Universidade do Minho
Escola de Engenharia

Tiago Filipe Azevedo Oliveira

Verificação de Software Criptográfico de Elevado Desempenho

Janeiro de 2012

Universidade do Minho
Escola de Engenharia

Tiago Filipe Azevedo Oliveira

Verificação de Software Criptográfico de Elevado Desempenho

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação de
Doutor Manuel Bernardo Barbosa

Janeiro de 2012

DECLARAÇÃO

Nome

_____

Endereço electrónico: _____ Telefone: _____ / _____

Número do Bilhete de Identidade: _____

Título dissertação □ / tese □

_____

_____

_____

Orientador(es):

_____

_____ Ano de conclusão: _____

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

_____

Universidade do Minho, ____ / ____ / _____

Assinatura: _____

Agradecimentos

Um agradecimento especial vai para o orientador desta dissertação, Doutor Manuel Bernardo Barbosa, pela dedicação, disponibilidade, suporte e sugestões dadas ao longo de todas as etapas. Tal contribuiu de forma muito positiva para o alcançar dos objectivos propostos.

Durante a realização desta dissertação foi-me atribuída uma Bolsa de Iniciação Científica (BIC) no âmbito do projecto SMART, financiado pelo programa ENIAC JU (GA 120224).

Também gostaria de agradecer à Barbara Vieira, que contribuiu com sua disponibilidade e conhecimentos numa etapa importante para a conclusão da dissertação.

Para finalizar, uma palavra de gratidão à família e amigos por todo o apoio.

Abstract

The protection of data that circulates on the Internet should always be a matter of concern, and security properties such as confidentiality, integrity and authenticity should be ensured whenever needed using cryptographic techniques. However, there is a non negligible computational cost associated to the use of such techniques. Therefore it is necessary to invest in optimizations using, for instance, programming languages that operate at the `assembly` level.

The aim of this dissertation is to present a method which allows static verification of cryptographic code developed in a low level language called `qhasm`. It consists in translating `qhasm` code into a semantically equivalent `C` code, in order to assure that all the verifications performed in `C` can be transposed back to the original `qhasm` code.

Keywords: cryptography, optimization, `qhasm`, `assembly`, `C`, verification, translation;

Resumo

A protecção dos dados que circulam na Internet deve ser alcançada, sempre que se justifique, ao nível da confidencialidade, integridade e autenticidade, com recurso a técnicas criptográficas. Contudo, está associado ao uso deste tipo de técnicas um custo computacional não negligenciável. Por isso, é necessário investir em optimizações recorrendo, por exemplo, a linguagens que se situam no nível do `assembly`.

O objectivo desta dissertação consiste em apresentar um método que permita a verificação estática de código criptográfico desenvolvido numa linguagem de baixo nível denominada `qhasm`. Este método consiste em traduzir código `qhasm` para código `C`, semanticamente equivalente, de forma a que todos os resultados da verificação sobre o código `C` traduzido possam ser transpostos para o código `qhasm` original.

Palavras-chave: criptografia, optimização, `qhasm`, `assembly`, `C`, verificação, tradução;

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Contextualização

Com a evolução da Internet e a massificação das novas tecnologias existem cada vez mais tarefas quotidianas que podem ser realizadas através da rede. Muitas dessas tarefas implicam, por exemplo, o envio de dados pessoais através de um canal de comunicação desprotegido ou mal protegido.

A protecção de um sistema é desta forma essencial. Contudo, aplicar as técnicas necessárias para garantir a protecção pretendida leva a diversas contrapartidas: sistemas com menos capacidade de resposta ou utilizadores insatisfeitos, por exemplo. A relação negativa de custo-benefício é também um impedimento bastante frequente.

Um dos componentes deste conceito que é a protecção, ou segurança, são os algoritmos criptográficos. Estes consistem na implementação das diversas técnicas criptográficas existentes e podem garantir, por exemplo, confidencialidade, integridade, autenticidade e não repúdio [23].

Contudo, o custo computacional deste tipo de algoritmos não é negligenciável em grande parte dos casos. Dependendo do contexto, ou seja, capacidade de processamento disponível e volume de dados a processar, este factor poderá ser mais ou menos grave.

Para agravar o problema do custo computacional associado a este tipo de técnicas, estas devem ainda ser implementadas adoptando estratégias que visam reduzir vulnerabilidades contra ataques por canais subliminares, $side\text{-}channel\ attacks$ [26].

Esta classe de ataques tira partido de lacunas presentes na implementação física das técnicas, com o objectivo de adquirir informação sobre os parâmetros secretos utilizados. O estudo da variância do tempo de execução de um algoritmo, $timing\text{-}attack$, é um exemplo.

Para contornar ataques deste tipo são frequentemente adicionadas instruções sem sentido lógico no código fonte, servindo apenas para igualar o peso computacional de diferentes ramificações. A estrutura de controlo de fluxo `if-else` é um exemplo em que frequentemente isso acontece.

A optimização de algoritmos criptográficos é limitada por esta classe de ataques pois, se realizada inadequadamente, pode introduzir vulnerabilidades neste contexto.

O `qhasm` [4], uma linguagem de baixo nível próxima do `assembly`, aparece como uma alternativa para a produção de código criptográfico. Utilizando esta linguagem é possível obter implementações simultaneamente seguras e com bom desempenho. No entanto, a programação com linguagens de baixo nível está mais sujeita a erros de implementação e é, sobretudo, difícil de validar.

O objectivo desta dissertação consiste em avaliar uma estratégia de verificação para a linguagem `qhasm` de forma a contribuir para a resolução deste problema.

1.2 Contribuições

Apresentam-se de seguida as principais contribuições desta dissertação:

- Estudo e apresentação detalhada da linguagem de programação `qhasm`;
- Definição e implementação de uma tradução de `qhasm` para `C` que pode ser utilizada para a compreensão de programas `qhasm`, através de depuração, por exemplo. A mesma tradução permite a portabilidade de código `qhasm`;
- Definição, implementação e validação de uma abordagem que permite a verificação de software criptográfico implementado em `qhasm`.

1.3 Estrutura da Dissertação

O capítulo 2 aborda o desenvolvimento criptográfico de elevado desempenho. O capítulo 3 descreve a linguagem de programação `qhasm` no contexto da arquitectura de computador `x86`. O capítulo 4 faz um levantamento das propriedades que se pretendem para o código `C` traduzido de código `qhasm`. De seguida apresenta o método desenvolvido para realizar essa tradução. No capítulo 5 é discutida a aplicabilidade prática do método de tradução elaborado, sendo este posto à prova com uma implementação em `qhasm` do AES [5], [9]. Para finalizar, no capítulo 6 são apresentadas as conclusões desta dissertação.

Capítulo 2

Software Criptográfico de Elevado Desempenho

Este capítulo tem como objectivo apresentar uma visão geral sobre o desenvolvimento de software criptográfico de elevado desempenho. Introduz e relaciona três componentes deste domínio: as linguagens de programação, as preocupações que devem existir na implementação de primitivas criptográficas no contexto do desempenho e da resistência a ataques realizados por canais subliminares e, por último, a verificação estática destes algoritmos.

A secção 2.1 apresenta uma visão geral sobre diferentes linguagens programação que podem ser utilizadas no domínio da criptografia. A secção 2.2 aborda com mais detalhe os ataques realizados por canais subliminares, $side\text{-}channel\ attacks$. A secção 2.3 aborda alguns métodos de optimização para este contexto. A secção 2.4 apresenta os conceitos fundamentais da verificação estática, sendo orientada às ferramentas e técnicas de verificação utilizadas nesta dissertação.

2.1 Linguagens de Programação

Os tipos de linguagens tipicamente utilizados para implementar software criptográfico são os seguintes:

- DSL, ou *Domain-specific language*, consiste numa linguagem de programação orientada a um domínio particular. No domínio da criptografia o CAO [2] e o Cryptol [20] são exemplos de linguagens DSL;
- Linguagens genéricas tal como o `C` ou o `C++`;
- Linguagens de baixo nível tal como o `qhasm` ou `assembly`.

As linguagens específicas para o domínio da criptografia tem como principal vantagem a simplificação da especificação de um algoritmo criptográfico, pois dispõem de instruções e construções apropriadas para o efeito. No caso das linguagens apresentadas, estas são também suportadas por ferramentas de verificação que simplificam o processo de validação dos algoritmos desenvolvidos. Tipicamente, existe a possibilidade de, com base nas especificações desenvolvidas neste tipo de linguagens, gerar o código respectivo para linguagens genéricas, tal como o `C` ou `C++`.

Por outro lado, dado que o nível de abstracção em linguagens genéricas como o `C` é menor, é possível utilizar diferentes técnicas para melhorar o desempenho. A linguagem de programação `C` é também suportada por uma framework de verificação de software, denominada por `Frama-c`. A verificação de código `C` utilizando esta ferramenta será discutida na secção 2.4.

Num nível próximo da linguagem máquina encontra-se o `qhasm`, que será convenientemente detalhada no capítulo 3. Esta linguagem, dependente da arquitectura, permite tirar partido de instruções e propriedades muito específicas de cada arquitectura de computador. Como tal, é possível obter bons resultados ao nível do desempenho, dado que possibilita utilizar de forma explicita instruções inacessíveis em linguagens de mais alto-nível. Actualmente, a principal desvantagem de utilizar uma linguagem deste tipo consiste na falta de suporte por parte de ferramentas auxiliares. Quanto à portabilidade de código escrito em `qhasm`, inexistente, esta deve ser considerada um custo e não propriamente uma desvantagem.

Talvez a conclusão mais interessante que se pode retirar desta analise é que, à medida que se desce no nível de abstracção das linguagens, surgem mais possibilidades de aumentar o desempenho, contudo tendem a surgir dois problemas: a optimização não deve ser realizada de forma descuidada, por forma a que a implementação cumpra alguns requisitos no que diz respeito à resistência a vulnerabilidades de ataques do tipo *side-channel* e, em segundo lugar, a verificação estática tende a ser mais complexa de realizar.

2.2 Ataques Side-Channel

Uma das áreas de investigação, no domínio da criptografia, que tem vindo a ganhar importância são os ataques por canais subliminares ou $side\text{-}channel\ attacks$. Esta classe de ataques tem como sua principal característica a exploração de vulnerabilidades nas propriedades físicas de cada implementação. Por exemplo, tentam concluir algo sobre atributos privados de uma cifra monitorizando o seu tempo de execução.

A principal razão que justifica o crescimento desta nova área de investigação consiste na eficácia desta classe de ataques. São frequentes os casos em que a eficácia destes é superior quando comparados com os melhores ataques que apenas consideram o modelo teórico da técnica criptográfica. O artigo [26] consiste numa introdução completa a este tipo de ataques. Apresentam-se de seguida alguns tópicos introdutórios sobre esta área da criptografia.

Apesar desta classe de ataques ser bastante dispersa, é possível agrupá-los dentro das seguintes categorias: invasivos e não invasivos, activos e passivos. Enquanto que um ataque do tipo invasivo requer acesso físico ao dispositivo a atacar para, por exemplo, observar os dados que circulam entre os componentes deste, um ataque não invasivo apenas utiliza a informação libertada pelo dispositivo (de forma não intencional) para o exterior. Já na categoria dos ataques activos, estes tentam originar erros na execução de forma a que o dispositivo alvo liberte mais alguma informação. Os passivos, por oposição, limitam-se a observar o comportamento destes.

Exemplos de ataques não invasivos e passivos, já com algum estudo na literatura, são os que exploram tempos de execução [16], consumo eléctrico [17] e radiação electromagnética [1].

Contudo não existe uma fórmula mágica para evitar ataques $side\text{-}channel$. Existem, contudo, boas práticas que podem ser aplicadas na implementação e optimização dos algoritmos que podem minimizar a incidência de ataques desse tipo, nomeadamente, ao assegurar que, pelo menos, o tempo de execução não varie em função de determinados atributos privados.

Para isso é necessário ter em especial consideração situações que envolvam ramificações condicionais. No artigo [18] é descrito um erro na implementação do Rijndael, AES [9], que consistia no uso de uma ramificação condicional na operação `MixColumn`. Isso traduzia-se numa vulnerabilidade, pois o tempo de execução variava indevidamente.

No que diz respeito à implementação do AES, esta tem sido estudada intensivamente nos últimos anos e tem sido demonstrado é propensa a ataques $side\text{-}channel$, direccionados sobretudo a variações do tempo de execução. Os artigos [24, 6] descrevem ataques a essa cifra e, o autor deste último, afirma que este tipo de ataque é sobretudo devido ao desenho do próprio AES. Para esta cifra por blocos, o artigo [19] propõe uma implementação de elevado desempenho para a arquitectura `amd64` com resistência a ataques temporais.

2.3 Técnicas de Optimização

Um dos objectivos de quem implementa primitivas criptográficas é que estas possuam um bom desempenho. Num contexto genérico, existem inúmeras alternativas de optimização, por exemplo: desenrolar ciclos, eliminar/reestruturar condições de ramificação, pré-calculo de valores, processamento paralelo, evitar chamadas a funções, alocar memória de forma continua e, num nível de abstracção mais baixo, tirar partido de determinadas instruções e recursos arquitectura de computador.

Relativamente à utilização de linguagens de baixo nível, tal como o `qhasm` ou `assembly`, casos de sucesso já foram apresentados na literatura. Por exemplo, no contexto da implementação eficiente de curvas elípticas foi já publicada uma implementação que tira partido das novas instruções da arquitectura `amd64` em [21]. Esta arquitectura é também frequentemente denominada por `x86-64`.

Outro exemplo, no contexto da implementação de cifras por blocos, é o caso do AES [5], que descreve um conjunto de optimizações para as diversas arquitecturas suportadas pela linguagem de programação `qhasm`.

Para além das optimizações ao nível do código podem ser consideradas outro tipo de estratégias. Por exemplo, a implementação AES utilizada como prova de conceito (ver secção 5.1), implementa uma especificação que permite juntar três das habituais fases numa sequência de procuras a tabelas.

Neste caso em concreto, existiu uma troca entre memória necessária para as tabelas de substituição (4096 bytes ao invés de 256 bytes) e velocidade. Nestes casos os recursos computacionais do dispositivo alvo devem ser analisados, por forma a garantir que as melhores decisões ao nível da implementação são tomadas.

Apesar desta abordagem implicar um maior investimento realizado, visto que é necessário possuir um conhecimento razoável sobre a arquitectura destino, geralmente tem excelentes resultados.

2.4 Verificação Estática

Esta secção tem como objectivo apresentar o *plugin* Jessie[1] [22] da ferramenta Frama-c [8]. O Jessie permite a verificação dedutiva de programas C com anotações ACSL [3]. Internamente o Jessie utiliza a linguagem e ferramentas disponibilizadas pela plataforma Why [12]. O Frama-c consiste num conjuntos de ferramentas de análise a código C.

Utilizando o Jessie, existem dois tipos de verificação que se podem realizar: verificação da *safety* e verificação funcional. A verificação da *safety* permite obter garantias sobre a validade de acessos a memória, *safety* de operações sobre inteiros e terminação de uma função. Estes três sub-conjuntos de provas são detalhados de seguida na secção 2.4.1. Já a verificação funcional permite obter garantias sobre a correcção de uma função. Por exemplo, para um dado conjunto de pré-requisitos, obter a garantia que determinada função produz os resultados esperados.

Para especificar as propriedades uma função, é utilizada a linguagem ACSL [3], presente na forma de comentários especiais ao longo do código C a verificar. O Jessie, ao receber código fonte com anotações ACSL, gera um conjunto de obrigações de prova. Essas obrigações de prova são depois submetidas a um *prover* que as tenta validar. Exemplos de *provers* são: z3 [10], Yices [11] e Alt-Ergo [7].

Relativamente às instruções C não suportadas pelo Jessie, estas devem ser evitadas na tradução de código de forma a garantir a compatibilidade com código C traduzido. As limitações relevantes para este contexto são:

- Instruções do tipo goto arbitrárias. Estas instruções apenas são suportadas caso o salto seja realizado para uma linha posterior;
- *Casts* de apontadores. Apenas são suportados *casts* entre apontadores de tipos inteiros.

[1] http://frama-c.com/jessie.html

Relativamente ao primeiro ponto, a definição de ciclos em `qhasm` usa instruções `goto` para linhas anteriores e, portanto, uma estratégia de tradução directa de `qhasm` para `C` não resultaria. É então necessário inferir os ciclos presentes no código `qhasm`, caso estes existam. O segundo ponto, limita a forma como são traduzidas instruções que realizam acessos a memória, pois torna-se necessário declarar um apontador para um tipo inteiro, por exemplo, e não um apontador para `void`.

2.4.1 Verificação da safety

Validade de acessos a memória Este sub-conjunto das provas de $safety$ tem como objectivo tirar conclusões sobre a validade dos acessos à memória realizados por uma função. Por exemplo, caso uma função aceda a posições de memória correspondentes a um vector que lhe foi passado por referência, serão geradas condições de verificação para cada um dos acessos. Estas garantem, se provadas com sucesso, que determinado acesso é sempre realizado sobre uma região de memória válida. Neste caso, um dos pré-requisitos, em forma de anotação, especifica qual o tamanho do vector.

Safety de operações sobre inteiros Este sub-conjunto das provas de $safety$ tem como objectivo gerar condições de verificação que permitam tirar conclusões sobre a $safety$ das operações aritméticas sobre inteiros, garantindo a ausência de divisões por zero e transbordamentos ($overflow$). Por exemplo, no contexto computacional, os inteiros possuem uma representação limitada. É necessário então assegurar, se pretendido, que não ocorrem operações cujos resultados não são possíveis de representar.

Terminação de uma função Este sub-conjunto de provas de $safety$ destina-se a gerar condições de verificação que, se provadas, garantem que uma dada função termina. Este tipo de verificação é útil quando existem ciclos dentro das funções. Tipicamente, o utilizador disponibiliza informação sobre as propriedades do ciclo presente, tal como o seu invariante e variante. Para que um ciclo termine o seu variante deve obrigatoriamente diminuir ao longo da execução do ciclo.

Capítulo 3

Linguagem de Programação qhasm

Este capítulo tem como objectivo apresentar uma análise da linguagem de programação `qhasm` no contexto da arquitectura de computador `x86`, focando-se nas instruções sobre inteiros. Pretende-se que a leitura deste capítulo introduza os tópicos principais da linguagem `qhasm` de modo a facilitar a compreensão do trabalho realizado. Esta análise é também justificada pela falta de detalhe, tanto do ponto de vista prático como funcional, que a documentação actualmente disponível apresenta.

A secção 3.1 apresenta os tópicos base sobre a arquitectura `x86`. A secção 3.2 apresenta a linguagem `qhasm` descrevendo as suas características e qual o contexto onde se insere. A secção 3.3 apresenta de forma detalhada os tipos disponibilizados pelo `qhasm`. Posteriormente é discutida a gestão das variáveis que persistem em registos. A secção 3.4 descreve as instruções de controlo de fluxo disponibilizadas pelo `qhasm` e são expostos alguns exemplos de estruturas desse tipo. Para terminar, a secção 3.5 apresenta uma análise sobre a estrutura típica de um ficheiro `qhasm` e compara a inicialização da $stack$ em `qhasm` com a inicialização em `C`, resultante da respectiva compilação.

3.1 Introdução à Arquitectura `x86`

Como o `qhasm` é uma linguagem de baixo nível, muito próxima do `assembly` é vantajoso apresentar alguns conceitos sobre a arquitectura escolhida para esta dissertação. Esta

secção introduz os tópicos que se consideram essenciais para a compreensão do trabalho realizado. Contudo, dado que este tópico é muito vasto, recomenda-se a consulta de [14] e de [15] para esclarecer questões técnicas sobre esta arquitectura.

No contexto da programação, uma arquitectura de computador pode ser vista como um conjunto de instruções, recursos e modo de funcionamento suportados por uma família de processadores. Uma extensão a uma arquitectura adiciona novas instruções e recursos à arquitectura base.

A arquitectura `x86` disponibiliza então um conjunto de instruções que alteram o estado dos recursos. Os recursos[1] são os registos do processador e a memória primária. De notar que os registos do processador possuem uma velocidade de leitura e escrita superior quando comparada com a memória primária.

