e Adiantum é que a cifra `ChaCha` é uma **cifra baseada em fluxo**, enquanto AES é uma cifra de bloco. O grande contraste entre os dois tipos de cifra é que enquanto cifras de bloco operam sobre um bloco de tamanho fixo, possivelmente grande, as cifras de fluxo encriptam apenas um bit ou byte por vez. Isso é feito através da chave em combinação com o *nonce* ou *tweak* para gerar um fluxo de bits pseduo-aleatórios, chamado *keystream*, e esses bits do keystream são combinados com os bits do texto claro, geralmente através de uma operação de XOR bit a bit. O tweak é uma

formalização do conceito conhecido como nonce, número arbitrário que só pode ser utilizado uma vez [19].

A operação do modo Adiantum se dá pela seguinte sequência de procedimentos: primeiro, 4080 dos 4096 bytes ($P_L$) passam por uma operação de hash através de `NH` e `Poly1305` ($NH$-$P$) e uma saída de 16 bytes é gerada. Como entrada da função de hash, o tweak ($T$), que é um valor introduzido para gerar variância, de 32 bytes, também é utilizado. Em seguida a saída do hash passa por uma operação de soma módulo 128 com os 16 bytes restantes do texto claro. Os 16 bytes resultantes ($P_M$) da soma são introduzidos no AES, que os encripta, gerando $C_M$, que é fornecido como entrada para `XChaCha12`. Após, uma operação de XOR é realizada entre os $P_L$ e a saída de `XChaCha12`. O resultado será concatenado com 16 bytes que serão gerados através do hash destes mesmos $P_L$ por meio de `NH` e `Poly1305` e que são subtraídos, por meio da subtração módulo 128, de $C_M$. A Figura 6 mostra esse processo.

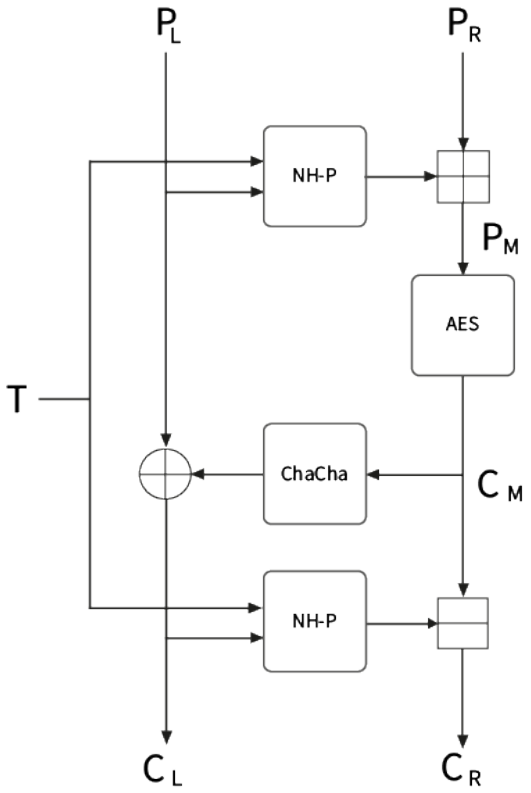


Figura 6: Visão geral do Adiantum. Adaptado de Google [8].

### 4.3.2 ChaCha

`ChaCha` é uma cifra de fluxo que é uma pequena variação da cifra `Salsa20`, ambas criadas por Daniel Julius Bernstein, e foram publicadas em 2007 e 2008, respectivamente. As cifras são baseadas no processo de geração de números pseudo-aleatórios por meio de

adição-XOR-rotação (AXR). O funcionamento chave desta cifra é mapear uma chave de 256 bits, um nonce de 64 bits e um contador de 64 bits para para um bloco de 512 bits pertencente à keystream.