A arquitectura `x86` disponibiliza formas distintas de utilizar esses recursos. No caso dos registos, estes são referenciados pelo nome. No caso da memória primária, esta é referenciada através de endereços. Um endereço da arquitectura `x86` aponta ao nível do byte, ou seja, cada unidade incrementada num endereço faz com que este aponte para o byte seguinte.

Relativamente à forma como os dados estão guardados na memória, é importante referir que a arquitectura `x86` é *little-endian*, ou seja, para o caso de um valor inteiro representado em 4 bytes, o primeiro byte deste é o menos significativo e o último byte o mais significativo. Existem outras arquitecturas que são *big-endian*, tal como a `ARM` ou `PowerPC`, em que o comportamento é inverso ao anterior.

Para flexibilizar a forma como se realizam os acessos a memória, existe também um conjunto de modos de endereçamento suportados que sumarizam os casos de acesso mais comuns.

As seguintes secções apresentam individualmente alguns dos componentes da arquitectura `x86`, nomeadamente: registos base, registos das extensões `MMX`[2] e `SSE`[3], modos de endereçamento disponíveis, modelo de memória utilizado e um sub-conjunto das instruções disponíveis em `x86`. Posteriormente, é apresentada a convenção de chamadas a funções `cdecl`.

[1] Apenas serão referidos os recursos relevantes para o caso de estudo.
[2] MultiMedia eXtension.
[3] Streaming SIMD Extensions

Para finalizar, é apresentada uma função em `C` e discutido o respectivo código `assembly`. De notar que a sintaxe utilizada é `AT&T`, ou seja, as instruções estão na forma `instrução origem, destino`.

3.1.1 Registos Disponíveis

Existem 2 grupos de registos nos quais estamos interessados, registos de uso geral e registos de uso especial. A tabela 3.1 apresenta o grupo de registos de uso geral. Todos eles possuem 32 bits.

Nome do Registo	Descrição
`eax`	Acumulador.
`ebx`	Endereços indirectos.
`ecx`	Contador.
`edx`	Guarda *overflow* de certas operações.
`esi`	Endereço origem.
`edi`	Endereço destino.
`ebp`	Posição base da *stack*.
`esp`	Posição do topo da *stack*.

Tabela 3.1: Registos de uso geral da arquitectura `x86`.

É também possível referenciar sub partes dos registos `eax`, `ebx`, `ecx` e `edx`. Os registos `ax`, `bx`, `cx` e `dx` referenciam os primeiros 2 bytes destes, respectivamente. Os registos `al`, `bl`, `cl` e `dl` referenciam o primeiro byte, respectivamente. Por último, os registos `ah`, `bh`, `ch` e `dh` referenciam o segundo byte destes, também respectivamente.

Apesar da tabela 3.1 indicar uma função para cada um dos registos, estes podem ser utilizados de forma alternada. Contudo, algumas instruções são mais eficientes usando o registo apropriado, e outras operam sobre um registo em específico [13].

O grupo de registos de uso especial contém dois registos, `eflags` (*extended flag*) e `eip` (*extended instruction pointer*). Estamos interessados no primeiro. O registo `eflags`, também com 32 bits, consiste na versão estendida do registo `flags`, com 16 bits, da arquitectura precedente à `x86`. Este registo tem como função guardar o valor das *flags* existentes. De notar que este registo não é utilizado de forma explícita.

Uma *flag* consiste num valor booleano e é representada por 1 bit. As duas *flags* relevantes para o caso de estudo são a *zero flag* e a *carry flag* guardadas nas posições 0 e 6, respectivamente, do registo `eflags`.

O valor das *flags*, presentes no `eflags`, é alterado pela execução de diversas instruções `assembly`, sendo a instrução `cmp` um exemplo. Esta instrução será detalhada na secção 3.1.4. O objectivo é permitir a construção de estruturas de controlo de fluxo, sendo que a instrução `cmp` tipicamente precede uma instrução de salto.

Para além dos registos base desta arquitectura anteriormente descritos, as extensões `MMX` e `SSE` acrescentam novos registos. Estes são apresentados na tabela 3.2.

Extensão	Registos	Bits
MMX	mm0, mm1, mm2, mm3, mm4, mm5, mm6, mm7	64
MMX	st0, st1, st2, st3, st4, st5, st6, st7	80
SSE	xmm0, xmm1, xmm2, xmm3, xmm4, xmm5, xmm6, xmm7	128

Tabela 3.2: Registos das extensões `MMX` e `SSE` da arquitectura `x86`.

Os registos `mm` e os registos `st` partilham o mesmo espaço, pelo que não podem ser simultaneamente utilizados no decorrer da mesma função. Os registos `mm` são suportados por instruções que permitem realizar operações sobre múltiplos inteiros de uma só vez. Dependendo do tamanho do inteiro, podem ser aplicadas operações em 2, 4 ou 8 inteiros, caso estes sejam representados em 32, 16 ou 8 bits.

Os registos `st` tem como objectivo guardar números representados em vírgula flutuante e a sua capacidade é de 80 bits, ou seja, 10 bytes. As instruções que suportam estes registos tipicamente trabalham com eles em modo *stack*. Mais concretamente, algumas operações que requerem 2 operandos utilizam por defeito o registo `st0` e o outro registo deve ser especificado. O resultado é deixado no registo `st0`.

Os registos `xmm` são similares aos registos `mm`, contudo possuem o dobro da capacidade. Estes registos não partilham o seu espaço com outro conjunto de registos.

As instruções correspondentes aos registos presentes na tabela 3.2 não serão abordadas na secção 3.1.4, uma vez que não trazem mais valia para este caso de estudo. Estes registos serão novamente discutidos na secção 3.3.1.

3.1.2 Modos de Endereçamento

A arquitectura x86 disponibiliza um conjunto de modos de endereçamento. Estes são úteis, por exemplo, quando é necessário aceder a posições de memória de forma indexada. Este tipo de operação possui 4 parâmetros e a sua forma é a seguinte:

```
1 offset_número rico(endereço_base,offset_registo,escalar_multiplicativo
    )
```

Exemplificando: assumindo que o registo ebp contém o endereço base da posição de memória a ser acedida e edx o $offset$, apresentam-se os seguintes exemplos, cada um acompanhado de uma descrição em notação da linguagem C.

```
1 (%ebp)            ; *%ebp
2 4(%ebp)           ; *(%ebp + 4)
3 4(%ebp, %edx)     ; *(%ebp + 4 + %edx)
4 4(%ebp, %edx, 5) ; *(%ebp + 4 + (%edx * 5))
5 (%ebp, %edx, 5)   ; *(%ebp + (%edx * 5))
```

3.1.3 Modelo de Memória Utilizado

Tal como referido anteriormente, um dos recursos disponíveis numa arquitectura consiste na memória primária. Essencialmente existem duas áreas de memória que nos interessam: a memória correspondente à $stack$ e a memória alocada por chamadas a primitivas de alocação pertencentes ao sistema operativo, por exemplo, a função malloc em C.

A $stack$ consiste numa zona de memória destinada aos seguintes propósitos: guardar as variáveis locais de funções, suportar a passagem de argumentos e zona de auxílio à gestão de registos e controlo de execução. A $stack$ tem 2 registos a si associados, ebp e esp, endereço de base e endereço do topo da $stack$, respectivamente. A zona entre estes 2 registos é frequentemente denominada por $stack\text{-}frame$ da função.

À medida vão sendo invocadas funções, a *stack* vai aumentado o seu tamanho, visto que é necessário colocar novos argumentos, novas variáveis locais e o endereço de retorno para a função invocadora. De forma similar, assim que funções terminam a sua execução, o tamanho da *stack* diminui. A figura 3.1 apresenta um exemplo de uma *stack-frame* de uma função.

Relativamente ao espaço alocado por `malloc`, por exemplo, este encontra-se numa secção diferente da memória. Pode no entanto ser referenciada da mesma forma.

3.1.4 Instruções `x86`

São agora apresentadas algumas instruções `assembly x86`. As instruções serão descritas de forma sucinta. Mais detalhes sobre estas podem ser encontrados em [15]. Os termos `oper1` e `oper2` referem-se, respectivamente, a operando 1 (origem) e operando 2 (destino).

`movl oper1, oper2` Instrução utilizada para copiar valores de 32 bits. Tipicamente os operandos podem ser os registos presentes na tabela 3.1, endereços de memória (discutidos na secção 3.1.2) ou constantes. De notar que as constantes apenas fazem sentido no primeiro operando.

`movzbl oper1, oper2` Copia 8 bits do valor de `oper1` e não faz extensão de sinal, ou seja, os restantes bits são colocados a `0`. Os operandos são similares aos descritos anteriormente para a instrução `movl`.

`leal oper1, oper2` Copia o endereço de memória dado por `oper1` para `oper2`. O operando `oper1` encontra-se na forma discutida na secção 3.1.2. O segundo operando é um dos registos presentes na tabela 3.1. Frequentemente esta instrução é utilizada para realizar cálculos numéricos devido à sua flexibilidade.

`addl oper1, oper2` Soma os valores de `oper1` e `oper2`. O resultado é armazenado em `oper2`. Os tipos de operandos suportados são similares aos da instrução `movl`.

`subl oper1, oper2` Subtrai o valor do `oper1` a `oper2`. O resultado é armazenado em `oper2`. Os tipos de operandos suportados são similares aos da instrução `movl`.

`pushl oper1` Copia para o topo da *stack* o valor dado por `oper1` e decrementa 4 unidades o valor do registo `esp`. No caso da operação realizada ser `pushl %esp`, é colocado no topo da *stack* o valor do registo `esp` antes da subtracção.

`popl oper1` Copia o valor presente no topo da *stack* para `oper1`. Incrementa 4 unidades ao valor do registo `esp`.

`jmp oper1` Realiza um salto incondicional para a *label* indicada em `oper1`.

`jb oper1` Instrução *jump if below*. No caso da *carry flag* possuir valor 1, ou seja, encontra-se activa, então o salto para a *label* indicada em `oper1` é realizado.

`leave` Reinicializa o estado dos registos `esp` e `ebp`. Função equivalente à execução das seguintes instruções: `movl %ebp, %esp` e `pop %ebp`.

`ret` Esta instrução é a última executada numa função e devolve o controlo da execução à função invocadora, identificada pelo endereço de retorno. No momento de invocação desta instrução o endereço de retorno deve estar presente no topo da *stack*.

3.1.5 Convenção de Chamadas a Funções

Uma convenção de chamadas a funções define um conjunto de regras que devem ser seguidas pela função invocadora e pela função invocada. A convenção de chamadas a funções `cdecl`, assumida como standard pelo `gcc`[4], define as seguintes regras:

[4] GNU Compiler Collection

- Apenas os valores dos registos `eax`, `ecx` e `edx` podem ser alterados pela função invocada, sendo que os restantes devem permanecer inalterados do ponto de vista da função invocadora;
- A função invocadora é responsável por retirar da $stack$ os argumentos passados a uma função, após esta retornar. Isto permite a chamada de funções sem passar todos os argumentos;
- Os valores são retornados no registo `eax`, ou no caso de ser um número real representado em vírgula-flutuante no registo `st0`;
- Os argumentos da função são colocados na $stack$ da direita para a esquerda.

3.1.6 Exemplo prático `x86`

De forma a completar esta pequena introdução à arquitectura `x86` apresenta-se um exemplo prático. O objectivo é definir uma simples função `C` e discutir o `assembly` gerado pela respectiva compilação. Considere-se a seguinte função `C` que calcula a soma dos elementos de um `array` e coloca o resultado em `result`.

```
1  void sum(unsigned char* array, unsigned int len,
2           unsigned int *result)
3  {
4     unsigned int i = 0;
5     *result = 0;
6     while(i < len)
7     {
8        *result += array[i];
9        i++;
10    }
11 }
```

O `assembly` gerado pelo compilador `gcc`[5] para o exemplo apresentado é o seguinte.

[5] Versão 4.5.2 i686-linux-gnu

```
1  sum:
2     pushl %ebp                ; guardar o %ebp na stack ;
3     movl  %esp, %ebp          ; copiar o valor de %esp para %ebp ;
4     subl  $16, %esp           ; subtrair 16 ao %esp ;
5     movl  $0, -4(%ebp)        ; i = 0 ;
6     movl  16(%ebp), %eax      ; copiar &result para %eax ;
7     movl  $0, (%eax)          ; *result = 0 ;
8     jmp   .L2                 ; saltar para L2 ;
9  .L3:                         ; início do ciclo while ;
10    movl  16(%ebp), %eax      ; copiar &result para %eax ;
11    movl  (%eax), %edx        ; copiar *result para %edx ;
12    movl  -4(%ebp), %eax      ; copiar i para %eax ;
13    movl  8(%ebp), %ecx       ; copiar &array para %ecx ;
14    leal  (%ecx,%eax), %eax   ; copiar endereço (array+i) para %eax ;
15    movzbl   (%eax), %eax     ; copiar array[i] para %eax ;
16    movzbl   %al, %eax        ; não altera o valor de %eax ;
17    addl  %eax, %edx          ; soma array[i] com *result ;
18    movl  16(%ebp), %eax      ; copiar &result para %eax ;
19    movl  %edx, (%eax)        ; actualiza *result ;
20    addl  $1, -4(%ebp)        ; i ++ ;
21 .L2:
22    movl  -4(%ebp), %eax      ; copia i para %eax ;
23    cmpl  12(%ebp), %eax      ; compara len com i ;
24    jb .L3                    ; se i < len salta para L3 ;
25    leave                     ; %ebp = %esp, %ebp = antigo %ebp ;
26    ret                       ; terminar execução da função ;
```

Dado que o exemplo do código `assembly` apresentado contém os comentários necessários, realçam-se apenas os aspectos mais relevantes.

Tipicamente, a primeira tarefa realizada por uma função consiste na inicialização da *stack-frame*, linha 2 até 4 do anterior bloco de código. Tem como objectivo colocar o `ebp`, *base pointer*, numa posição adequada para referenciar tanto os argumentos passados à função como as variáveis locais. Ao valor de `esp`, *stack pointer*, são subtraídas 16 unidades de

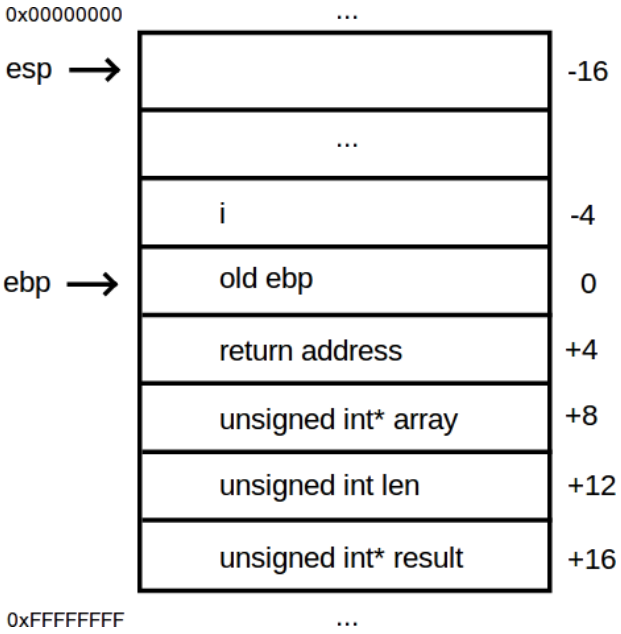

Figura 3.1: Exemplo de uma stack-frame.

modo a reservar espaço para as variáveis locais. Assim, posteriores operações `push` não reescrevem o valor das variáveis locais.

A figura 3.1 apresenta o estado da *stack-frame* da função para a linha 5 do código `assembly`. A *stack-frame* contém os argumentos da função, endereço de retorno, antigo valor do registo `ebp` para posteriormente ser restaurado e, por último, as variáveis locais. O endereço de retorno é colocado na *stack* em consequência da instrução `call`, utilizada para invocar uma função.

3.2 Introdução à Linguagem qhasm

Esta secção tem como objectivo principal fornecer uma primeira visão sobre a linguagem de programação `qhasm` descrevendo para tal as suas principais características.

O `qhasm` é uma linguagem de programação dependente da arquitectura de computador e, portanto, próxima do `assembly`. O seu objectivo é oferecer uma alternativa válida para a produção de código de elevado desempenho, permitindo tirar o melhor partido de cada plataforma/arquitectura. É sobretudo direccionada aos problemas existentes no contexto criptográfico.

As arquitecturas de computador suportadas pelo qhasm são as seguintes: x86, sparc, ppc32-linux, ppc32-macos e amd64. Esta dissertação, tal como já foi mencionado, incide no sub-conjunto das operações sobre inteiros da arquitectura x86.

A linguagem de programação qhasm, ao estar definida praticamente ao nível do assembly, permite tirar partido de instruções e comportamentos muito específicos. Numa linguagem multi-plataforma, tal como o C, não existe esse nível de controlo.

Apesar das ferramentas de compilação disponíveis alcançarem bons resultados na optimização de programas, não conseguem ainda explorar convenientemente essa margem. Um exemplo que ilustra esta afirmação são os recordes de velocidade atingidos numa implementação do AES [9] utilizando a linguagem de programação qhasm [5].

Realizando uma análise mais prática, apresentam-se as particularidades de funções implementadas em qhasm:

- Não invocam outras funções;
- Todo o espaço alocado em memória destina-se às variáveis locais;
- Não é possível retornar valores, sendo similares às funções com o tipo void em C;
- Os resultados calculados são persistidos em memória alocada ou pertencente à função invocadora.

As características acima descritas reforçam a especificidade do qhasm. Apesar de não ser uma linguagem genérica, a sua interacção com outras linguagens é possível. Contudo, a linguagem que invoca as funções escritas em qhasm deve utilizar a convenção de chamadas a funções cdecl.

Funções escritas em qhasm podem então ser utilizadas em código C, por exemplo. O processo de integração para este exemplo é descrito de seguida. Dado um ficheiro (com código qhasm) ao compilador de qhasm este gera código assembly. No código C, para importar as funções, a declaração do cabeçalho destas deve ser antecedida pela palavra extern. Na fase de compilação o ficheiro assembly também deve ser incluído na lista de ficheiros fonte.

Para finalizar, é de ressalvar que a linguagem de programação qhasm está ainda num estado de protótipo. Está planeada, ainda que sem data anunciada, uma completa reestruturação

da linguagem [4]. Apesar de existirem mais ferramentas para além do compilador, tal como o contador de ciclos de relógio e o analisador de intervalos de variáveis, estas não se encontram compatíveis com a versão actual do `qhasm`.

O desenvolvimento de uma nova ferramenta, que adicione valor ao `qhasm` e seja extensível a uma nova versão, pode contribuir de forma positiva para a adopção desta linguagem.

3.3 Tipos e Gestão das Variáveis

Ao longo desta secção serão introduzidos dois tópicos, os tipos das variáveis existentes no `qhasm`, secção 3.3.1, e também a gestão de variáveis que persistem em registos, secção 3.3.2. Como forma de contextualização, são também apresentados os tipos de variáveis disponibilizados pelo `C`.

Os tipos das variáveis do `qhasm` são um pouco diferentes quando comparados com outras linguagens de programação. Em `C`, os tipos mais recorrentes são caracteres, inteiros, números reais representados em vírgula flutuante e apontadores. A tabela 3.3 apresenta as diferentes declarações para cada um destes tipos em `C` assim como o espaço usualmente ocupado[6].

Tipo	**Declaração**	**Bytes**
Caracteres	`char`	1
Inteiros	`short int`	2
	`int`	4
	`long int`	4
	`long long int`	8
Vírgula Flutuante	`float`	4
	`double`	8
	`long double`	12
Apontadores	`void*`	4

Tabela 3.3: Tipos de variáveis em C.

Assumindo uma arquitectura de 32 bits, tal como a tabela 3.3 apresenta, as variáveis correspondentes a caracteres ocupam 1 byte e são declaradas como `char`. De notar que

[6]Em `C`, o espaço ocupado por uma variável pode ser diferente dos valores apresentados, diferindo em função das características da máquina. Apresentam-se os valores típicos para uma arquitectura de 32 bits.

caracteres podem ser encarados como inteiros. Tipos inteiros variam entre 2 e 8 bytes, e os números de vírgula flutuante entre 4 e 12 bytes. Um apontador para um destes tipos de dados, ou para qualquer estrutura de dados, ocupa 4 bytes. Tanto caracteres como inteiros podem ser declarados como sendo `unsigned`, ou seja, sem sinal.

As declarações de variáveis em `C` podem ser precedidas pela palavra chave `register`, que indica ao compilador que deve, preferencialmente, manter a variável num dos registos disponíveis, de forma a aumentar o desempenho. Contudo, não existe a garantia que a variável seja efectivamente mantida num registo. No `qhasm`, é possível obter essa garantia.

Quanto às declarações de variáveis em `qhasm`, estas não são auto-descritivas como em `C`, como poderá ser observado ao longo da próxima secção, 3.3.1.

3.3.1 Tipos das Variáveis

Em `qhasm`, existem dois conjuntos de tipos de variáveis, os que persistem em registos do processador e os que persistem em memória[7]. A tabela 3.4 contém os tipos que persistem em registos. Já a tabela 3.5 contém os tipos que persistem em memória.

Tipo	Bytes	Registos
`int32`	4	`eax`, `ecx`, `edx`, `ebx`, `esi`, `edi`, `ebp`
`int3232`	8	`mm0`, `mm1`, `mm2`, `mm3`, `mm4`, `mm5`, `mm6`, `mm7`
`int6464`	16	`xmm0`, `xmm1`, `xmm2`, `xmm3`, `xmm4`, `xmm5`, `xmm6`, `xmm7`
`float80`	10	`st0`, `st1`, `st2`, `st3`, `st4`, `st5`, `st6`, `st7`

Tabela 3.4: Tipos de variáveis qhasm em registos.

Tipo	Bytes
`stack32`	4
`stack64`	8
`stack128`	16
`stack256`	32
`stack512`	64

Tabela 3.5: Tipos de variáveis qhasm em memória.

[7] Assume-se, por razões de conveniência, que a palavra memória refere-se à memória primária de um dispositivo, tipicamente denominada por RAM, sendo que neste caso específico é referente à *stack-frame* de um programa.

Pela designação dos tipos, apresentados nas tabelas 3.4 e 3.5, constata-se que são orientados ao seu suporte físico e não a um tipo específico de dados.

Quanto à declaração das variáveis, esta é realizada linha a linha. O tipo pretendido precede o nome da variável a declarar. Nota importante: a declaração das variáveis é considerada global e, regra geral, o primeiro bloco de código num ficheiro `qhasm` consiste na declaração das variáveis.

A declaração dos argumentos das funções é também considerada global. Os argumentos são então comuns a todas as funções presentes no ficheiro. Tipicamente, são declarados com o tipo `stack32` e posteriormente redeclarados como `input`.

Outro caso que merece destaque, é a declaração das variáveis que suportam a salvaguarda dos valores dos registos, para posterior restauro no fim de execução da função. Pelas regras impostas pela convenção `cdecl`, a função apenas pode utilizar 3 registos, `eax`, `ecx` e `edx`. Para que o alocador de registos do `qhasm` tenha acesso aos restantes registos, `ebx`, `esi`, `edi` e `ebp`, é necessário criar uma cópia em memória. Para tal, é necessário declarar uma variável com o tipo `int32` com o nome do registo e redeclarar a variável como `caller`. É também necessária uma variável do tipo `stack32` para guardar, e posteriormente restaurar, o valor.

O seguinte bloco de código ilustra os casos discutidos, apresentado a declaração de argumentos e variáveis. No que diz respeito às declarações para efeitos de salvaguarda dos valores de registos, é apenas apresentado um caso exemplo para o registo `ebx`.

```
1  int32 ebx
2  caller ebx
3  stack32 ebx_stack
4
5  int32 var1
6  int32 var2
7  stack32 var3
8
9  stack32 arg1
10 input arg1
```

Os tipos disponibilizados pelo `qhasm` são detalhados individualmente de seguida. São também fornecidos exemplos de instruções para cada um deles, retirados da documentação disponível [25].

Tipo `int32`

Tal como referido na tabela 3.4, o tipo `int32` persiste nos registos `eax`, `ecx`, `edx`, `ebx`, `esi`, `edi` e `ebp`. Como tal, este tipo possui uma capacidade de 4 bytes.

Na arquitectura `x86`, as operações lógicas, aritméticas e acessos a posições de memória utilizam os registos acima mencionados. Portanto, este tipo de dados é usado para conter valores inteiros, ou endereços de memória. A distinção entre os dois não é explicita nesta linguagem. Este facto adiciona um desafio à tradução de código `qhasm` para código `C`, como será discutido no capítulo 4. A tabela 3.6 apresenta exemplos de instruções `qhasm` que usam variáveis do tipo `int32`.

Instrução	**Entrada**	**Saída**	**Assembly**
`r = s`	`stack32 s`	`int32 r`	`movl s,r`
`r += s`	`int32 s, int32 r`	`int32 r`	`addl s,r`
`r = *(uint16 *)(s + n)`	`int32 s, immediate n`	`int32 r`	`movzwl n(s),r`

Tabela 3.6: Exemplos de instruções para o tipo ***`int32`****.*

De notar também que ao todo estão disponíveis apenas 7 registos pelo que é mais do que comum existirem casos em que estes não são suficientes para satisfazer a correcta execução da função. A secção 3.3.2 apresenta uma forma para lidar com este tipo de problema.

Tipo `int3232`

Este tipo de variável, tal como referido na tabela 3.4, persiste nos registos `mm0`, `mm1`, `mm2`, `mm3`, `mm4`, `mm5`, `mm6` e `mm7` e tem uma capacidade de 8 bytes.

Estes registos, e respectivas instruções que sobre eles operam, fazem parte do conjunto de instruções MMX [13]. Na actual versão do `qhasm` este tipo de variáveis é apenas suportado por duas instruções. Uma para copiar valores de variáveis com o tipo `int32` para

Instrução	Entrada	Saída	Assembly
`r = s`	`int3232 s`	`int32 r`	`movd s,r`
`r = s`	`int32 s`	`int3232 r`	`movd s,r`

Tabela 3.7: Exemplos de instruções para o tipo ***int3232****.*

variáveis com o tipo `int3232`. A outra consiste na operação inversa. Estas instruções são apresentadas na tabela 3.7.

Na prática, apesar deste tipo de variáveis possuir uma capacidade de 8 bytes, apenas são utilizados 4 bytes, capacidade das variáveis `int32`. Quando utilizadas deve ser invocada a instrução `emms` antes final da função. Servem essencialmente para gerir a escassez de registos associados a variáveis `int32`, como será descrito na secção 3.3.2.

Nota importante: não podem ser utilizadas variáveis deste tipo em simultâneo com variáveis `float80`, pois partilham o mesmo espaço.

Tipo `int6464`

Este tipo de variável, apresentado na tabela 3.4, persiste nos registos `xmm0`, `xmm1`, `xmm2`, `xmm3`, `xmm4`, `xmm5`, `xmm6` e `xmm7` e tem uma capacidade de 16 bytes.

Estes registos fazem parte da extensão SSE. São interessantes do ponto de vista prático pois a sua capacidade é 4 vezes superior quando comparados com o tipo `int32`, o que permite realizar operações aritméticas sobre 4 inteiros de forma simultânea. A tabela 3.8 apresenta dois exemplos de instruções para variáveis deste tipo.

Instrução	Entrada	Saída	Assembly
`r &= s`	`int6464 r, int6464 s`	`int6464 r`	`pand s,r`
`uint32323232 r += s`	`int6464 r, int6464 s`	`int6464 r`	`paddd s,r`

Tabela 3.8: Exemplos de instruções para o tipo ***int6464****.*

A primeira instrução apresentada na tabela 3.8 consiste na operação de "e" ao nível do bit. A segunda instrução apresentada soma 4 inteiros a outros 4 inteiros, cada um deles com 32 bits.

Tipo float80

Este tipo de variável é o único nesta linguagem de programação que se destina a armazenar valores de vírgula flutuante. São utilizados os registos `st0`, `st1`, `st2`, `st3`, `st4`, `st5`, `st6` e `st7` que integram no conjunto de instruções MMX, tal como o tipo de variável `int3232`. A tabela 3.9 apresenta dois exemplos de instruções para este tipo de variável.

Instrução	Entrada	Saída	Assembly
`r *= s`	`float80 r, float80 s`	`float80 r`	`fmulp s,r!pop`
`r += s`	`float80 r, float80 s`	`float80 r`	`faddp s,r!pop`

Tabela 3.9: Exemplos de instruções para o tipo ***`float80`****.*

As operações sobre este tipo de dados são baseadas num esquema de pilha. Por exemplo, a primeira instrução da tabela 3.9 reflecte no `assembly` uma instrução que multiplica o valor do registo `st(i)` pelo registo `st0`, coloca o resultado em `st(i)` e não remove o `st0` da pilha (`!pop`). Isto levanta outro tipo de preocupações no que diz respeito à escrita de código `qhasm` executável, na medida em que a abordagem a tomar para fazer a gestão eficiente deste tipo de variáveis pode seguir um esquema diferente de variáveis não baseadas num esquema de pilha, `int32` por exemplo.

Tipos `stack[32-512]`

Este tipo de variável persiste na *stack frame* da função e é similar a uma variável local de uma função `C`. O limite de espaço destinado a variáveis deste tipo está dependente do tamanho máximo permitido para uma *stack-frame* sendo este atributo dependente das configurações do sistema operativo. A tabela 3.10 apresenta alguns exemplos de instruções que utilizam este tipo de dados.

Instrução	Entrada	Saída	Assembly
`r = s`	`stack32 s`	`int32 r`	`movl s,r`
`r += s`	`int32 r, stack32 s`	`int32 r`	`addl s,r`
`uint32323232 r += s`	`int6464 r, stack128 s`	`int6464 r`	`paddd s,r`

Tabela 3.10: Exemplos de instruções para os tipos ***`stack32-512`****.*

Tal como as variáveis do tipo `int32`, variáveis com o tipo `stack32` podem conter valores inteiros ou apontadores, não existindo também neste caso distinção entre ambos os tipos.

Tipos `stack` com tamanho superior a 32 bits, podem ser encarados como vectores com o tamanho indicado.

3.3.2 Gestão das Variáveis

Os tipos de variáveis `qhasm` que persistem em registos, apresentam um problema no que diz respeito à sua racionalização, visto que existem poucos. A seguinte discussão aplica-se a variáveis com o tipo `int32`, detalhado na secção 3.3.1.

O problema é que apenas existem 7 registos que podem ser utilizados. Isto é resolvido em parte pelo compilador de `qhasm`, que não mapeia de forma estática uma variável num registo. A outra parte é resolvida pelo programador, que deve ter em atenção que o número de variáveis num estado activo para cada momento da execução, não ultrapassa o número de registos disponíveis.

Uma variável encontra-se num estado activo se o seu valor não pode ser descartado, num dado momento da execução, sem que isso altere a correcção. De outra forma, variáveis que não são alvo de uma operação de escrita antes da próxima leitura.

Numa função sem estruturas de controlo de fluxo, `if`, `if-else`, `while` ou `do-while`, garantir que não existem mais de 7 variáveis activas do tipo `int32` é relativamente fácil. Contudo, a presença deste tipo de estruturas introduz diferentes caminhos de execução para diferentes parâmetros. Daqui se retira que: para todos os caminhos que a execução possa tomar (inclusive os cíclicos) deve existir pelo menos uma forma de mapear as variáveis activas nos registos disponíveis.

É bastante comum existirem casos em que não é possível ao compilador de `qhasm` alocar todas as variáveis. Quando tal acontece, o código `assembly` gerado não compila, pois existe uma instrução `assembly` sem um dos operandos. Como o ficheiro `assembly` contém, em forma de comentários, a instrução `qhasm` respectiva e meta-informação junto de cada instrução `assembly`, é fácil detectar a origem do erro.

Para corrigir este tipo de erros, é necessário identificar qual das variáveis activas é menos utilizada e guardar o seu valor em memória, tipo `stack`, ou noutro conjunto de registos tal como o `int3232`. Deste modo um registo é libertado. De notar que é necessário restaurar o valor da variável escolhida antes da leitura desta acontecer.

Como nota final, é de ressalvar que a maximização do uso de registos aumenta o desempenho do código, pois existe menos latência na obtenção dos dados. Um bom trabalho nesta área compensa o investimento realizado.

3.4 Estruturas de Controlo de Fluxo

Esta secção tem como objectivo apresentar as instruções `qhasm` que suportam a implementação de estruturas de controlo de fluxo. São apresentados exemplos deste tipo de estruturas, `while`, `do-while`, `if` e `if-else`. É também apresentada a versão em `C` de cada um dos exemplos.

Dois dos exemplos, `do-while` e `if`, são retirados de uma implementação da cifra AES-CTR[8] 128 bits escrita em `qhasm`. As instruções `qhasm` usadas para programar este tipo de construções são apresentadas nas tabelas 3.11 e 3.12.

Instrução	Entrada	Flags avaliadas	Assembly
`f:`	`immediate f`		`._f:`
`goto f`	`immediate f`		`jmp ._f`
`goto f if=`	`immediate f`	`=`	`je ._f`
`goto f if!=`	`immediate f`	`=`	`jne ._f`
`goto f if unsigned>`	`immediate f`	`unsigned>`	`ja ._f`
`goto f if !unsigned>`	`immediate f`	`unsigned>`	`jbe ._f`
`goto f if unsigned<`	`immediate f`	`unsigned<`	`jb ._f`
`goto f if !unsigned<`	`immediate f`	`unsigned<`	`jae ._f`

Tabela 3.11: Instruções de salto.

A tabela 3.11 contém as instruções que permitem realizar saltos em `qhasm`, excluindo a primeira que consiste na declaração de uma *label*. No que diz respeito às instruções de salto, a segunda entrada da tabela 3.11 realiza o salto de forma incondicional e, todas as outras, consideram do valor da *flag* em causa. Na versão actual do `qhasm`, apenas são suportados saltos que tem como base a comparação entre inteiros sem sinal. A tabela 3.12, contém as instruções `qhasm` usadas para alterar o valor das *flags*.

[8] Advanced Encryption Standard - Counter Mode. A framework de testes que contém a implementação usada pode ser obtida em: http://cr.yp.to/streamciphers/timings.html.

Instrução	Entrada	Flags modificadas	Assembly
`r - s`	`int32 r, int32 s`	`=, unsigned>, unsigned<`	`cmp s,r`
`r - n`	`int32 r, immediate n`	`=, unsigned>, unsigned<`	`cmp $n,r`

Tabela 3.12: Instruções de teste e declaração de labels.

As instruções da tabela 3.12, realizam a comparação entre dois parâmetros de entrada. O primeiro é obrigatoriamente uma variável do tipo `int32` e o segundo pode ser uma variável com o mesmo tipo ou uma constante. Ao executar uma destas instruções as *flags* do `qhasm` são alteradas mediante o resultado da avaliação.

Flags qhasm	Descrição	Flags x86	Assembly
`=`	Saltar se igual (=)	`ZF == 1`	`je ._f`
`!=`	Saltar se diferente (!=)	`ZF == 0`	`jne ._f`
`unsigned>`	Saltar se maior (>)	`CF == 0 && ZF == 0`	`ja ._f`
`!unsigned>`	Saltar se menor ou igual (<=)	`CF == 1 || ZF == 1`	`jbe ._f`
`unsigned<`	Saltar se menor (<)	`CF == 1`	`jb ._f`
`!unsigned<`	Saltar se maior ou igual (>=)	`CF == 0`	`jae ._f`

Tabela 3.13: Flags `qhasm` em detalhe.

Quanto às *flags* da linguagem de programação `qhasm`, estas consistem numa abstracção das *flags* da arquitectura `x86`. A tabela 3.13 estabelece a relação entre as *flags* `qhasm` e as *flags* `x86`. De notar que `ZF` e `CF` correspondem a *zero flag* e *carry flag*, respectivamente. Estas *flags* são alteradas pela execução da instrução de teste da tabela 3.12, que deve preceder a instrução de salto, tabela 3.11.

3.4.1 Ciclos

Apresentadas as instruções que suportam a construção de estruturas de controlo de fluxo, são agora detalhados casos práticos de implementações de ciclos. O seguinte excerto de código consiste num ciclo do tipo `do-while`.

```
1 mainloop:
2    # ...
3                unsigned>? len - 0
4 goto done if !unsigned>
```

```
goto mainloop
done:
```

Numa primeira observação, o código anterior pouco se assemelha a uma estrutura do tipo `do-while` em `C`, mas contêm as propriedades necessárias de modo a ser considerado uma estrutura deste tipo: o código dentro do ciclo é executado pelo menos uma vez; no final é realizado um teste à variável `len` e, no caso de ser menor ou igual que `0`, o ciclo termina. No caso de `len` ser maior que `0` o ciclo continua. O código equivalente em `C` seria:

```
do{
   // ...
}while( len > 0 );
```

De forma similar, um ciclo do tipo `while` poderia ser definido da seguinte forma:

```
mainloop:
               unsigned>? len - 0
goto done if !unsigned>
   # ...
goto mainloop:
done:
```

E a implementação equivalente em `C` seria a seguinte:

```
while( len > 0){
   // ...
}
```

3.4.2 Condições de ramificação

A codificação de estruturas de controlo de fluxo do tipo `if` ou `if-else` segue a mesma lógica da implementação dos ciclos. O seguinte exemplo apresentado consiste numa implementação de uma estrutura `if`, retirado da implementação do AES em estudo. A título de curiosidade, o ciclo `do-while` anteriormente exposto está contido dentro do seguinte bloco de código.

```
                    unsigned>? len - 0
goto nothingtodo if !unsigned>
   # ...
nothingtodo:
```

Neste caso a variável `len` é testada e, caso o seu valor seja menor ou igual que `0`, é realizado um salto para a *label* `nothingtodo`. Assim, no caso do valor de `len` ser maior que `0` o código dentro da estrutura é executado. O código equivalente em `C` seria o seguinte:

```
if(len > 0){
   // ...
}
```

De forma similar, uma estrutura do tipo `if-else` poderia ser definida da seguinte forma:

```
                   unsigned>? len - 0
goto elselabel if !unsigned>
   # ...
goto done
elselabel:
   # ...
done:
```

E a implementação equivalente em C seria a seguinte:

```
if(len > 0){
   // ...
}else{
   // ...
}
```

3.5 Estrutura de um Ficheiro qhasm

Esta secção tem como objectivo detalhar qual a estrutura típica de um ficheiro com código qhasm. São também discutidas algumas diferenças entre código assembly gerado pelo compilador de qhasm e pelo compilador de C, gcc, no que diz respeito à inicialização da *stack* da função.

Relativamente à estrutura de um ficheiro qhasm, existem duas secções distintas: declaração de variáveis e declaração de funções. A declaração de variáveis já foi discutida na secção 3.3 pelo que passamos à declaração de funções qhasm. Considere-se o seguinte excerto de código qhasm:

```
stack32 arg1
stack32 arg2
input arg1
input arg2

int32 eax
caller eax
stack32 eax_stack

int32 c
int3232 c_stack

```

```
13 enter ECRYPT_process_bytes stackaligned4096 aes_constants
14     eax_stack = eax
15     #...
16     c = arg2
17     c_stack = c
18     #...
19     emms
20     eax = eax_stack
21     #...
22 leave
```

A instrução `enter` declara o início de uma nova função. O argumento obrigatório é o nome da função a declarar, neste caso `ECRYPT_process_bytes`. Os outros dois argumentos presentes são opcionais. O argumento `stackaligned4096` indica ao compilador de `qhasm` que deve alinhar a $stack$ a 4096 bytes. Isto provoca um ligeiro aumento de desempenho. Quanto ao terceiro argumento, `aes_constants`, consiste na indicação da zona de memória onde se encontram algumas constantes que são utilizadas no decorrer da função.