Internamente, as entradas são dispostas em uma matriz 4x4 chamada Estado, assim como no AES. Cada célula de Estado é uma word de 32 bits. Além da chave, contador de posição e nonce, uma string constante é inserida em Estado. A string escolhida pelo autor da cifra é `expand 32-byte k`, que é considerado um "Nothing-up-my-sleeve number", uma constante utilizada em algoritmos de criptografia que estão acima de qualquer suspeita relacionada à introdução intencional de vulnerabilidades. Em `Salsa20` o Estado inicial é:

| | | | |
|---|---|---|---|
| const. | chave | chave | chave |
| chave | const. | nonce | nonce |
| contador | contador | const. | chave |
| chave | chave | chave | const. |

Tabela 3: Estado inicial de `Salsa20`. Adaptado de Bernstein [5].

A função principal de `Salsa20` é chamada de QuarterRound, e tem como entrada 4 words de Estado, assim, podendo ser referida por `QR(a,b,c,d)`. As operações dentro de `QR` são:

$$b = b \oplus (a \boxplus d) \lll 7$$

$$c = c \oplus (b \boxplus a) \lll 9$$

$$d = d \oplus (c \boxplus b) \lll 13$$

$$a = a \oplus (d \boxplus c) \lll 18$$

Onde $\oplus$ representa a operação XOR bit a bit, $\boxplus$ representa a operação de soma módulo $2^{32}$ e $\lll$ uma operação de rotação à esquerda de distância constante.

O algoritmo `Salsa20` realiza 20 rodadas de execução, sendo que nas rodadas ímpares a função `QR` é aplicada nos elementos de todas as colunas, e nas rodadas pares, `QR` é aplicada nos elementos de todas as linhas de Estado.

Já a cifra `ChaCha` utiliza a seguinte matriz Estado:

| | | | |
|---|---|---|---|
| const. | const. | const. | const. |
| chave | chave | chave | chave |
| chave | chave | chave | chave |
| contador | nonce | nonce | nonce |

Tabela 4: Estado inicial de `ChaCha`. Adaptado de Bernstein [5].

As principais diferenças com relação à matriz de `Salsa20` é que agora o nonce possui 96 bits de tamanho, enquanto o contador possui apenas 32. A disposição das entradas em Estado também são diferentes.

A função `QR(a,b,c,d)` em `ChaCha` também sofre alterações. Ainda são utilizadas 4 operações de soma módulo 32, 4 operações de XOR e 4 operações de shift, porém, cada word é atualizada *duas vezes* por rodada, o que aumenta a velocidade da difusão [5]. As operações de `QR` em `ChaCha` são:

$$\begin{array}{lll} a = a \boxplus b & d = d \oplus a & d = d \lll 16 \\ c = c \boxplus d & b = b \oplus c & b = b \lll 12 \\ a = a \boxplus b & d = d \oplus a & d = d \lll 8 \\ c = c \boxplus d & b = b \oplus c & b = b \lll 7 \end{array}$$

A equipe do Google, no entanto, optou por uma variante de `ChaCha` com apenas 12 rodadas, em vez de 20, para tornar a encriptação de disco rápida o suficiente na maior parte dos dispositivos. Uma variante de 7 rodadas já foi quebrada em 2008, e desde então diversos artigos foram publicados visando melhorar este ataque, mas até o momento, nenhum ataque é conhecido por ter conseguido quebrar a variante de 8 rodadas [15].

`XSalsa` e `XChaCha` são variações das cifras `Salsa` e `ChaCha`, respectivamente, que utilizam valores de nonce com tamanho maior, 128 bits, e esta é a versão utilizada pelo Adiantum [4] [1].

### 4.3.3 Poly1305

`Poly1305` é um autenticador criptográfico (MAC) [20]. Os autenticadores, ou algoritmos MAC são uados para verificar a integridade e autenticidade do dados. A função `Poly1305` recebe uma chave/nonce e 32 bytes e uma mensagem de tamanho arbitrário e produz um valor chamado de acumulador, ou tag de 16 bytes.