Após a declaração do início de função surgem as instruções que guardam os valores dos registos em variáveis locais. Os registos cujos valores são guardados são: `eax`, `ebx`, `esi`, `edi` e `ebp`. Relembrando a convenção de chamadas a funções `cdecl`, os registos que podem ser utilizados pela função invocada são os registos `eax`, `ecx` e `edx`. Contudo, existe uma razão para o registo `eax` ser guardado e será discutida de seguida. Daqui também se retira que todos os registos irão ser usados com a excepção do `esp`. Esta afirmação entra em conflito com o método de gestão de registos utilizado pelo `gcc`, em que o registo `ebp` é o endereço usado como referência (ver secção 3.1) para aceder aos argumentos e a variáveis locais.

Posteriormente é executado o código que leva a cabo o processamento desejado. O exemplo aqui apresentado, por questões de simplicidade resume-se a 2 instruções, linha 16 e 17. A primeira copia o valor de um argumento para uma variável `int32` e a segunda copia o valor da variável `c` para a variável `c_stack`, do tipo `int3232`. Tal como referido previamente, sempre que se utilizam variáveis com o tipo `int3232` é necessário invocar a instrução `emms` antes do final da função.

Por fim restauram-se os valores dos registos e é invocada a instrução `leave`. Esta instrução, que não corresponde à instrução `leave` em `assembly`, dá como terminada a execução da função. Após esta linha pode ser declarada uma nova função.

Duas notas: a ordem assumida para os argumentos das funções é a ordem pela qual as variáveis são redeclaradas como `input` e os argumentos são considerados globais e, como tal, comuns a todas as funções.

Analisa-se agora a forma como o compilador de `qhasm` gere a sua *stack*. Considere-se o seguinte excerto de código `assembly` correspondente ao código `qhasm` anteriormente apresentado.

```
1  ECRYPT_process_bytes:
2     mov %esp,%eax              ; copia valor de esp para eax ;
3     sub $aes_constants,%eax    ; subtrai $aes_constants a eax ;
4     and $4095,%eax             ; eax = eax % 4096 ;
5     add $224,%eax              ; eax += 224 ;
6     sub %eax,%esp              ; aloca espaço na stack ;
7
8     # qhasm: eax_stack = eax
9     movl %eax,0(%esp)          ; guarda eax no topo da stack ;
10
11    # qhasm: c = arg2
12    movl 8(%esp,%eax),%ebx     ; acede ao argumento 2 ;
13
14    # qhasm: c_stack = c
15    movd %ebx,%mm0             ; copia ebx para mm0 ;
16
17    # qhasm: emms
18    emms                       ; reinicia contexto MMX ;
19
20    # qhasm: eax = eax_stack
21    movl 0(%esp),%eax          ; restaura valor eax ;
22
23    # qhasm: leave
24    add %eax,%esp              ; restaura o valor de esp ;
25    ret
```

O código `assembly` gerado pelo compilador de `qhasm` é substancialmente diferente do `assembly` gerado pelo `gcc`, no que diz respeito à inicialização da $stack$. A instrução presente na linha 4 coloca em `eax` o valor a subtrair a `esp` para este ficar alinhado a 4096 bytes. Contudo, a este valor ainda é necessário somar o espaço necessário para as variáveis locais. Posto isto, o valor final é subtraído ao `esp` e a $stack$ encontra-se inicializada.

A principal diferença é que as variáveis locais passam a ser acedidas tendo como base o registo `esp`. Então, ao invés de se situarem logo após os argumentos, são colocadas a partir do cimo da $stack$. Este método implica a não existência de instruções `push`. Como o registo `eax` contém o valor subtraído a `esp`, este deve ser guardado para no final ser possível restaurar o `esp`. A principal vantagem desta abordagem passa pela disponibilização de mais um registo, `ebp`.

Capítulo 4

Tradução de qhasm para C

Este capítulo tem como objectivo apresentar uma estratégia de tradução de `qhasm` para `C`. Esta tradução deve preservar a equivalência semântica entre o código original `qhasm` e o código traduzido `C`. O objectivo é que as propriedades verificadas estaticamente no código `C` possam ser transpostas para o código `qhasm` original. O principal desafio consiste em contornar as várias limitações das ferramentas de verificação, sem que isso invalide a equivalência entre os dois programas. Ao longo deste capítulo serão utilizados excertos de código retirados de uma implementação `qhasm` do AES [5], *counter-mode*, que é apresentada nesta dissertação na secção 5.1.

A secção 4.1 define a noção de equivalência semântica entre dois programas no contexto aqui apresentado. De notar que a equivalência semântica aqui tratada refere-se exclusivamente ao contexto da implementação de algoritmos criptográficos de baixo-nível, tais como cifras por blocos e funções de hash. A secção 4.2 descreve a estratégia de tradução no contexto das instruções e tipos de variáveis. Para finalizar, a secção 4.3, apresenta a tradução de estruturas de controlo de fluxo.

4.1 Objectivos

Esta secção tem como objectivo apresentar uma primeira especificação dos objectivos pretendidos no contexto na tradução automática de código escrito em `qhasm` para código `C`.

A construção de um modelo que estabeleça contornos bem definidos sobre as propriedades do código traduzido é de todo relevante pois torna possível transpor os resultados da verificação realizada sobre código `C` traduzido para o código `qhasm` original.

A arquitectura que se pretende implementar é a seguinte: dada uma função `qhasm`, possivelmente com anotações `ACSL`, obter de forma automática uma função em `C`, com as anotações caso existam, e semanticamente equivalente. As propriedades verificadas numa função `C` traduzida podem ser assumidas como válidas na função `qhasm`.

O termo semanticamente equivalente pode introduzir ambiguidades, portanto avança-se com uma primeira definição informal. Considerando **O** como a função original, em `qhasm`, e **T** como a função traduzida, em `C`:

- Considera-se que uma determinada função **T** é semanticamente equivalente a uma função **O** se **T** possui o mesmo comportamento $input/output$.

No entanto, esta definição pode não ser satisfatória quando se tem como objectivo que os resultados da verificação de código `C` traduzido sejam transpostos para o código `qhasm` que lhe deu origem. É necessário que exista um mapeamento, de um para um, ao nível das variáveis, de forma a ser possível relacionar as mesmas entre `qhasm` e `C`. A garantia que a alteração das variáveis é realizada de forma equivalente deve ser também assegurada. O comportamento ao nível do controlo de fluxo deverá ser também preservado.

É necessário incrementar a anterior definição para satisfazer os novos requisitos. Considera-se que uma determinada função **T** é semanticamente equivalente a uma função **O** se:

1. Para cada variável em **O** existe uma variável equivalente em **T**, identificável de forma directa;

2. Para cada instrução em **O** existe uma ou mais instruções correspondentes em **T** de tal forma que, no final da execução desta(s), o estado das variáveis utilizadas, comuns a **O** e **T**, é o mesmo;

3. O processo de alteração de estado das variáveis deve preservar a semântica. Por exemplo, um $overflow$ em `qhasm` deve originar um $overflow$ em `C`;

4. Todas as condições de salto presentes em **O** são transpostas para **T** de tal forma que o fluxo de execução do código seja preservado.

4.2 Tipos de Variáveis e Tradução de Instruções

Como referido previamente, os tipos da linguagem de programação qhasm são orientados ao seu suporte físico e não a um tipo de dados específico. Por exemplo, uma variável do tipo int32, ou stack32, tanto pode conter um apontador como um inteiro. A informação sobre o tipo de uma variável não está na declaração deste, mas sim nas instruções em que é utilizado.

Considere-se as seguintes instruções qhasm, em que as variáveis in, b0 e z2 são do tipo int32.

```
b0 = *(uint8 *) (in + 0)
z2 = *(uint32 *) (in + 8)
```

Na primeira instrução, são copiados 8 bits a partir da posição de memória apontada por in para uma variável com 32 bits. Não é realizada a extensão do sinal, ou seja, os restantes bits são colocados a 0. Na segunda instrução, são copiados 32 bits a partir da posição de memória apontada por in, também para uma variável com 32 bits.

Para este par de instruções, e considerando as restrições da ferramenta Jessie, existem apenas duas alternativas de obter o mesmo comportamento em C:

```
unsigned char* in;
unsigned int b0, z2;

b0 = in[0];
z2 = *(unsigned int*)(in + 8);
```

```
unsigned int* in;
unsigned int b0, z2;

b0 = *(( (unsigned char*) in) +
   0);
z2 = in[2];
```

Em qhasm é muito comum os apontadores serem utilizados desta forma. Por vezes são utilizados como apontadores para caracteres, outras vezes como apontadores para inteiros. As razões que justificam esta forma de utilização estão relacionadas com o desempenho. É absolutamente necessário contornar este problema na tradução.

Ao observar com cuidado as duas alternativas de tradução apresentadas, conclui-se que a tradução para `C` de cada instrução `qhasm` está directamente relacionada com o tipo escolhido para a variável `in`. Já os tipos das variáveis `b0` e `z2` não interferem com a tradução a escolher.

Daqui se retira que, para cada instrução `qhasm`, pode existir uma ou mais traduções para `C` e cada tradução requer que as suas variáveis possuam certos tipos. Então, para escolher a tradução a utilizar para cada instrução `qhasm` é necessário fixar os tipos das variáveis.

O anexo B contém uma lista de instruções `qhasm` e, para cada uma delas, indica quais as traduções que estão disponíveis. Cada possível tradução de uma instrução `qhasm`, especificada nesse anexo, indica também quais os tipos de variáveis que requer. De notar que os tipos de variáveis estão na forma de acrónimo. A lista dos acrónimos utilizados estão presentes no anexo A.

Voltando à discussão, o problema é o de decidir qual o tipo a atribuir a cada variável de uma função `qhasm`, dado que as traduções a utilizar estão dependentes da maneira como estas serão declaradas.

O caso mais simples acontece quando a variável, em todas as possíveis traduções para cada instrução `qhasm`, possui o mesmo tipo. Casos deste género são frequentes quando as variáveis são utilizadas como inteiros. Relativamente ao exemplo apresentado, as variáveis `b0` e `z2` possuem apenas um tipo possível.

O verdadeiro problema acontece quando a variável pode assumir mais do que um tipo na tradução. No exemplo anteriormente apresentado, esse é o caso da variável `in`. Considere-se um novo exemplo de código `qhasm`, em que todas as variáveis utilizadas são do tipo `int32`.

```
1 b0 = *(uint8 *) (in + 0)
2 z2 = *(uint32 *) (in + 8)
3 z3 = *(uint32 *) (in + 12)
4 z4 = *(uint32 *) (in + 16)
```

Considere-se agora as duas alternativas de tradução possíveis para o código `qhasm` apresentado, tal como anteriormente.

```
unsigned char* in;
unsigned int b0, z2, z3, z4;

b0 =  in[0];
z2 =  *(unsigned int*)(in + 8);
z3 =  *(unsigned int*)(in + 12);
z4 =  *(unsigned int*)(in + 16);
```

```
unsigned int* in;
unsigned int b0, z2, z3, z4;

b0 = *(unsigned char*)(in + 0);
z2 = in[2];
z3 = in[3];
z4 = in[4];
```

Ao contrário das alternativas de tradução expostas anteriormente em que, qualquer que fosse o tipo da variável `in`, existia sempre um $cast$ e um acesso de forma nativa, neste exemplo isso mudou. Para o caso agora apresentado existe uma clara tendência para o tipo da variável `in` ser `unsigned int*`, pois é mais vezes utilizada de forma nativa para este tipo.

Existem, contudo, casos em que é absolutamente indiferente optar por determinado tipo em detrimento de outro. O primeiro exemplo apresentado é um desses casos.

Resumindo os tópicos discutidos durante esta secção temos que:

- Uma instrução `qhasm` pode ter uma ou mais traduções para `C`;
- A instrução `C` a ser escolhida para substituir a original `qhasm` depende do tipo que será atribuído em `C` às variáveis que nela participam;
- No caso em que uma variável tem apenas um tipo possível é com esse que será declarada;
- No caso em que uma variável possui mais do que um tipo possível é necessário escolher entre os disponíveis o melhor deles.

4.2.1 Implementação

A forma como está implementado o algoritmo de tradução e inferência de tipos de dados segue os princípios anteriormente apresentados, ou seja, a tradução realizada depende dos tipos das variáveis. Numa primeira fase, cada instrução `qhasm` é relacionada com uma das entradas disponíveis num ficheiro de mapeamento (consultar anexo B), isto, claro, se existir a entrada correspondente à instrução.

Posteriormente são eliminadas todas as traduções incompatíveis entre si, ou seja, traduções que requerem que a variável seja declarada com um tipo que não é suportado por, pelo menos uma, possível tradução de cada instrução `qhasm` em que a variável participa. Isto exige que os ficheiros de mapeamento sejam bem construídos. De seguida são calculadas as variáveis que já possuem o tipo definido, ou seja, as que apenas possuem um tipo possível.

Após as fases anteriores é necessário calcular iterativamente quais os tipos a atribuir às variáveis que ainda não estão definidas. Então, em cada iteração é calculado o tipo com que uma variável será declarada. De notar que os tipos das variáveis dependentes destas serão também definidos. A dependência entre os tipos para duas variáveis é originada por operações de atribuição (consultar segunda entrada de mapeamento do anexo B).

Cada iteração é composta por 3 passos:

1. Escolher qual a variável à qual será atribuído um tipo. É escolhida a mais utilizada pois é mais influente;
2. Escolher qual o tipo correspondente à variável que pertence a mais possíveis traduções e fixá-lo. Em caso de empate, escolher o tipo nativo da variável, tal como foi discutido no último exemplo apresentado;
3. Eliminar traduções incompatíveis e actualizar os tipos das variáveis.

Quanto ao passo 2, assume-se que nas entradas de mapeamento a primeira tradução indicada é aquela que faria mais sentido utilizar. No caso da afirmação anterior não fazer sentido para uma dada instrução tal não cria problemas.

Para determinar o tipo nativo de uma variável, é necessário consultar a entrada de mapeamento (em termos genéricos) que mais associações tem com a variável. Dessa entrada, seleccionar o tipo que se encontra na primeira possível tradução.

Um dos requisitos para que a implementação funcione é que, para cada instrução `qhasm`, não exista mais do que uma possível tradução, definida no ficheiro de mapeamento, cujo conjunto de tipos exigidos para as variáveis seja igual.

O objectivo em mente ao definir esta estratégia foi ser suficientemente genérico para permitir uma maior flexibilidade na forma como são definidos os ficheiros de mapeamento e, consequentemente, realizadas as traduções.

4.3 Tradução de Estruturas de Controlo de Fluxo

Existem duas possíveis abordagens para traduzir instruções do tipo `goto`. A primeira abordagem consiste em traduzir de forma directa as instruções presentes na tabela 3.11, potencialmente originando problemas de compatibilidade com o `Frama-c` no caso dos ciclos. A segunda consiste em inferir e traduzir o código para estruturas de mais alto-nível.

Esta segunda abordagem apenas é bem sucedida se a implementação destas estruturas seguir o padrão assumido, o que geralmente acontece. Caso uma instrução de salto condicional `qhasm` não possa ser associada a uma construção mais alto-nível é utilizada a tradução directa dessa mesma instrução. Contudo, nesse caso, os objectivos podem não ser alcançados. A tabela 4.1 a tradução directa de instruções do tipo `goto`.

Condição de Teste	**Instrução** `qhasm`	**Instrução** `C`
	`goto f`	`goto f;`
`=? len-n`	`goto f if=`	`if(len == n) goto f;`
`=? len-n`	`goto f if!=`	`if(len != n) goto f;`
`unsigned>? len-n`	`goto f if unsigned>`	`if(len > n) goto f;`
`unsigned>? len-n`	`goto f if !unsigned>`	`if(len <= n) goto f;`
`unsigned<? len-n`	`goto f if unsigned<`	`if(len < n) goto f;`
`unsigned<? len-n`	`goto f if !unsigned<`	`if(len >= n) goto f;`

Tabela 4.1: Tradução directa de instruções de salto.

Assume-se que as estruturas de controlo de fluxo são definidas tal como apresentadas na secção 3.4. Um tradutor deverá então ser capaz de detectar a presença, em código `qhasm`, de 4 tipos de estruturas de controlo de fluxo: `do-while`, `while`, `if` e `if-else`. Na especificação das estruturas ao longo das próximas secções será usada a terminologia presente na tabela 4.2.

Designação	Descrição	Valores possíveis
L1/L2	Nome genérico para uma *label*.	n/a
CODE	Código escrito em qhasm.	n/a
TEST	Teste realizado antes de salto condicional.	=? r - s
		unsigned>? r - s
		unsigned<? r - s
CONDITION	Condição sobre a qual o salto é dado.	=
		!=
		unsigned>
		!unsigned>
		unsigned<
		!unsigned<

Tabela 4.2: Terminologia da especificação de estruturas de controlo de fluxo.

4.3.1 Ciclo do while

A sequência de instruções sobre a qual é assumida a definição de um ciclo deste tipo é a seguinte:

```
1 L1:
2    #CODE
3 TEST
4 goto L2 if CONDITION
5 goto L1
6 L2:
```

É de notar que o código do ciclo, identificado na caixa por #CODE, deve respeitar as seguinte restrições:

- Não deverão existir saltos, condicionais ou não, para *labels* que não se encontrem dentro do ciclo;

- No caso de existirem instruções de salto estas apenas podem referenciar a *label* L2. Tal caso possui comportamento equivalente à instrução break do C.

Um dos procedimentos da tradução deve incluir a negação da condição de teste, `CONDITION`, de modo a manter a equivalência semântica. As condições de teste a serem negadas estão presentes na tabela 4.1.

Por exemplo, instanciando a anterior definição de um ciclo `do while` num caso específico com `TEST` igual a `unsigned>?len-0`, `CONDITION` igual a `!unsigned>` e com base na tabela 4.1, a tradução obtida seria a seguinte:

```
do{
  //CODE
} while (!(len <= 0));
```

A condição é equivalente a `len > 0`.

4.3.2 Ciclo while

A sequência de instruções sobre a qual é assumida a definição de um ciclo deste tipo é a seguinte:

```
L1:
TEST
goto L2 if CONDITION
   #CODE
goto L1
L2:
```

Este tipo de ciclo é similar ao anterior, apenas podem existir saltos para a *label* `L2`. A condição de teste também deverá ser negada de modo a manter a equivalência semântica. Se se instanciar esta definição com os mesmos parâmetros do exemplo anterior a tradução ficaria:

```
while(len > 0){
   //CODE
}
```

4.3.3 Condição de ramificação if

A sequência de instruções sobre a qual é assumida a definição de uma condição de ramificação deste tipo é a seguinte:

```
TEST
goto L1 if CONDITION
   #CODE
L1:
```

Dentro do código da condição de ramificação, identificado na caixa por `#CODE`, não pode existir qualquer salto para fora da estrutura, nem para a *label* `L1`. Salvaguarda-se a possibilidade de existir um salto para uma *label* externa apenas se esta corresponder à *label* `L2` de um ciclo, caso este se encontre um nível acima. Por exemplo, no caso de existirem ciclos aninhados, o salto apenas poderá ser dado para a *label* `L2` do ciclo mais próximo, comportando-se assim de forma similar a uma instrução do tipo `break`. Considerando as parametrizações já usadas nos exemplos anteriores, a tradução deste tipo de estrutura ficaria:

```
if (len > 0)
   //CODE
L1:
```

4.3.4 Condição de ramificação if else

A sequência de instruções sobre a qual é assumida a definição de uma condição de ramificação deste tipo é a seguinte:

```
TEST
goto L1 if CONDITION
   #CODE
   goto L2
L1:
   #CODE
L2:
```

As restrições são idênticas à condição de ramificação `if` previamente apresentada, com a chamada de atenção que são aplicadas nos dois blocos de código. Também considerando as parametrizações já usadas nos exemplos anteriores, a tradução deste tipo de estrutura ficaria:

```
if(len > 0){
   //CODE
}else{
   //CODE
}
```

4.3.5 Algoritmo de Pesquisa de Estruturas de Controlo de Fluxo

Esta secção descreve o algoritmo implementado para pesquisar estruturas de controlo de fluxo em ficheiros `qhasm`.