Inicialmente um acumulador, `ac` é inicializado com 0, e uma constante `P` é inicializada com o valor $(2^{130}) - 5$. Então, a chave é dividida em duas partes, `r` e `s`, tal que o par `(r,s)` seja único e sempre imprevisível, independente de cada chamada. Pode ser que `r` constante, mas ele precisa atender o seguintes requerimentos:

- `r[3]`, `r[7]`, `r[11]`, e `r[15]` precisam ter os bits do topo limpos, isto é, ser menor que 16
- `r[4]`, `r[8]`, e `r[12]` precisam ter os últimos 2 bits limpos, isto é, divisíveis por 4.

O processo de adequar `r` a estes requerimentos é chamado de *clamp*.

Após a divisão da chave e feito o clamp em `r`, a mensagem é quebrada em blocos de 16 bytes. Os blocos, então, passam a ser tratados como números no padrão little-endian. Então, 1 bit é adicionado ao topo, o que é equivalente a somar $2^{128}$ à um bloco de 16 bytes. Para o último bloco, caso seja menor de 16 bytes, pode ser qualquer potência de 2 que é divisível por 8, desde $2^{120}$ até $2^8$. Caso o último bloco, após a soma for menor que 17 bytes, ele é completado com zeros. O valor do bloco é então adicionado ao acumulador. O valor do acumulador então é multiplicado por `r` e é feita uma operação de módulo, com a constante `P`, ou seja, $ac = (ac \cdot r) \mod P$. Este procedimento é realizado para todos os blocos da mensagem. No final, o valor de `s` é adicionado ao acumulador, e os 32 bytes menos significativos são serializados para gerar a tag.

### 4.3.4 NH

`NH` é uma função de hash que é parte da família de de autenticadores UMAC, criada por Ted Krovetz [18]. Na proposta original, uma chave $K$ é utilizada como entrada, e também uma mensagem $M$, de tamanho em bytes divisível por 32 bytes. Uma saída $Y$, de 8 bytes é gerada.

O procedimento se inicia por inicializar $Y$ com 0 e por repartir $M$ e $K$ em partes de 4 bytes. Também é calculado $t$, que é o tamanho de $M$ em bytes, dividido por 4. Então, as seguintes operações são repetidas, variando-se $i$ de 0 até $t$:

$$Y = Y \overset{64}{+} ((M[i+0] \overset{32}{+} K[i+0]) \overset{64}{\times} (M[i+4] \overset{32}{+} K[i+4]))$$

$$Y = Y \overset{64}{+} ((M[i+1] \overset{32}{+} K[i+1]) \overset{64}{\times} (M[i+5] \overset{32}{+} K[i+5]))$$

$$Y = Y \overset{64}{+} ((M[i+2] \overset{32}{+} K[i+2]) \overset{64}{\times} (M[i+6] \overset{32}{+} K[i+6]))$$

$$Y = Y \overset{64}{+} ((M[i+3] \overset{32}{+} K[i+3]) \overset{64}{\times} (M[i+7] \overset{32}{+} K[i+7]))$$

Onde $\overset{64}{+}$ é a soma módulo $2^{64}$, $\overset{32}{+}$ representa a soma módulo $2^{32}$ e $\overset{64}{\times}$ representa a multiplicação módulo $2^{64}$.

## 4.4 AES-256-XTS versus Adiantum

Dada a complexidade de se compilar e configurar imagens do Sistema Operacional Android, e da necessidade de se possuir diversos dispositivos com processadores diferentes para conduzir um experimento de performance, os dados apresentados neste trabalho são aqueles fornecidos pela própria Google em seu blog e no artigo de publicação do Adiantum. O gráfico da Figura 7 mostra uma comparação de desempenho de Adiantum e AES-256-XTS, usando um processador ARM Cortex-A7, que não possui hardware dedicado para aceleração de criptografia. O resultado é que em média, Adiantum usa 10.6 ciclos de processamento para encriptar e decriptar um byte, em um sistema de setores de 4096 bytes. O gráfico da aponta para uma velocidade de encriptação de 24 MB/s para o AES-256-XTS, contra 112 MB/s do Adiantum, e uma velocidade de decriptação de 20 MB/s para AES-256-XTS e 112 MB/s para o Adiantum.

A falta de dados sobre desempenho em processadores topo de linha, com suporte à instruções específicas para criptografia impede uma comparação de escopo mais abrangente. Porém, se um usuário estiver com dúvidas quanto à velocidade e segurança, esse não será um empecilho caso realmente desejar usar o modo Adiantum provido nos dispositivos Android.

E caso o usuário tiver conhecimento técnico em matemática e criptografia, poderá consultar as provas rigorosas apresentadas nos artigos citados neste trabalho. Por se tratar de um sistema de código aberto, a implementação do sistema de criptografia do Android pode ser consultada em [13]. Vale citar que a parte relacionada a criptografia no sistema é escrita na linguagem C. Por se tratar de um sistema muito versátil e complexo, outras partes do sistema são escrita em diferentes linguagens de programação, como Java, C++ e Rust.

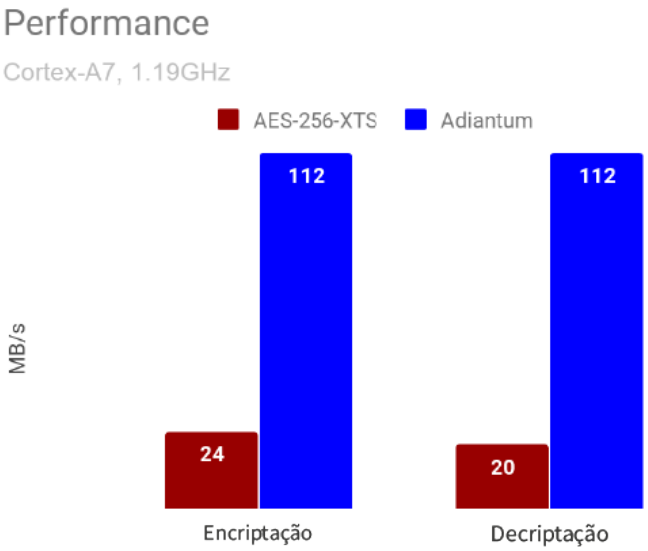


Figura 7: Comparação de desempenho de Adiantum e AES-256-XTS. Adaptado de Google Security Blog [15].

## 5 Conclusão

Este projeto expôs o autor a algoritmos e criptografia pela primeira vez durante a graduação, e a experiência foi de grande valia. A compreensão de conceitos básicos de criptografia é algo cada vez mais importante para o desenvolvedor de software moderno.

Dada essa importância, a descrição de algoritmos criptográficos pode por muitas vezes ser complexa, uma vez que estes geralmente costumam utilizar definições e operações matemáticas não muito triviais. Porém, este trabalho se propôs a explicar algumas das técnicas e algoritmos mais utilizados hoje em dia de uma forma simples e clara. É importante que os usuários estejam convencidos de que o software que usam é seguro, e estas descrições são um esforço para tornar isso possível.

O empenho dos desenvolvedores da Google para tornar a criptografia acessível à dispositivos de baixo custo e processamento é algo que pode ser visto com bons olhos. Não restrito à este aspecto, o algoritmo Adiantum já está disponível para usuários de computadores de mesa, através da versão 5.0 do Kernel Linux, ou mais recentes [6].

Por fim, cabe a análise de que um algoritmo novo não precisa ser inteiramente inédito. Adiantum é, na verdade, uma combinação de algoritmos que existem há pelo menos uma década. Cabe aos desenvolvedores terem a criatividade e ousadia de tentar algo diferente com elementos já existentes.

## Referências

[1] S. Arciszewski. Xchacha: extended-nonce chacha and aead-xchacha20-poly1305. Technical report, Internet-Draft draft-irtf-cfrg-xchacha-03. Work in Progress. Internet ...,

2020.

[2] ARM Developer. Arm cortex-a53 mpcore processor technical reference manual r0p3. `https://developer.arm.com/documentation/ddi0500/e/CJHDEBAF`. Acesso em 18/12/2021.