Numa primeira fase, e para cada função a traduzir, as instruções relacionadas com a implementação de estruturas de controlo de fluxo são colocadas num vector, pela ordem que

aparecem no código fonte. O número da linha em que a instrução ocorre é também guardado. São consideradas instruções `goto`, condicionais ou não, condições de teste e *labels*. Estas instruções encontram-se detalhadas nas tabelas 3.11 e 3.12.

A segunda fase consiste pesquisar no vector sequências de instruções que possam ser traduzidas em estruturas mais alto-nível. A procura é realizada recorrendo a 2 ciclos. O segundo ciclo está contido dentro do primeiro. O seguinte código ilustra a forma como a pesquisa é realizada. Assume-se que o vector contêm 21 entradas, em conformidade com os índices apresentados na figura 4.1. Os índices variam então de `0` até `20`.

```
for(j = 20; j >= 2; j--)
{
   encontrada = 0;
   for(i = 0; i <= (j-2); i++)
   {
      encontrada = procura_estrutura(vector, j, i);
      if(encontrada) break;
   }
   if(encontrada) break;
}
```

A estrutura de controlo de fluxo `if` é a que utiliza menor número de instruções para ser representada. São 3 no total. Como tal, não faz sentido procurar estruturas em secções com menos de 3 posições e daí surgem as condições de paragem `j>=2` e `i<=(j-2)`.

A função `procura_estrutura` verifica se existe um padrão de uma estrutura nas bordas daquela secção do vector. A secção está compreendida entre os índices `j` e `i`. No caso de ser detectado um padrão, é verificada a validade da estrutura. Por exemplo, no caso de um ciclo `while`, não deve existir nenhum `goto` que referencie uma *label* exterior ao ciclo.

Caso a estrutura seja válida, a função `procura_estrutura` retorna o valor 1. Contudo, antes de retornar, inicia uma nova pesquisa, recursivamente, em duas secções do vector: na secção superior e interior. A coluna 2 e 3 da figura 4.1 representa o estado da pesquisa: antes de uma estrutura `if` ser detectada e após ser validada, respectivamente.

De notar que no caso em que a estrutura de controlo de fluxo é do tipo `if-else`, a nova fase de pesquisa é iniciada em 3 secções, a superior, a correspondente ao *then* e ao *else*.

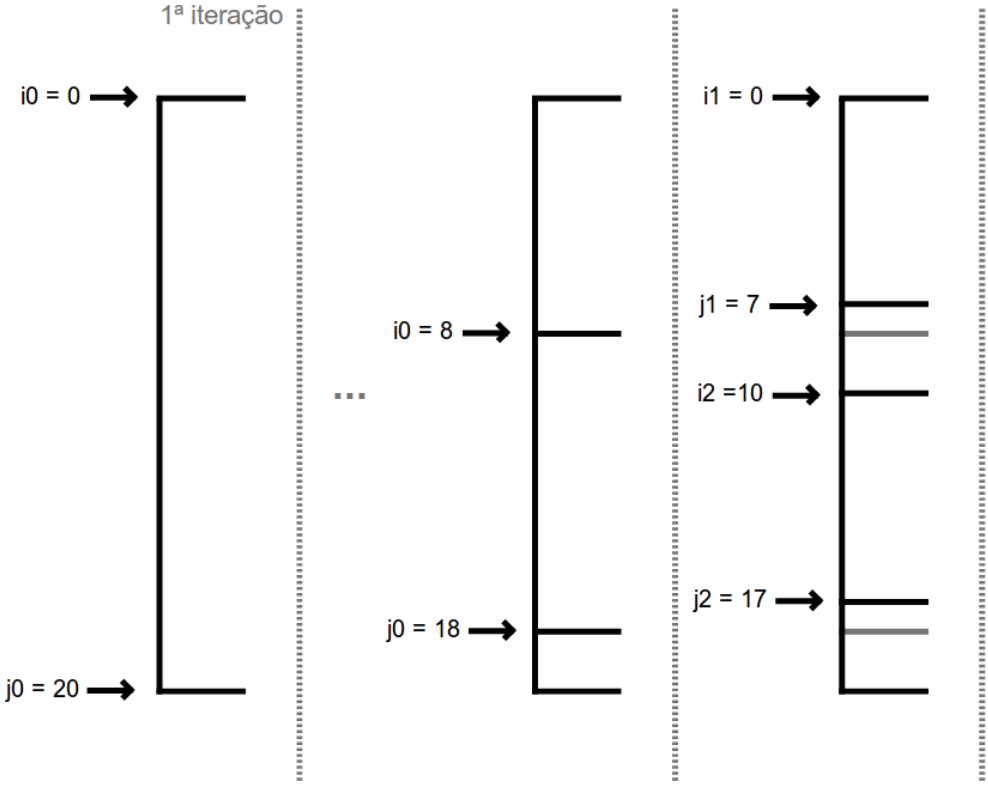

Figura 4.1: Algoritmo de pesquisa de estruturas de controlo de fluxo.

Capítulo 5

Verificação de Código qhasm

Este capítulo tem como objectivo apresentar o processo de validação, no contexto da $safety$, da implementação AES em estudo. As técnicas de optimização utilizadas nesta implementação foram publicadas em [5].

A secção 5.1 apresenta a implementação do AES a validar. A secção 5.2 apresenta as anotações `ACSL` desenvolvidas para auxiliar a validação das obrigações de prova. Por fim, na secção 5.3, são discutidos os resultados obtidos.

5.1 Apresentação da Cifra por Blocos AES

O AES [9], originalmente denominado por Rijndael, consiste numa cifra por blocos que pode operar com chaves de 16, 24 e 32 bytes e, dependendo do tamanho da chave, o número de rondas que realiza é de 10, 12 e 14, respectivamente. Cada bloco tem um tamanho fixo de 16 bytes.

A forma como está implementada a cifra por blocos AES escrita em `qhasm`, escolhida como prova de conceito da metodologia de tradução aqui apresentada, diverge das implementações habituais. Tipicamente, o AES é descrito como uma sequência de operações em matrizes 4x4 denominadas por `SubBytes`, `ShiftRows`, `MixColumns` e `AddRoundKey`. Na última ronda não é realizada a operação `MixColumns`.

A implementação em análise, utiliza uma técnica descrita pelos autores da Rijndael, que permite substituir as operações `SubBytes`, `ShiftRows` e `MixColumns`, bastando para isso realizar 16 acessos a 4 tabelas, cada uma com 256 entradas de 32 bits cada. As tabelas estão definidas no anexo E. A implementação traduzida encontra-se no anexo D.

O modo utilizado nesta implementação é o $counter\text{-}mode$. Este modo permite emular uma cifra sequencial síncrona. O contador utilizado pode ser conjugado de diversas formas com o $nonce$ (vector de inicialização) e o único requisito imposto para o contador é que possua valores distintos para todos os blocos.

A implementação possui 3 funções, nomeadamente:

`ECRYPT_ivsetup`: Inicializa o contador copiando o `iv` passado como argumento;

`ECRYPT_keysetup`: Realiza a expansão da chave;

`ECRYPT_process_bytes`: Cifra ou decifra uma mensagem.

A verificação incidirá especialmente na função `ECRYPT_process_bytes`. As outras serão apenas discutidas na secção 5.3.

5.2 Tradução e Anotações Desenvolvidas

Esta secção tem como objectivo apresentar quais as anotações e lemas desenvolvidos com o intuito de proceder à validação estática no contexto da $safety$ do código AES, traduzido para `C` seguindo o método exposto.

5.2.1 Lemas

Para validar certos tipos de operações é necessário desenvolver lemas, na forma de anotações `ACSL`. Um lema disponibiliza informação adicional em forma de predicados ao `Frama-c`. Esta informação adicional é necessária pois o `Frama-c` não possui dados suficientes sobre a semântica de certas operações. Durante a verificação, estes predicados são assumidos

como verdadeiros, mas devem ser validados individualmente. Contudo, os lemas aqui apresentados são de tal forma simples que a sua validação segue directamente da inspecção e semântica do C.

Devido ao elevado custo computacional associado à validação de código complexo, os lemas pretendem-se o mais simples possível, ao invés de serem demasiado genéricos.

Após realizado um levantamento de todas as operações, presentes no código AES traduzido e nas macros, que poderiam originar problemas na verificação da ausência de $overflows$, foram desenvolvidos 3 conjuntos de lemas. Serão apresentados ao longo desta secção.

Disjunção Exclusiva

O primeiro conjunto de lemas incide sobre o comportamento de operações de disjunção exclusiva, xor em assembly. Os lemas são direccionados para inteiros sem sinal, unsigned int, e são apresentados de seguida. De notar que é utilizada a constante UINT_MAX definida em limits.h.

```
1 /*@ lemma r1: \forall unsigned int x, y;
2   @   (x <= 255) && (y <= 255) ==> 0<=(x^y)<=255;
3   @
4   @ lemma r2: \forall unsigned int x, y;
5   @   (x <= UINT_MAX) && (y <= UINT_MAX) ==> 0<=(x^y)<=UINT_MAX;
6   @*/
```

O primeiro lema, r1, define que caso 2 inteiros sem sinal se encontrem na gama de valores [0..255] então a disjunção exclusiva entre estes 2 valores encontra-se também nessa gama. O segundo lema é similar ao primeiro com a excepção que o limite superior é maior. Para concluir sobre a validade destes lemas é necessário perceber bem qual o funcionamento da operação xor.

A instrução xor, ou disjunção exclusiva, é definida no standard do C como: cada bit do resultado deve ser colocado a 1, se e só se apenas um dos bits correspondentes dos operandos

possuírem valor `1`. Ou seja, a operação é realizada bit a bit e, caso os bits tenham valores diferentes, o bit correspondente do resultado é `1`, caso contrário é colocado a `0`.

Daqui se conclui que, no caso dos inteiros sem sinal, o valor resultante de uma operação de disjunção exclusiva nunca será inferior a `0` (é `0` quando os dois operandos são idênticos). Quanto ao limite superior, este consiste no primeiro inteiro com todos os bits a `1` que surge após o maior valor dos operandos.

Conjunção Lógica

O segundo conjunto de lemas incide sobre o comportamento de operações de conjunção lógica, ou `and`. Também para este caso os lemas são direccionados para inteiros sem sinal, `unsigned int`, e são apresentados na seguinte caixa de código.

```
/*@ lemma r3: \forall unsigned int x;
  @   (x <= UINT_MAX) ==> 0 <= (x&0x3) <= 0x3;
  @
  @ lemma r4: \forall unsigned int x;
  @   (x <= UINT_MAX) ==> 0 <= (x&0x0f) <= 0x0f;
  @
  @ lemma r5: \forall unsigned int x;
  @   (x <= UINT_MAX) ==> 0 <= (x&0xff) <= 0xff;
  @
  @ lemma r6: \forall unsigned int x;
  @   (x <= UINT_MAX) ==> 0 <= (x&0xff00) <= 0xff00;
  @
  @ lemma r7: \forall unsigned int x;
  @   (x <= UINT_MAX) ==> 0 <= (x&0xff0000) <= 0xff0000;
  @
  @ lemma r8: \forall unsigned int x;
  @   (x <= UINT_MAX) ==> 0 <= (x&0xff000000) <= 0xff000000;
  @*/
```

O primeiro lema, `r3`, define que o resultado de um "e lógico" entre um inteiro sem sinal e a constante `0x3` terá como resultado um valor no intervalo `[0..0x3]`.

A forma como é definida este tipo de instrução no `C` é a seguinte: cada bit do resultado deve ser colocado a `1`, se e só se os bits correspondentes dos operandos possuírem valor `1`. Ou seja, caso ambos os bits possuam valor `1` o resultado é `1`, caso contrário é `0`.

Por exemplo, o resultado da instrução `x & 0xff00` com `0 <= x <= UINT_MAX` seria `0` no caso de `x` não partilhar nenhum dos bits com a máscara ou estaria definido num subconjunto do intervalo `[0x0100..0xff00]`, dado que o primeiro byte do resultado seria sempre igual a `0x00`.

Dado que os lemas aqui definidos tem como objectivo oferecer um mecanismo que permita a validação da $safety$ das operações não é necessário ser rigoroso na especificação dos intervalos para os lemas `r6`, `r7` e `r8`. O intervalo definido para os 3 primeiros, `r3`, `r4` e `r5`, consiste no intervalo completo.

Shift Lógico

O terceiro e último conjunto de lemas incide sobre o comportamento de operações de deslocação de bits, em `assembly` `shr` ou `shl` dependendo se o deslocamento é realizado para a direita ou esquerda, respectivamente. A seguinte caixa de código apresenta os lemas desenvolvidos.

```
1  /*@ lemma r9: \forall unsigned int x;
2    @   (x <= UINT_MAX) ==> 0 <= (x >> 2) <= UINT_MAX;
3    @
4    @ lemma r10: \forall unsigned int x;
5    @   (x <= UINT_MAX) ==> 0 <= (x >> 8) <= UINT_MAX;
6    @
7    @ lemma r11: \forall unsigned int x;
8    @   (x <= UINT_MAX) ==> 0 <= (x >> 16) <= UINT_MAX;
9    @
10   @ lemma r12: \forall unsigned int x;
11   @   (x <= UINT_MAX) ==> 0 <= (x >> 24) <= UINT_MAX;
```

```
12   @*/
```

Existem dois tipos de operações de deslocamento, lógicas e aritméticas. A principal diferença reside no facto da desconsideração do bit de sinal. No caso de deslocamentos realizados para a direita, a deslocação aritmética preserva o sinal do inteiro, caso este exista, enquanto que a deslocação lógica não. Como apenas estamos interessados em inteiros sem sinal esta particularidade pode ser desconsiderada.

A forma como está definida a deslocação lógica para inteiros sem sinal no `C` é a seguinte: o resultado de `E1 >> E2` é dado por uma deslocação para a direita de `E2` bits sendo que do lado esquerdo são inseridos zeros. Consiste na seguinte divisão inteira: $\texttt{E1}/2^{\texttt{E2}}$. O resultado de `E1 << E2` é similar ao anterior só que a deslocação é para a esquerda e o resultado é dado pela expressão $\texttt{E1} \times 2^{\texttt{E2}}\ mod\ 2^{\texttt{n}}$, sendo `n` o número de bits da representação utilizada, neste caso 32 bits.

O primeiro lema, `r9`, define que o resultado de um deslocamento para a direita de 2 bits (insere 2 zeros do lado esquerdo) está entre `0` e `UINT_MAX`. Os restantes lemas apresentados são similares. De notar que inicialmente os lemas foram definidos com mais restrição no intervalo, `0 <= (x » 2) <= (UINT_MAX » 2)`. Contudo, tal introduzia muito peso computacional ao provar as obrigações pelo que foi necessário assumir um intervalo mais vasto.

5.2.2 Pré-Condições

Os pré-requisitos de uma função consistem nas propriedades requeridas para que a função tenha o comportamento esperado. Os seguintes pré-requisitos foram especificados para a função `ECRYPT_process_bytes` da implementação em estudo.

```
1 /*@ requires arg5 >= 0 &&
2             \valid(arg2 + (0..17)) &&
3             \valid(arg3 + (0..arg5-1)) &&
4             \valid(arg4 + (0..arg5-1)) &&
5             \valid(aes_tablex + (0..4095)) &&
6             \valid(aes_table0 + (0..4092)) &&
```

```
7               \valid(aes_table1 + (0..4093)) &&
8               \valid(aes_table2 + (0..4094)) &&
9               \valid(aes_table3 + (0..1022));
10    @*/
11  void ECRYPT_process_bytes(void *arg1, unsigned int *arg2,
12                            unsigned char *arg3, unsigned char *arg4,
13                            unsigned int arg5)
14  {
```

Visto que a função `ECRYPT_process_bytes` em `C`, presente no anexo D, foi traduzida de forma automática a partir da implementação `qhasm`, esta preservou, como é suposto, os nomes pouco explícitos dos argumentos. Apresentam-se nos seguintes parágrafos os argumentos da função.

`arg1` Este argumento não é utilizado no decorrer do código, contudo deve ser declarado para que a implementação em `C` seja compatível com a implementação `qhasm`, ou seja, é possível substituir a implementação `qhasm` pela respectiva implementação traduzida em `C`. A declaração dos argumentos em `qhasm` é global e, portanto, comum a todas as funções (ver secção 3.5). O tipo usado é `void*` de forma a fazer sobressair essa não utilização. Não existe, claro, nenhum pré-requisito que envolva esta variável.

`arg2` Apontador para a estrutura que contém a chave e o contador. Esta estrutura é internamente é composta por 2 vectores de inteiros sem sinal, cada um com 32 bits. O primeiro vector, destinado à chave, possui 14 posições de memória e o segundo, pertencente ao contador, 4 posições. O intervalo válido para este apontador é de `0` até `17` portanto.

`arg3` Contém o conteúdo do *input* que pode ser texto limpo ou cifrado. O seu comprimento é de `arg5` posições pelo que a região de memória válida está definida no intervalo de `0` até `arg5-1`.

`arg4` Consiste na memória para *output*. Pode ser destinado a texto limpo ou cifrado dependendo do *input*. Deve possuir `arg5` posições de memória válidas, ou seja um intervalo válido de `0` até `arg5-1`.

`arg5` Consiste no comprimento da mensagem. O pré-requisito `arg5 >= 0` acaba por ser redundante uma vez que a variável está declarada como `unsigned int`.

Quanto às pré-condições definidas para as tabelas pré-calculadas `aes_tablex`, `aes_table0`, `aes_table1`, `aes_table2` e `aes_table3` (presentes no anexo E), estas são necessárias pois referenciam a tabela `aes_full`.

5.2.3 Variantes e Invariantes de Ciclo

Um variante de ciclo consiste numa expressão cujo valor deve obrigatoriamente diminuir ao longo da execução de um ciclo. Permite concluir sobre a terminação deste. Já um invariante consiste numa expressão ou conjunto de expressões que devem permanecer válidas antes, durante e após o ciclo. É útil, por exemplo, no contexto da validação de acessos a memória que são indexados por uma variável mutável dentro do ciclo.

A função `ECRYPT_process_bytes` contém um ciclo. Mais concretamente um ciclo do tipo `do-while`. É portanto necessário especificar um variante e um invariante para este ciclo. O seguinte excerto de código consiste num resumo do ciclo e contém já o variante e os invariantes.

```
1  in = arg3;
2  out = arg4;
3  len = arg5;
4
5  if (len > 0) {
6     //...
7
8     /*@ loop invariant 0 < len <= arg5;
9       @ loop invariant (in + len) == (arg3 + arg5);
10      @ loop invariant (out + len) == (arg4 + arg5);
```

```
11      @ loop variant len;
12      @*/
13    do{
14
15       len -= 16;
16       in += 16;
17       out += 16;
18
19    }while(len > 0);
20
21    //...
22 }
```

O variante definido é a própria variável `len`, que é decrementada em 16 unidades a cada iteração.

Existem 3 invariantes. O primeiro especifica que `len` deve ser superior a `0` e igual ou inferior ao antigo valor de `len`. De notar que o valor de `len` não decrementa para além de `0` pois, dentro do ciclo e antes do decremento é invocada a instrução `break` no caso do valor de `len` ser inferior a 16. Uma particularidade na definição deste invariante: no caso da variável `len` possuir um valor múltiplo de 16, o ciclo termina com `len` igual a `0`. Contudo, pela forma como este ciclo é traduzido para a linguagem intermédia do `Jessie`, a preservação deste invariante é validada. A validação que `len` é maior que `0` é necessária para obter a garantia que um dos acessos a memória realizados pelas variáveis `in` e `out` é válido.

O segundo e o terceiro invariante surgem em consequência de um caso especial que existe nesta implementação, o incremento de apontadores. Como este incremento ocorre dentro do ciclo a ferramenta de verificação não consegue validar os acessos a memória realizados através das variáveis `in` e `out`.

A forma encontrada para lidar com o problema foi a especificação de 2 invariantes, um para cada variável. Basicamente relacionam os valores antigos (anteriores ao ciclo) de `len`, `in` e `out` com os respectivos valores actuais.

5.3 Verificação e Resultados

Esta secção discutirá o processo e os resultados da verificação realizada.