[3] A. Basu, A. Bhargav-Spantzel, and G. Pappas. Intel® aes-ni performance testing over full disk encryption, 2012.

[4] D. J. Bernstein. Extending the salsa20 nonce. In *Workshop record of Symmetric Key Encryption Workshop*, volume 2011. Citeseer, 2011.

[5] D. J. Bernstein et al. Chacha, a variant of salsa20. In *Workshop record of SASC*, volume 8, pages 3–5, 2008.

[6] E. Biggers. crypto: adiantum - add adiantum support. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=059c2a4d8e164dccc3078e49e7f286023b019a98`. Acesso em 18/12/2021.

[7] D. Coppersmith, D. B. Johnson, and S. M. Matyas. A proposed mode for triple-des encryption. *IBM Journal of Research and Development*, 40(2):253–262, 1996.

[8] P. Crowley and E. Biggers. Adiantum: length-preserving encryption for entry-level processors. *IACR Transactions on Symmetric Cryptology*, pages 39–61, 2018.

[9] J. Daemen and V. Rijmen. The rijndael block cipher: Aes proposal. In *First candidate conference (AeS1)*, pages 343–348, 1999.

[10] W. Diffie and M. Hellman. Exhaustive cryptanalysis of the nbs data encryption standard. *Computer*, 1977.

[11] GlobalStats. Operating system market share worldwide. `https://gs.statcounter.com/os-market-share`. Acesso em 18/12/2021.

[12] Google. Adiantum and hpolyc specification and test vectors. `https://github.com/google/adiantum`. Acesso em 18/12/2021.

[13] Google. Android source code - crypto. `https://cs.android.com/android/kernel/superproject/+/common-android-mainline:common/crypto/`. Acesso em 18/12/2021.

[14] Google. File-based encryption — android open source project. `https://source.android.com/security/encryption/file-based`. Acesso em 18/12/2021.

[15] Google. Introducing adiantum: Encryption for the next billion users. `https://security.googleblog.com/2019/02/introducing-adiantum-encryption-for.html`. Acesso em 18/12/2021.

[16] L. S. Hill. Cryptography in an algebraic alphabet. *The American Mathematical Monthly*, 36(6):306–312, 1929.

[17] Kingston Technology. O modo aes-xts block cipher é usado em pendrives criptografados da kingston. `https://www.kingston.com/br/solutions/data-security/xts-encryption`. Acesso em 18/12/2021.

[18] T. Krovetz, J. Black, S. Halevi, A. Hevia, H. Krawczyk, and P. Rogaway. Umac: Message authentication code using universal hashing. Technical report, RFC 4418, 2006.

[19] M. Liskov, R. L. Rivest, and D. Wagner. Tweakable block ciphers. In *Annual International Cryptology Conference*, pages 31–46. Springer, 2002.

[20] Y. Nir and A. Langley. Chacha20 and poly1305 for ietf protocols. *Internet Engineering Task Force*, 2015.

[21] Qualcomm Technologies, Inc. File based encryption. `https://www.qualcomm.com/media/documents/files/file-based-encryption.pdf`. Acesso em 18/12/2021.

[22] P. Rogaway. Nonce-based symmetric encryption. In *International workshop on fast software encryption*, pages 348–358. Springer, 2004.

[23] Seagate Technology. 128-bit versus 256-bit aes encryption. *Seagate Technology Paper*, 2008.

[24] C. E. Shannon. Communication theory of secrecy systems. *The Bell system technical journal*, 28(4):656–715, 1949.

[25] W. Stallings. *Criptografia e segurança de redes: princípios e práticas*. Pearson, 2014.

[26] Tecmundo. Adiantum: Google cria solução de criptografia para celulares mais baratos. `https://www.tecmundo.com.br/dispositivos-moveis/138644-adiantum-google-cria-solucao-criptografia-celulares-baratos.htm`. Acesso em 18/12/2021.