Devido ao peso computacional associado a este tipo de provas, especialmente quando o código a verificar é complexo, foi necessário substituir as chamadas às macros presentes no anexo C por funções com código equivalente. Existiu deste modo uma redução do número de obrigações de prova geradas, absolutamente necessária pois, com a complexidade anterior, a ferramenta não tinha memória suficiente para gerar as obrigações de prova[1].

Para a função `ECRYPT_process_bytes` foram geradas 2123 obrigações de prova, e excluídas as provas de lemas que são 12, foram validadas com sucesso 2097 obrigações de prova. Não foi possível validar 14 obrigações de prova relativas a *overflows*. Essas obrigações de prova são relativas à linha 846 do código presente no anexo D, `y0 += 1;`. A variável `y0` corresponde à parte menos significativa do valor do contador e é incrementada em 1 unidade a cada bloco processado.

Conclui-se que no caso da diferença da variável `y0` no estado inicial em relação ao inteiro máximo representável ser menor que o número de blocos a processar irá ocorrer um *overflow*. Contudo, este *overflow* não cria problemas neste contexto dado que a única restrição a que o contador está sujeito é ser diferente em todos os blocos, pelo que quando o *overflow* ocorre a variável é colocada a `0`. Num modelo ideal, em que o custo computacional poderia ser sempre negligenciado, faria todo o sentido realizar a soma ou a colocação a `0` da variável somente após a realização de uma condição de teste.

Para as funções `ECRYPT_keysetup` e `ECRYPT_ivsetup`, foram geradas 688 obrigações de prova. Foi no entanto, para além dos lemas anteriormente definidos adicionar o seguinte lema, muito similar com o lema `r2`:

```
1 /*@ lemma s2: \forall integer x, y;
2   @   (0 <= x <= UINT_MAX) && (0 <= y <= UINT_MAX) ==> 0<=(x^y)<=
      UINT_MAX;
3   @*/
```

[1] A quantidade máxima de memória alocada a um processo na máquina de testes estava limitada a aproximadamente 3GB.

A diferença consiste na declaração `integer` ao invés de `unsigned int`. Isto porque existem operações de disjunção exclusiva entre variáveis e constantes. A operação `ROTATE`, definida no anexo C, e utilizada na função `ECRYPT_keysetup`, gera um *overflow*. Contudo, esse *overflow* é suposto acontecer. Em consequência do "ou lógico", realizado pela operação `ROTATE`, foi necessário acrescentar o seguinte lema:

```
1 /*@ lemma s3: \forall unsigned int x, y;
2   @    (x <= UINT_MAX) && (y <= UINT_MAX) ==> 0<=(x|y)<=UINT_MAX;
3   @*/
```

Posto isto foram validadas 675 obrigações de prova, sendo que as 13 que não foram validadas correspondem aos lemas e ao *overflow* gerado pela instrução `ROTATE`.

Capítulo 6

Conclusão

Esta dissertação foi sobretudo orientada ao estudo e desenvolvimento de uma solução prática que possibilitasse a validação de software criptográfico de elevado desempenho desenvolvido em `qhasm`. Para tal objectivo ser alcançado o primeiro passo consistiu em realizar estudo sobre esta linguagem pouco divulgada e com fins muito específicos. Esse estudo foi elaborado tendo como base a experimentação prática uma vez que a documentação disponível apenas apresenta as instruções suportadas pelo `qhasm` de forma muito sucinta.

Depois de analisar a linguagem de programação `qhasm` constatou-se que, a menos de algumas instruções muito específicas de cada arquitectura de computador, o comportamento do `qhasm` e `C` era muito similar, pelo que uma possível abordagem seria adaptar e traduzir código `qhasm` para código `C` de forma a tirar partido das melhores ferramentas de verificação actualmente disponíveis. Tal abordagem tornou-se viável a partir do momento em que foi possível contornar os problemas criados por instruções do tipo `goto` e apontadores do tipo `void*`.

Após as ideias gerais estarem maturadas o principal objectivo foi testar o trabalho desenvolvido com uma implementação real e de valor reconhecido. Contudo, o peso computacional da interface gráfica da ferramenta de verificação foi um dos maiores entraves a ultrapassar, pois a complexidade do algoritmo que se pretendia validar era considerável. Tornou-se possível ultrapassar esta limitação, após algum estudo da ferramenta, abdicando da interface gráfica disponibilizada e substituindo-a por um pequeno programa que fazia a gestão da validação das obrigações de prova.

Posto isto, conclui-se que as duas principais dificuldades encontradas no decorrer desta dissertação consistiram na falta de documentação da linguagem `qhasm`, e na incapacidade que a interface gráfica da ferramenta de verificação utilizada tem em lidar com casos de dimensão razoável.

Como principais contribuições do trabalho realizado, em primeiro lugar destaca-se a criação de um mecanismo que permite que um sub-conjunto de instruções da linguagem de programação `qhasm` possa ser validado tanto no contexto da $safety$ como no funcional. Outra mais valia deste modelo de tradução é que permite que código `C` traduzido de `qhasm` seja submetido a um depurador, o que pode auxiliar a adopção desta linguagem por parte da comunidade. A validação da $safety$ de uma implementação real foi outro resultado alcançado.

Como possível trabalho futuro de investigação seria interessante, por exemplo, formalizar a definição de equivalência semântica abordada na secção 4.1. Estudar a aplicabilidade prática desta abordagem em plataformas 64 bits ou investigar sobre a adaptabilidade do método desenvolvido para o grupo de instruções sobre números em vírgula flutuante. Criar uma instância da linguagem de programação `qhasm` para a plataforma `ARM` e posteriormente suportar essa plataforma na ferramenta aqui desenvolvida seria outra possibilidade.

Bibliografia

[1] Dakshi Agrawal, Bruce Archambeault, Josyula Rao, and Pankaj Rohatgi. The EM side-channel(s). In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 29–45. Springer Berlin / Heidelberg, 2003. 7

[2] Manuel Barbosa. CACE Deliverable D5.2: Formal Specification Language Definitions and Security Policy Extensions, 2009. 6

[3] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specfication Language*. CEA LIST and INRIA, 2010. ACSL Version 1.4 Implementation in Boron-20100401. 9

[4] Daniel Bernstein. qhasm: tools to help write high-speed software, 2007. 2, 22

[5] Daniel Bernstein and Peter Schwabe. New aes software speed records. In Dipanwita Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *Progress in Cryptology - INDOCRYPT 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 322–336. Springer Berlin / Heidelberg, 2008. 3, 8, 21, 37, 51

[6] Daniel J. Bernstein. Cache-timing attacks on aes. Technical report, 2005. 8

[7] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo: a theorem prover for polymorphic first-order logic modulo theories. 2006. 9

[8] Loïc Correnson, Pascal Cuoq, Armand Puccetti, and Julien Signoles. *Frama-C User Manual*. CEA LIST, 2009. Release Boron-20100401. 9

[9] Joan Daemen, Joan Daemen, Joan Daemen, Vincent Rijmen, and Vincent Rijmen. Aes proposal: Rijndael, 1998. 3, 7, 21, 51

[10] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin / Heidelberg, 2008. 9

[11] Bruno Dutertre and Leonardo De Moura. The yices smt solver. Technical report, 2006. 9

[12] Jean-Christophe Filliâtre. Verification of non-functional programs using interpretations in type theory. *J. Funct. Program.*, 13:709–745, July 2003. 9

[13] Randall Hyde. *The Art of Assembly Language*. No Starch Press, San Francisco, CA, USA, 2003. 13, 25

[14] Intel. *Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 1997. 12

[15] Intel. *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, 1999. 12, 16

[16] Paul Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer Berlin / Heidelberg, 1996. 7

[17] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology - CRYPTO' 99*, volume 1666 of *Lecture Notes in Computer Science*, pages 789–789. Springer Berlin / Heidelberg, 1999. 7

[18] Francois Koeune, Francois Koeune, Jean-Jacques Quisquater, and Jean jacques Quisquater. A timing attack against rijndael. Technical report, 1999. 7

[19] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant aes-gcm. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin / Heidelberg, 2009. 8

[20] Jeff Lewis. Cryptol: specification, implementation and verification of high-grade cryptographic applications. In *Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, FMSE '07, New York, NY, USA, 2007. ACM. 6

[21] Patrick Longa and Catherine Gebotys. Efficient techniques for high-speed elliptic curve cryptography. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 80–94. Springer Berlin / Heidelberg, 2010. 8

[22] Claude Marché and Yannick Moy. *Jessie Plugin Tutorial*. INRIA, 2010. Frama-C version: Boron Jessie plugin version: 2.26. 9

[23] Alfred J. Menezes, Paul C. Van Oorschot, Scott A. Vanstone, and R. L. Rivest. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, USA, 1st edition, 1996. 1

[24] Dag Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin / Heidelberg, 2006. 8

[25] Peter Schwabe. qhasm documentation, 2007. 25

[26] François-Xavier Standaert. Introduction to side-channel attacks. In Ingrid M.R. Verbauwhede, editor, *Secure Integrated Circuits and Systems*, Integrated Circuits and Systems, pages 27–42. Springer US, 2010. 1, 7

Anexo A

Lista de Tipos das Entradas de Mapeamento

```
rui: register unsigned int
rus: register unsigned short
ruc: register unsigned char

ui: unsigned int
us: unsigned short
uc: unsigned char

ruip: register unsigned int*
rusp: register unsigned short*
rucp: register unsigned char*

uip: unsigned int*
usp: unsigned short*
ucp: unsigned char*
```

Anexo B

Entradas de Mapeamento Genéricas

```
inst: $r = $n
in: int32 r, ii n
tr: $r = $n;---$r:rui

inst: $r = $s
in: int32 r, stack32 s
tr: $r = $s;---$r:rui;$s:ui
tr: $r = $s;---$r:ruip;$s:uip
tr: $r = $s;---$r:rusp;$s:usp
tr: $r = $s;---$r:rucp;$s:ucp

inst: $r = *(uint8 *)($s + $n)
in: int32 r, int32 s, ii n
tr: $r = $s[$n];---$r:rui;$s:rucp;

inst: $r = *(uint8 *)($s + $t)
in: int32 r, int32 s, int32 t
tr: $r = $s[$t];---$r:rui;$s:rucp;$t:rui

inst: $r = *(uint32 *)($s + $n)
in: int32 r, int32 s, ii n
tr: $r = $s[$p0];---$r:rui;$s:ruip;---let $p0=$n/sizeof(int)
tr: $r = READ_CHAR_AS_INT($s,$n);---$r:rui;$s:rucp
tr: $r = READ_SHORT_AS_INT($s,$n);---$r:rui;$s:rusp

inst: $r = *(uint32 *)($s + $t)
in: int32 r, int32 s, int32 t
tr: $r = $s[$t>>2];---$r:rui;$s:ruip;$t:rui
tr: $r = READ_CHAR_AS_INT($s,$t);---$r:rui;$s:rucp;$t:rui
tr: $r = READ_SHORT_AS_INT($s,$t);---$r:rui;$s:rusp;$t:rui

inst: $r = *(uint8 *)(&$n + $s * 8)
```

```
in: int32 r, iv n, int32 s
tr: $r = $n[$s * 8];---$r:rui;$n:ucp;$s:rui

inst: $r = *(uint16 *)(&$n + $s * 8)
in: int32 r, iv n, int32 s
tr: $r = $n[$s * $p0];---$r:rui;$n:usp;$s:rui---let $p0=8/sizeof(short)
tr: $r = READ_CHAR_AS_SHORT($n,$s * 8);---$r:rui;$n:ucp;$s:rui

inst: $r = *(uint32 *)(&$n + $s * 8)
in: int32 r, iv n, int32 s
tr: $r = $n[$s * $p0];---$r:rui;$n:uip;$s:rui---let $p0=8/sizeof(int)
tr: $r = READ_SHORT_AS_INT($n,$s * 8);---$r:rui;$n:usp;$s:rui
tr: $r = READ_CHAR_AS_INT($n,$s * 8);---$r:rui;$n:ucp;$s:rui

inst: $r ^= $s
in: int32 r, stack32 s
tr: $r ^= $s;---$r:rui;$s:ui

inst: $r ^= *(uint32 *)($s + $n)
in: int32 r, int32 s, ii n
tr: $r ^= $s[$p0];---$r:rui;$s:ruip;---let $p0=$n/sizeof(int)
tr: $r ^= READ_SHORT_AS_INT($s,$n);---$r:rui;$s:rusp;
tr: $r ^= READ_CHAR_AS_INT($s,$n);---$r:rui;$s:rucp;

inst: $r ^= *(uint32 *)(&$n + $s * 8)
in: int32 r, iv n, int32 s
tr: $r ^= $n[$s * $p0];---$r:rui;$n:uip;$s:rui---let $p0=8/sizeof(int)
tr: $r ^= READ_SHORT_AS_INT($n,$s * 8);---$r:rui;$n:usp;$s:rui
tr: $r ^= READ_CHAR_AS_INT($n,$s * 8);---$r:rui;$n:ucp;$s:rui

inst: $r = $s
in: stack32 r, int32 s
tr: $r = $s;---$r:ui;$s:rui
tr: $r = $s;---$r:uip;$s:ruip
tr: $r = $s;---$r:usp;$s:rusp
tr: $r = $s;---$r:ucp;$s:rucp

inst: *(uint8 *)($s + $n) = $r
in: int32 s, ii n, int32 r
tr: $s[$n] = $r;---$s:rucp;$r:rui

inst: *(uint8 *)($s + $t) = $r
in: int32 s, int32 t, int32 r
tr: $s[$t] = $r & 0xff;---$s:rucp;$t:rui;$r:rui

inst: *(uint32 *)($s + $n) = $r
in: int32 s, ii n, int32 r
tr: $s[$p0] = $r;---$s:ruip;$r:rui---let $p0=$n/sizeof(int)
tr: SAVE_CHAR_AS_INT($s,$n,$r);---$s:rucp;$r:rui

```

```
inst: $r = $s
in: int32 r, int32 s
tr: $r = $s;---$r:rui;$s:rui
tr: $r = $s;---$r:ruip;$s:ruip
tr: $r = $s;---$r:rusp;$s:rusp
tr: $r = $s;---$r:rucp;$s:rucp

inst: $r = $s
in: int32 r, int3232 s
tr: $r = $s;---$r:rui;$s:rui
tr: $r = $s;---$r:ruip;$s:ruip
tr: $r = $s;---$r:rusp;$s:rusp
tr: $r = $s;---$r:rucp;$s:rucp

inst: $r = $s
in: int3232 r, int32 s
tr: $r = $s;---$r:rui;$s:rui
tr: $r = $s;---$r:ruip;$s:ruip
tr: $r = $s;---$r:rusp;$s:rusp
tr: $r = $s;---$r:rucp;$s:rucp

inst: $r = $s & 255
in: int32 r, int32 s
tr: $r = $s & 255;---$r:rui;$s:rui

inst: $r = ($s >> 8) & 255
in: int32 r, int32 s
tr: $r = ($s >> 8) & 255;---$r:rui;$s:rui

inst: $r &= $n
in: int32 r, ii n
tr: $r &= $n;---$r:rui;

inst: $r += $s
in: int32 r, int32 s
tr: $r += $s;---$r:rui;$s:rui
tr: $r += $s;---$r:rucp;$s:rui
tr: $r += ($s>>1);---$r:rusp;$s:rui
tr: $r += ($s>>2);---$r:ruip;$s:rui

inst: $r += $n
in: int32 r, ii n
tr: $r += $n;---$r:rui
tr: $r += $p0;---$r:rucp;---let $p0=$n/sizeof(char)
tr: $r += $p0;---$r:rusp;---let $p0=$n/sizeof(short)
tr: $r += $p0;---$r:ruip;---let $p0=$n/sizeof(int)

inst: $r -= $n
in: int32 r, ii n
tr: $r -= $n;---$r:rui
```

```
133
134 inst: $r <<<= $n
135 in: int32 r, ii n
136 tr: $r = ROTATE($r,$n);---$r:rui
137
138 inst: (uint32) $r >>= $n
139 in: int32 r, ii n
140 tr: $r >>= $n;---$r:rui;
141
142 inst: $r ^= $s
143 in: int32 r, int32 s
144 tr: $r ^= $s;---$r:rui;$s:rui
145
146 inst: $r ^= $n
147 in: int32 r, ii n
148 tr: $r ^= $n;---$r:rui;
149
150 inst: $c? $r-$n
151 in: tflg c, int32 r, ii n
152 tr: //$c? $r-$n;---$r:rui
153 ps: test;$c;$r;$n
154
155 inst: $c? $r-$s
156 in: tflg c, int32 r, int32 s
157 tr: //$c? $r-$s;---$r:rui;$s:rui
158 ps: test;$c;$r;$s
159
160 inst: goto $f if $t
161 in: it f, flg t
162 tr: if($w $cnd $z){goto $f;}
163 ps: gotoif;$f;$t
164
165 inst: goto $f
166 in: it f
167 tr: goto $f;
168 ps: goto;$f
169
170 inst: $f:
171 in: it f
172 tr: $f:
173 ps: label;$f
```

Anexo C

Macros

```
#define ROTATE(VALUE,PLACES) ((VALUE<<PLACES)|(VALUE>>(32-PLACES)))

//No limite lê 3 bytes além do devido, apenas retorna conteúdo da posição válida.
#define READ_INT_AS_CHAR(PTR, OFFSET) ((PTR[OFFSET>>2]>>((OFFSET&0x3)<<3))&0xff)

//No limite lê 3 bytes além do devido, apenas retorna conteúdo da posição válida.
#define READ_INT_AS_SHORT(PTR, OFFSET) ((((PTR[OFFSET>>2]>>24)&0xff)+((PTR[(OFFSET>>2)
    +1]&0xff)<<8))*((OFFSET&0x2)>>1)*(OFFSET&0x1))+(((PTR[OFFSET>>2]>>((OFFSET&0x3)<<3))&0
    xffff)*((((OFFSET&0x2)>>1)^(OFFSET&0x1))^(!(OFFSET&0x3))))

//No limite lê 1 byte além do devido, apenas retorna conteúdo da posição válida.
#define READ_SHORT_AS_CHAR(PTR, OFFSET) ((PTR[OFFSET>>1]>>((OFFSET&0x1)<<3))& 0xff)

//No limite lê 1 byte além do devido, apenas retorna conteúdo da posição válida.
#define READ_SHORT_AS_INT(PTR, OFFSET) (((PTR[(OFFSET>>1)+1]<< 16)+PTR[OFFSET>>1])*(!(
    OFFSET&0x1))) +(((PTR[(OFFSET>>1)+2]&0xff)<<24)+(PTR[(OFFSET>>1)+1]<<8)+(PTR[(OFFSET
    >>1)]>>8))*(OFFSET&0x1)

#define READ_CHAR_AS_INT(PTR, OFFSET) PTR[OFFSET]+(PTR[OFFSET+1]<<8)+(PTR[OFFSET+2]<<16)+(
    PTR[OFFSET+3]<<24))

#define READ_CHAR_AS_SHORT(PTR, OFFSET) (PTR[OFFSET]+(PTR[OFFSET+1]<<8))

//No limite lê 3 bytes além do devido, apenas retorna conteúdo da posição válida.
#define SAVE_INT_AS_CHAR(PTR,OFFSET,VALUE) ((PTR[OFFSET>>2]&ROTATE(0xffffff00,((OFFSET&0x3
    )<<3)))|((VALUE&0xff)<<((OFFSET&0x3)<<3)))

#define SAVE_CHAR_AS_INT(PTR,OFFSET,VALUE) PTR[OFFSET]=VALUE&0xff;PTR[OFFSET+1]=(VALUE&0
    xff00)>>8;PTR[OFFSET+2]=(VALUE&0xff0000)>>16;PTR[OFFSET+3]=(VALUE&0xff000000)>>24;
```

Anexo D

Código Traduzido AES

```
/*@ lemma r1: \forall unsigned int x, y;
  @   (x <= 255) && (y <= 255) ==> 0<=(x^y)<=255;
  @
  @ lemma r2: \forall unsigned int x, y;
  @   (x <= UINT_MAX) && (y <= UINT_MAX) ==> 0<=(x^y)<=UINT_MAX;
  @*/

/*@ lemma r3: \forall unsigned int x;
  @   (x <= UINT_MAX) ==> 0 <= (x&0x3) <= 0x3;
  @
  @ lemma r4: \forall unsigned int x;
  @   (x <= UINT_MAX) ==> 0 <= (x&0x0f) <= 0x0f;
  @
  @ lemma r5: \forall unsigned int x;
  @   (x <= UINT_MAX) ==> 0 <= (x&0xff) <= 0xff;
  @
  @ lemma r6: \forall unsigned int x;
  @   (x <= UINT_MAX) ==> 0 <= (x&0xff00) <= 0xff00;
  @
  @ lemma r7: \forall unsigned int x;
  @   (x <= UINT_MAX) ==> 0 <= (x&0xff0000) <= 0xff0000;
  @
  @ lemma r8: \forall unsigned int x;
  @   (x <= UINT_MAX) ==> 0 <= (x&0xff000000) <= 0xff000000;
  @*/

/*@ lemma r9: \forall unsigned int x;
  @   (x <= UINT_MAX) ==> 0 <= (x >> 2) <= UINT_MAX;
  @
  @ lemma r10: \forall unsigned int x;
  @   (x <= UINT_MAX) ==> 0 <= (x >> 8) <= UINT_MAX;
  @
```

```
33    @ lemma r11: \forall unsigned int x;
34    @   (x <= UINT_MAX) ==> 0 <= (x >> 16) <= UINT_MAX;
35    @
36    @ lemma r12: \forall unsigned int x;
37    @   (x <= UINT_MAX) ==> 0 <= (x >> 24) <= UINT_MAX;
38    @*/
39
40  /*@ requires arg5 >= 0 &&
41              \valid(arg2 + (0..17)) &&
42              \valid(arg3 + (0..arg5-1)) &&
43              \valid(arg4 + (0..arg5-1)) &&
44              \valid(aes_tablex + (0..4095)) &&
45              \valid(aes_table0 + (0..4092)) &&
46              \valid(aes_table1 + (0..4093)) &&
47              \valid(aes_table2 + (0..4094)) &&
48              \valid(aes_table3 + (0..1022));
49    @*/
50  void ECRYPT_process_bytes(void *arg1, unsigned int *arg2, unsigned char *arg3, unsigned
        char *arg4, unsigned int arg5)
51  {
52
53      register unsigned int b0;
54      register unsigned int b1;
55      register unsigned int b2;
56      register unsigned int b3;
57      register unsigned int *c;
58      register unsigned int *c_stack;
59      register unsigned char *in;
60      register unsigned char *in_stack;
61      register unsigned int len;
62      register unsigned int len_stack;
63      unsigned int    n0;
64      unsigned int    n1;
65      unsigned int    n2;
66      unsigned int    n3;
67      register unsigned char *out;
68      register unsigned char *out_stack;
69      register unsigned int p00;
70      register unsigned int p01;
71      register unsigned int p02;
72      register unsigned int p03;
73      register unsigned int p10;
74      register unsigned int p11;
75      register unsigned int p12;
76      register unsigned int p13;
77      register unsigned int p20;
78      register unsigned int p21;
79      register unsigned int p22;
80      register unsigned int p23;
81      register unsigned int p30;
```

```
82      register unsigned int p31;
83      register unsigned int p32;
84      register unsigned int p33;
85      unsigned int    r0;
86      unsigned int    r1;
87      unsigned int    r10;
88      unsigned int    r11;
89      unsigned int    r12;
90      unsigned int    r13;
91      unsigned int    r14;
92      unsigned int    r15;
93      unsigned int    r16;
94      unsigned int    r17;
95      unsigned int    r18;
96      unsigned int    r19;
97      unsigned int    r2;
98      unsigned int    r20;
99      unsigned int    r21;
100     unsigned int    r22;
101     unsigned int    r23;
102     unsigned int    r24;
103     unsigned int    r25;
104     unsigned int    r26;
105     unsigned int    r27;
106     unsigned int    r28;
107     unsigned int    r29;
108     unsigned int    r3;
109     unsigned int    r30;
110     unsigned int    r31;
111     unsigned int    r32;
112     unsigned int    r33;
113     unsigned int    r34;
114     unsigned int    r35;
115     unsigned int    r36;
116     unsigned int    r37;
117     unsigned int    r38;
118     unsigned int    r39;
119     unsigned int    r4;
120     unsigned int    r40;
121     unsigned int    r41;
122     unsigned int    r42;
123     unsigned int    r43;
124     unsigned int    r5;
125     unsigned int    r6;
126     unsigned int    r7;
127     unsigned int    r8;
128     unsigned int    r9;
129     register unsigned int x0;
130     register unsigned int x1;
131     register unsigned int x2;
```

```
    register unsigned int x3;
    register unsigned int y0;
    register unsigned int y1;
    register unsigned int y2;
    register unsigned int y3;
    register unsigned int y3_stack;
    register unsigned int z0;
    register unsigned int z1;
    register unsigned int z1_stack;
    register unsigned int z2;
    register unsigned int z2_stack;
    register unsigned int z3;
    register unsigned int z3_stack;

    c = arg2;
    in = arg3;
    out = arg4;
    len = arg5;

    c_stack = c;
    in_stack = in;
    out_stack = out;
    len_stack = len;

    if (len > 0) {
        x0 = c[0];
        x1 = c[1];
        x2 = c[2];
        x3 = c[3];
        r0 = x0;
        r1 = x1;
        r2 = x2;
        r3 = x3;
        x0 = c[4];
        x1 ^= x0;
        x2 ^= x1;
        x3 ^= x2;
        r4 = x0;
        r5 = x1;
        r6 = x2;
        r7 = x3;
        x0 = c[5];
        x1 ^= x0;
        x2 ^= x1;
        x3 ^= x2;
        r8 = x0;
        r9 = x1;
        r10 = x2;
        r11 = x3;
        x0 = c[6];
```

```
        x1 ^= x0;
        x2 ^= x1;
        x3 ^= x2;
        r12 = x0;
        r13 = x1;
        r14 = x2;
        r15 = x3;
        x0 = c[7];
        x1 ^= x0;
        x2 ^= x1;
        x3 ^= x2;
        r16 = x0;
        r17 = x1;
        r18 = x2;
        r19 = x3;
        x0 = c[8];
        x1 ^= x0;
        x2 ^= x1;
        x3 ^= x2;
        r20 = x0;
        r21 = x1;
        r22 = x2;
        r23 = x3;
        x0 = c[9];
        x1 ^= x0;
        x2 ^= x1;
        x3 ^= x2;
        r24 = x0;
        r25 = x1;
        r26 = x2;
        r27 = x3;
        x0 = c[10];
        x1 ^= x0;
        x2 ^= x1;
        x3 ^= x2;
        r28 = x0;
        r29 = x1;
        r30 = x2;
        r31 = x3;
        x0 = c[11];
        x1 ^= x0;
        x2 ^= x1;
        x3 ^= x2;
        r32 = x0;
        r33 = x1;
        r34 = x2;
        r35 = x3;
        x0 = c[12];
        x1 ^= x0;
        x2 ^= x1;
```

```
232         x3 ^= x2;
233         r36 = x0;
234         r37 = x1;
235         r38 = x2;
236         r39 = x3;
237         x0 = c[13];
238         x1 ^= x0;
239         x2 ^= x1;
240         x3 ^= x2;
241         r40 = x0;
242         r41 = x1;
243         r42 = x2;
244         r43 = x3;
245         y0 = c[14];
246         y1 = c[15];
247         y2 = c[16];
248         y3 = c[17];
249
250       /*@ loop invariant 0 < len <= arg5;
251        @ loop invariant (in + len) == (arg3 + arg5);
252        @ loop invariant (out + len) == (arg4 + arg5);
253        @ loop variant len;
254        @*/
255         do {
256             n0 = y0;
257             n1 = y1;
258             n2 = y2;
259             n3 = y3;
260             y0 ^= r0;
261             y1 ^= r1;
262             y2 ^= r2;
263             y3 ^= r3;
264             y3_stack = y3;
265             p00 = y0 & 255;
266             z0 = READ_CHAR_AS_INT(aes_table0, p00 * 8);
267             p03 = (y0 >> 8) & 255;
268             y0 >>= 16;
269             z3 = READ_CHAR_AS_INT(aes_table1, p03 * 8);
270             p02 = y0 & 255;
271             z2 = READ_CHAR_AS_INT(aes_table2, p02 * 8);
272             p01 = (y0 >> 8) & 255;
273             z1 = aes_table3[p01 * 2];
274             p10 = y1 & 255;
275             z1 ^= READ_CHAR_AS_INT(aes_table0, p10 * 8);
276             p11 = (y1 >> 8) & 255;
277             z0 ^= READ_CHAR_AS_INT(aes_table1, p11 * 8);
278             y1 >>= 16;
279             p12 = y1 & 255;
280             z3 ^= READ_CHAR_AS_INT(aes_table2, p12 * 8);
281             p13 = (y1 >> 8) & 255;
```

```
            z2 ^= aes_table3[p13 * 2];
            y3 = y3_stack;
            p20 = y2 & 255;
            z2 ^= READ_CHAR_AS_INT(aes_table0, p20 * 8);
            p21 = (y2 >> 8) & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table1, p21 * 8);
            y2 >>= 16;
            p22 = y2 & 255;
            z0 ^= READ_CHAR_AS_INT(aes_table2, p22 * 8);
            p23 = (y2 >> 8) & 255;
            z3 ^= aes_table3[p23 * 2];
            p30 = y3 & 255;
            z3 ^= READ_CHAR_AS_INT(aes_table0, p30 * 8);
            p31 = (y3 >> 8) & 255;
            z2 ^= READ_CHAR_AS_INT(aes_table1, p31 * 8);
            y3 >>= 16;
            p32 = y3 & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table2, p32 * 8);
            p33 = (y3 >> 8) & 255;
            z0 ^= aes_table3[p33 * 2];
            y0 = r4;
            y0 ^= z0;
            y1 = r5;
            y1 ^= z1;
            y2 = r6;
            y2 ^= z2;
            y3 = r7;
            y3 ^= z3;
            y3_stack = y3;
            p00 = y0 & 255;
            z0 = READ_CHAR_AS_INT(aes_table0, p00 * 8);
            p03 = (y0 >> 8) & 255;
            y0 >>= 16;
            z3 = READ_CHAR_AS_INT(aes_table1, p03 * 8);
            p02 = y0 & 255;
            z2 = READ_CHAR_AS_INT(aes_table2, p02 * 8);
            p01 = (y0 >> 8) & 255;
            z1 = aes_table3[p01 * 2];
            p10 = y1 & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table0, p10 * 8);
            p11 = (y1 >> 8) & 255;
            z0 ^= READ_CHAR_AS_INT(aes_table1, p11 * 8);
            y1 >>= 16;
            p12 = y1 & 255;
            z3 ^= READ_CHAR_AS_INT(aes_table2, p12 * 8);
            p13 = (y1 >> 8) & 255;
            z2 ^= aes_table3[p13 * 2];
            y3 = y3_stack;
            p20 = y2 & 255;
            z2 ^= READ_CHAR_AS_INT(aes_table0, p20 * 8);
```

```
            p21 = (y2 >> 8) & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table1, p21 * 8);
            y2 >>= 16;
            p22 = y2 & 255;
            z0 ^= READ_CHAR_AS_INT(aes_table2, p22 * 8);
            p23 = (y2 >> 8) & 255;
            z3 ^= aes_table3[p23 * 2];
            p30 = y3 & 255;
            z3 ^= READ_CHAR_AS_INT(aes_table0, p30 * 8);
            p31 = (y3 >> 8) & 255;
            z2 ^= READ_CHAR_AS_INT(aes_table1, p31 * 8);
            y3 >>= 16;
            p32 = y3 & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table2, p32 * 8);
            p33 = (y3 >> 8) & 255;
            z0 ^= aes_table3[p33 * 2];
            y0 = r8;
            y0 ^= z0;
            y1 = r9;
            y1 ^= z1;
            y2 = r10;
            y2 ^= z2;
            y3 = r11;
            y3 ^= z3;
            y3_stack = y3;
            p00 = y0 & 255;
            z0 = READ_CHAR_AS_INT(aes_table0, p00 * 8);
            p03 = (y0 >> 8) & 255;
            y0 >>= 16;
            z3 = READ_CHAR_AS_INT(aes_table1, p03 * 8);
            p02 = y0 & 255;
            z2 = READ_CHAR_AS_INT(aes_table2, p02 * 8);
            p01 = (y0 >> 8) & 255;
            z1 = aes_table3[p01 * 2];
            p10 = y1 & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table0, p10 * 8);
            p11 = (y1 >> 8) & 255;
            z0 ^= READ_CHAR_AS_INT(aes_table1, p11 * 8);
            y1 >>= 16;
            p12 = y1 & 255;
            z3 ^= READ_CHAR_AS_INT(aes_table2, p12 * 8);
            p13 = (y1 >> 8) & 255;
            z2 ^= aes_table3[p13 * 2];
            y3 = y3_stack;
            p20 = y2 & 255;
            z2 ^= READ_CHAR_AS_INT(aes_table0, p20 * 8);
            p21 = (y2 >> 8) & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table1, p21 * 8);
            y2 >>= 16;
            p22 = y2 & 255;
```

```
382             z0 ^= READ_CHAR_AS_INT(aes_table2, p22 * 8);
383             p23 = (y2 >> 8) & 255;
384             z3 ^= aes_table3[p23 * 2];
385             p30 = y3 & 255;
386             z3 ^= READ_CHAR_AS_INT(aes_table0, p30 * 8);
387             p31 = (y3 >> 8) & 255;
388             z2 ^= READ_CHAR_AS_INT(aes_table1, p31 * 8);
389             y3 >>= 16;
390             p32 = y3 & 255;
391             z1 ^= READ_CHAR_AS_INT(aes_table2, p32 * 8);
392             p33 = (y3 >> 8) & 255;
393             z0 ^= aes_table3[p33 * 2];
394             y0 = r12;
395             y0 ^= z0;
396             y1 = r13;
397             y1 ^= z1;
398             y2 = r14;
399             y2 ^= z2;
400             y3 = r15;
401             y3 ^= z3;
402             y3_stack = y3;
403             p00 = y0 & 255;
404             z0 = READ_CHAR_AS_INT(aes_table0, p00 * 8);
405             p03 = (y0 >> 8) & 255;
406             y0 >>= 16;
407             z3 = READ_CHAR_AS_INT(aes_table1, p03 * 8);
408             p02 = y0 & 255;
409             z2 = READ_CHAR_AS_INT(aes_table2, p02 * 8);
410             p01 = (y0 >> 8) & 255;
411             z1 = aes_table3[p01 * 2];
412             p10 = y1 & 255;
413             z1 ^= READ_CHAR_AS_INT(aes_table0, p10 * 8);
414             p11 = (y1 >> 8) & 255;
415             z0 ^= READ_CHAR_AS_INT(aes_table1, p11 * 8);
416             y1 >>= 16;
417             p12 = y1 & 255;
418             z3 ^= READ_CHAR_AS_INT(aes_table2, p12 * 8);
419             p13 = (y1 >> 8) & 255;
420             z2 ^= aes_table3[p13 * 2];
421             y3 = y3_stack;
422             p20 = y2 & 255;
423             z2 ^= READ_CHAR_AS_INT(aes_table0, p20 * 8);
424             p21 = (y2 >> 8) & 255;
425             z1 ^= READ_CHAR_AS_INT(aes_table1, p21 * 8);
426             y2 >>= 16;
427             p22 = y2 & 255;
428             z0 ^= READ_CHAR_AS_INT(aes_table2, p22 * 8);
429             p23 = (y2 >> 8) & 255;
430             z3 ^= aes_table3[p23 * 2];
431             p30 = y3 & 255;
```

```
            z3 ^= READ_CHAR_AS_INT(aes_table0, p30 * 8);
            p31 = (y3 >> 8) & 255;
            z2 ^= READ_CHAR_AS_INT(aes_table1, p31 * 8);
            y3 >>= 16;
            p32 = y3 & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table2, p32 * 8);
            p33 = (y3 >> 8) & 255;
            z0 ^= aes_table3[p33 * 2];
            y0 = r16;
            y0 ^= z0;
            y1 = r17;
            y1 ^= z1;
            y2 = r18;
            y2 ^= z2;
            y3 = r19;
            y3 ^= z3;
            y3_stack = y3;
            p00 = y0 & 255;
            z0 = READ_CHAR_AS_INT(aes_table0, p00 * 8);
            p03 = (y0 >> 8) & 255;
            y0 >>= 16;
            z3 = READ_CHAR_AS_INT(aes_table1, p03 * 8);
            p02 = y0 & 255;
            z2 = READ_CHAR_AS_INT(aes_table2, p02 * 8);
            p01 = (y0 >> 8) & 255;
            z1 = aes_table3[p01 * 2];
            p10 = y1 & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table0, p10 * 8);
            p11 = (y1 >> 8) & 255;
            z0 ^= READ_CHAR_AS_INT(aes_table1, p11 * 8);
            y1 >>= 16;
            p12 = y1 & 255;
            z3 ^= READ_CHAR_AS_INT(aes_table2, p12 * 8);
            p13 = (y1 >> 8) & 255;
            z2 ^= aes_table3[p13 * 2];
            y3 = y3_stack;
            p20 = y2 & 255;
            z2 ^= READ_CHAR_AS_INT(aes_table0, p20 * 8);
            p21 = (y2 >> 8) & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table1, p21 * 8);
            y2 >>= 16;
            p22 = y2 & 255;
            z0 ^= READ_CHAR_AS_INT(aes_table2, p22 * 8);
            p23 = (y2 >> 8) & 255;
            z3 ^= aes_table3[p23 * 2];
            p30 = y3 & 255;
            z3 ^= READ_CHAR_AS_INT(aes_table0, p30 * 8);
            p31 = (y3 >> 8) & 255;
            z2 ^= READ_CHAR_AS_INT(aes_table1, p31 * 8);
            y3 >>= 16;
```

```
            p32 = y3 & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table2, p32 * 8);
            p33 = (y3 >> 8) & 255;
            z0 ^= aes_table3[p33 * 2];
            y0 = r20;
            y0 ^= z0;
            y1 = r21;
            y1 ^= z1;
            y2 = r22;
            y2 ^= z2;
            y3 = r23;
            y3 ^= z3;
            y3_stack = y3;
            p00 = y0 & 255;
            z0 = READ_CHAR_AS_INT(aes_table0, p00 * 8);
            p03 = (y0 >> 8) & 255;
            y0 >>= 16;
            z3 = READ_CHAR_AS_INT(aes_table1, p03 * 8);
            p02 = y0 & 255;
            z2 = READ_CHAR_AS_INT(aes_table2, p02 * 8);
            p01 = (y0 >> 8) & 255;
            z1 = aes_table3[p01 * 2];
            p10 = y1 & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table0, p10 * 8);
            p11 = (y1 >> 8) & 255;
            z0 ^= READ_CHAR_AS_INT(aes_table1, p11 * 8);
            y1 >>= 16;
            p12 = y1 & 255;
            z3 ^= READ_CHAR_AS_INT(aes_table2, p12 * 8);
            p13 = (y1 >> 8) & 255;
            z2 ^= aes_table3[p13 * 2];
            y3 = y3_stack;
            p20 = y2 & 255;
            z2 ^= READ_CHAR_AS_INT(aes_table0, p20 * 8);
            p21 = (y2 >> 8) & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table1, p21 * 8);
            y2 >>= 16;
            p22 = y2 & 255;
            z0 ^= READ_CHAR_AS_INT(aes_table2, p22 * 8);
            p23 = (y2 >> 8) & 255;
            z3 ^= aes_table3[p23 * 2];
            p30 = y3 & 255;
            z3 ^= READ_CHAR_AS_INT(aes_table0, p30 * 8);
            p31 = (y3 >> 8) & 255;
            z2 ^= READ_CHAR_AS_INT(aes_table1, p31 * 8);
            y3 >>= 16;
            p32 = y3 & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table2, p32 * 8);
            p33 = (y3 >> 8) & 255;
            z0 ^= aes_table3[p33 * 2];
```

```
532             y0 = r24;
533             y0 ^= z0;
534             y1 = r25;
535             y1 ^= z1;
536             y2 = r26;
537             y2 ^= z2;
538             y3 = r27;
539             y3 ^= z3;
540             y3_stack = y3;
541             p00 = y0 & 255;
542             z0 = READ_CHAR_AS_INT(aes_table0, p00 * 8);
543             p03 = (y0 >> 8) & 255;
544             y0 >>= 16;
545             z3 = READ_CHAR_AS_INT(aes_table1, p03 * 8);
546             p02 = y0 & 255;
547             z2 = READ_CHAR_AS_INT(aes_table2, p02 * 8);
548             p01 = (y0 >> 8) & 255;
549             z1 = aes_table3[p01 * 2];
550             p10 = y1 & 255;
551             z1 ^= READ_CHAR_AS_INT(aes_table0, p10 * 8);
552             p11 = (y1 >> 8) & 255;
553             z0 ^= READ_CHAR_AS_INT(aes_table1, p11 * 8);
554             y1 >>= 16;
555             p12 = y1 & 255;
556             z3 ^= READ_CHAR_AS_INT(aes_table2, p12 * 8);
557             p13 = (y1 >> 8) & 255;
558             z2 ^= aes_table3[p13 * 2];
559             y3 = y3_stack;
560             p20 = y2 & 255;
561             z2 ^= READ_CHAR_AS_INT(aes_table0, p20 * 8);
562             p21 = (y2 >> 8) & 255;
563             z1 ^= READ_CHAR_AS_INT(aes_table1, p21 * 8);
564             y2 >>= 16;
565             p22 = y2 & 255;
566             z0 ^= READ_CHAR_AS_INT(aes_table2, p22 * 8);
567             p23 = (y2 >> 8) & 255;
568             z3 ^= aes_table3[p23 * 2];
569             p30 = y3 & 255;
570             z3 ^= READ_CHAR_AS_INT(aes_table0, p30 * 8);
571             p31 = (y3 >> 8) & 255;
572             z2 ^= READ_CHAR_AS_INT(aes_table1, p31 * 8);
573             y3 >>= 16;
574             p32 = y3 & 255;
575             z1 ^= READ_CHAR_AS_INT(aes_table2, p32 * 8);
576             p33 = (y3 >> 8) & 255;
577             z0 ^= aes_table3[p33 * 2];
578             y0 = r28;
579             y0 ^= z0;
580             y1 = r29;
581             y1 ^= z1;
```

```
            y2 = r30;
            y2 ^= z2;
            y3 = r31;
            y3 ^= z3;
            y3_stack = y3;
            p00 = y0 & 255;
            z0 = READ_CHAR_AS_INT(aes_table0, p00 * 8);
            p03 = (y0 >> 8) & 255;
            y0 >>= 16;
            z3 = READ_CHAR_AS_INT(aes_table1, p03 * 8);
            p02 = y0 & 255;
            z2 = READ_CHAR_AS_INT(aes_table2, p02 * 8);
            p01 = (y0 >> 8) & 255;
            z1 = aes_table3[p01 * 2];
            p10 = y1 & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table0, p10 * 8);
            p11 = (y1 >> 8) & 255;
            z0 ^= READ_CHAR_AS_INT(aes_table1, p11 * 8);
            y1 >>= 16;
            p12 = y1 & 255;
            z3 ^= READ_CHAR_AS_INT(aes_table2, p12 * 8);
            p13 = (y1 >> 8) & 255;
            z2 ^= aes_table3[p13 * 2];
            y3 = y3_stack;
            p20 = y2 & 255;
            z2 ^= READ_CHAR_AS_INT(aes_table0, p20 * 8);
            p21 = (y2 >> 8) & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table1, p21 * 8);
            y2 >>= 16;
            p22 = y2 & 255;
            z0 ^= READ_CHAR_AS_INT(aes_table2, p22 * 8);
            p23 = (y2 >> 8) & 255;
            z3 ^= aes_table3[p23 * 2];
            p30 = y3 & 255;
            z3 ^= READ_CHAR_AS_INT(aes_table0, p30 * 8);
            p31 = (y3 >> 8) & 255;
            z2 ^= READ_CHAR_AS_INT(aes_table1, p31 * 8);
            y3 >>= 16;
            p32 = y3 & 255;
            z1 ^= READ_CHAR_AS_INT(aes_table2, p32 * 8);
            p33 = (y3 >> 8) & 255;
            z0 ^= aes_table3[p33 * 2];
            y0 = r32;
            y0 ^= z0;
            y1 = r33;
            y1 ^= z1;
            y2 = r34;
            y2 ^= z2;
            y3 = r35;
            y3 ^= z3;
```

```
632             y3_stack = y3;
633             p00 = y0 & 255;
634             z0 = READ_CHAR_AS_INT(aes_table0, p00 * 8);
635             p03 = (y0 >> 8) & 255;
636             y0 >>= 16;
637             z3 = READ_CHAR_AS_INT(aes_table1, p03 * 8);
638             p02 = y0 & 255;
639             z2 = READ_CHAR_AS_INT(aes_table2, p02 * 8);
640             p01 = (y0 >> 8) & 255;
641             z1 = aes_table3[p01 * 2];
642             p10 = y1 & 255;
643             z1 ^= READ_CHAR_AS_INT(aes_table0, p10 * 8);
644             p11 = (y1 >> 8) & 255;
645             z0 ^= READ_CHAR_AS_INT(aes_table1, p11 * 8);
646             y1 >>= 16;
647             p12 = y1 & 255;
648             z3 ^= READ_CHAR_AS_INT(aes_table2, p12 * 8);
649             p13 = (y1 >> 8) & 255;
650             z2 ^= aes_table3[p13 * 2];
651             y3 = y3_stack;
652             p20 = y2 & 255;
653             z2 ^= READ_CHAR_AS_INT(aes_table0, p20 * 8);
654             p21 = (y2 >> 8) & 255;
655             z1 ^= READ_CHAR_AS_INT(aes_table1, p21 * 8);
656             y2 >>= 16;
657             p22 = y2 & 255;
658             z0 ^= READ_CHAR_AS_INT(aes_table2, p22 * 8);
659             p23 = (y2 >> 8) & 255;
660             z3 ^= aes_table3[p23 * 2];
661             p30 = y3 & 255;
662             z3 ^= READ_CHAR_AS_INT(aes_table0, p30 * 8);
663             p31 = (y3 >> 8) & 255;
664             z2 ^= READ_CHAR_AS_INT(aes_table1, p31 * 8);
665             y3 >>= 16;
666             p32 = y3 & 255;
667             z1 ^= READ_CHAR_AS_INT(aes_table2, p32 * 8);
668             p33 = (y3 >> 8) & 255;
669             z0 ^= aes_table3[p33 * 2];
670             y0 = r36;
671             y0 ^= z0;
672             y1 = r37;
673             y1 ^= z1;
674             y2 = r38;
675             y2 ^= z2;
676             y3 = r39;
677             y3 ^= z3;
678             y3_stack = y3;
679             z0 = y0 & 255;
680             z0 = aes_table2[z0 * 8];
681             z3 = (y0 >> 8) & 255;
```

```
682             z3 = READ_CHAR_AS_SHORT(aes_tablex, z3 * 8);
683             y0 >>= 16;
684             z2 = y0 & 255;
685             z2 = READ_CHAR_AS_INT(aes_table0, z2 * 8);
686             z2 &= 0x00ff0000;
687             z1 = (y0 >> 8) & 255;
688             z1 = READ_CHAR_AS_INT(aes_table1, z1 * 8);
689             z1 &= 0xff000000;
690             z0 ^= r40;
691             z3 ^= r43;
692             z1 ^= r41;
693             z2 ^= r42;
694             p10 = y1 & 255;
695             p10 = aes_table2[p10 * 8];
696             z1 ^= p10;
697             p11 = (y1 >> 8) & 255;
698             p11 = READ_CHAR_AS_SHORT(aes_tablex, p11 * 8);
699             z0 ^= p11;
700             y1 >>= 16;
701             p12 = y1 & 255;
702             p12 = READ_CHAR_AS_INT(aes_table0, p12 * 8);
703             p12 &= 0x00ff0000;
704             z3 ^= p12;
705             p13 = (y1 >> 8) & 255;
706             p13 = READ_CHAR_AS_INT(aes_table1, p13 * 8);
707             p13 &= 0xff000000;
708             z2 ^= p13;
709             y3 = y3_stack;
710             p20 = y2 & 255;
711             p20 = aes_table2[p20 * 8];
712             z2 ^= p20;
713             p21 = (y2 >> 8) & 255;
714             p21 = READ_CHAR_AS_SHORT(aes_tablex, p21 * 8);
715             z1 ^= p21;
716             y2 >>= 16;
717             p22 = y2 & 255;
718             p22 = READ_CHAR_AS_INT(aes_table0, p22 * 8);
719             p22 &= 0x00ff0000;
720             z0 ^= p22;
721             p23 = (y2 >> 8) & 255;
722             p23 = READ_CHAR_AS_INT(aes_table1, p23 * 8);
723             p23 &= 0xff000000;
724             z3 ^= p23;
725             p30 = y3 & 255;
726             p30 = aes_table2[p30 * 8];
727             z3 ^= p30;
728             p31 = (y3 >> 8) & 255;
729             p31 = READ_CHAR_AS_SHORT(aes_tablex, p31 * 8);
730             z2 ^= p31;
731             y3 >>= 16;
```

```
732             p32 = y3 & 255;
733             p32 = READ_CHAR_AS_INT(aes_table0, p32 * 8);
734             p32 &= 0x00ff0000;
735             z1 ^= p32;
736             p33 = (y3 >> 8) & 255;
737             p33 = READ_CHAR_AS_INT(aes_table1, p33 * 8);
738             p33 &= 0xff000000;
739             z0 ^= p33;
740 
741             len = len_stack;
742             if (len < 16) {
743                 z1_stack = z1;
744                 z2_stack = z2;
745                 z3_stack = z3;
746                 //in = in_stack;
747                 //out = out_stack;
748                 b0 = in[0];
749                 b0 ^= z0;
750                 out[0] = b0 & 0xff;
751                 if (len > 1) {
752                     z0 >>= 8;
753                     b1 = in[1];
754                     b1 ^= z0;
755                     out[1] = b1 & 0xff;
756                     if (len > 2) {
757                         z0 >>= 8;
758                         b2 = in[2];
759                         b2 ^= z0;
760                         out[2] = b2 & 0xff;
761                         if (len > 3) {
762                             z0 >>= 8;
763                             b3 = in[3];
764                             b3 ^= z0;
765                             out[3] = b3 & 0xff;
766                             if (len > 4) {
767                                 z1 = z1_stack;
768                                 b0 = in[4];
769                                 b0 ^= z1;
770                                 out[4] = b0 & 0xff;
771                                 if (len > 5) {
772                                     z1 >>= 8;
773                                     b1 = in[5];
774                                     b1 ^= z1;
775                                     out[5] = b1 & 0xff;
776                                     if (len > 6) {
777                                         z1 >>= 8;
778                                         b2 = in[6];
779                                         b2 ^= z1;
780                                         out[6] = b2 & 0xff;
781                                         if (len > 7) {
```

```
                                                z1 >>= 8;
                                                b3 = in[7];
                                                b3 ^= z1;
                                                out[7] = b3 & 0xff;
                                                if (len > 8) {
                                                    z2 = z2_stack;
                                                    b0 = in[8];
                                                    b0 ^= z2;
                                                    out[8] = b0 & 0xff;
                                                    if (len > 9) {
                                                        z2 >>= 8;
                                                        b1 = in[9];
                                                        b1 ^= z2;
                                                        out[9] = b1 & 0xff;
                                                        if (len > 10) {
                                                            z2 >>= 8;
                                                            b2 = in[10];
                                                            b2 ^= z2;
                                                            out[10] = b2 & 0xff;
                                                            if (len > 11) {
                                                                z2 >>= 8;
                                                                b3 = in[11];
                                                                b3 ^= z2;
                                                                out[11] = b3 & 0xff;
                                                                if (len > 12) {
                                                                  z3 = z3_stack;
                                                                  b0 = in[12];
                                                                  b0 ^= z3;
                                                                  out[12] = b0 & 0xff;
                                                                  if (len > 13) {
                                                                    z3 >>= 8;
                                                                    b1 = in[13];
                                                                    b1 ^= z3;
                                                                    out[13] = b1 & 0xff;
                                                                    if (len > 14) {
                                                                      z3 >>= 8;
                                                                      b2 = in[14];
                                                                      b2 ^= z3;
                                                                      out[14] = b2 & 0xff;
                                                                      if (len > 15) {
                                                                       b3 = in[15];
                                                                       b3 ^= z3;
                                                                       out[15] = b3 & 0xff;
                                                                       out[15] = z3 & 0xff;
                                                                      }
                                                                     }
                                                                   }
                                                                }
                                                            }
                                                        }
                                                    }
```

```
                                                    }
                                                }
                                            }
                                        }
                                    }
                                }
                            }
                        }

                        y0 = n0;
                        y1 = n1;
                        y2 = n2;
                        y3 = n3;
                        y0 += 1;
                        break;
                    }
                    len -= 16;
                    len_stack = len;
                    in = in_stack;
                    z0 ^= READ_CHAR_AS_INT(in, 0);
                    z1 ^= READ_CHAR_AS_INT(in, 4);
                    z2 ^= READ_CHAR_AS_INT(in, 8);
                    z3 ^= READ_CHAR_AS_INT(in, 12);
                    in += 16;
                    in_stack = in;
                    out = out_stack;
                    SAVE_CHAR_AS_INT(out, 0, z0);
                    SAVE_CHAR_AS_INT(out, 4, z1);
                    SAVE_CHAR_AS_INT(out, 8, z2);
                    SAVE_CHAR_AS_INT(out, 12, z3);
                    out += 16;
                    out_stack = out;
                    y0 = n0;
                    y1 = n1;
                    y2 = n2;
                    y3 = n3;
                    y0 += 1;
                    len = len_stack;
                } while (len > 0);

                c = c_stack;
                c[14] = y0;
                c[15] = y1;
                c[16] = y2;
                c[17] = y3;
            }
            return;
}
```

Anexo E

Tradução das Tabelas AES

```
unsigned int aes_full[] = {
0xc6a56300, 0xc6a56363, 0xf8847c00, 0xf8847c7c, 0xee997700, 0xee997777, 0xf68d7b00,
0xf68d7b7b, 0xff0df200, 0xff0df2f2, 0xd6bd6b00, 0xd6bd6b6b, 0xdeb16f00, 0xdeb16f6f,
0x9154c500, 0x9154c5c5, 0x60503000, 0x60503030, 0x02030100, 0x02030101, 0xcea96700,
0xcea96767, 0x567d2b00, 0x567d2b2b, 0xe719fe00, 0xe719fefe, 0xb562d700, 0xb562d7d7,
0x4de6ab00, 0x4de6abab, 0xec9a7600, 0xec9a7676, 0x8f45ca00, 0x8f45caca, 0x1f9d8200,
0x1f9d8282, 0x8940c900, 0x8940c9c9, 0xfa877d00, 0xfa877d7d, 0xef15fa00, 0xef15fafa,
0xb2eb5900, 0xb2eb5959, 0x8ec94700, 0x8ec94747, 0xfb0bf000, 0xfb0bf0f0, 0x41ecad00,
0x41ecadad, 0xb367d400, 0xb367d4d4, 0x5ffda200, 0x5ffda2a2, 0x45eaaf00, 0x45eaafaf,
0x23bf9c00, 0x23bf9c9c, 0x53f7a400, 0x53f7a4a4, 0xe4967200, 0xe4967272, 0x9b5bc000,
0x9b5bc0c0, 0x75c2b700, 0x75c2b7b7, 0xe11cfd00, 0xe11cfdfd, 0x3dae9300, 0x3dae9393,
0x4c6a2600, 0x4c6a2626, 0x6c5a3600, 0x6c5a3636, 0x7e413f00, 0x7e413f3f, 0xf502f700,
0xf502f7f7, 0x834fcc00, 0x834fcccc, 0x685c3400, 0x685c3434, 0x51f4a500, 0x51f4a5a5,
0xd134e500, 0xd134e5e5, 0xf908f100, 0xf908f1f1, 0xe2937100, 0xe2937171, 0xab73d800,
0xab73d8d8, 0x62533100, 0x62533131, 0x2a3f1500, 0x2a3f1515, 0x080c0400, 0x080c0404,
0x9552c700, 0x9552c7c7, 0x46652300, 0x46652323, 0x9d5ec300, 0x9d5ec3c3, 0x30281800,
0x30281818, 0x37a19600, 0x37a19696, 0x0a0f0500, 0x0a0f0505, 0x2fb59a00, 0x2fb59a9a,
0x0e090700, 0x0e090707, 0x24361200, 0x24361212, 0x1b9b8000, 0x1b9b8080, 0xdf3de200,
0xdf3de2e2, 0xcd26eb00, 0xcd26ebeb, 0x4e692700, 0x4e692727, 0x7fcdb200, 0x7fcdb2b2,
0xea9f7500, 0xea9f7575, 0x121b0900, 0x121b0909, 0x1d9e8300, 0x1d9e8383, 0x58742c00,
0x58742c2c, 0x342e1a00, 0x342e1a1a, 0x362d1b00, 0x362d1b1b, 0xdcb26e00, 0xdcb26e6e,
0xb4ee5a00, 0xb4ee5a5a, 0x5bfba000, 0x5bfba0a0, 0xa4f65200, 0xa4f65252, 0x764d3b00,
0x764d3b3b, 0xb761d600, 0xb761d6d6, 0x7dceb300, 0x7dceb3b3, 0x527b2900, 0x527b2929,
0xdd3ee300, 0xdd3ee3e3, 0x5e712f00, 0x5e712f2f, 0x13978400, 0x13978484, 0xa6f55300,
0xa6f55353, 0xb968d100, 0xb968d1d1, 0x00000000, 0x00000000, 0xc12ced00, 0xc12ceded,
0x40602000, 0x40602020, 0xe31ffc00, 0xe31ffcfc, 0x79c8b100, 0x79c8b1b1, 0xb6ed5b00,
0xb6ed5b5b, 0xd4be6a00, 0xd4be6a6a, 0x8d46cb00, 0x8d46cbcb, 0x67d9be00, 0x67d9bebe,
0x724b3900, 0x724b3939, 0x94de4a00, 0x94de4a4a, 0x98d44c00, 0x98d44c4c, 0xb0e85800,
0xb0e85858, 0x854acf00, 0x854acfcf, 0xbb6bd000, 0xbb6bd0d0, 0xc52aef00, 0xc52aefef,
0x4fe5aa00, 0x4fe5aaaa, 0xed16fb00, 0xed16fbfb, 0x86c54300, 0x86c54343, 0x9ad74d00,
0x9ad74d4d, 0x66553300, 0x66553333, 0x11948500, 0x11948585, 0x8acf4500, 0x8acf4545,
0xe910f900, 0xe910f9f9, 0x04060200, 0x04060202, 0xfe817f00, 0xfe817f7f, 0xa0f05000,
```

```
0xa0f05050, 0x78443c00, 0x78443c3c, 0x25ba9f00, 0x25ba9f9f, 0x4be3a800, 0x4be3a8a8,
0xa2f35100, 0xa2f35151, 0x5dfea300, 0x5dfea3a3, 0x80c04000, 0x80c04040, 0x058a8f00,
0x058a8f8f, 0x3fad9200, 0x3fad9292, 0x21bc9d00, 0x21bc9d9d, 0x70483800, 0x70483838,
0xf104f500, 0xf104f5f5, 0x63dfbc00, 0x63dfbcbc, 0x77c1b600, 0x77c1b6b6, 0xaf75da00,
0xaf75dada, 0x42632100, 0x42632121, 0x20301000, 0x20301010, 0xe51aff00, 0xe51affff,
0xfd0ef300, 0xfd0ef3f3, 0xbf6dd200, 0xbf6dd2d2, 0x814ccd00, 0x814ccdcd, 0x18140c00,
0x18140c0c, 0x26351300, 0x26351313, 0xc32fec00, 0xc32fecec, 0xbee15f00, 0xbee15f5f,
0x35a29700, 0x35a29797, 0x88cc4400, 0x88cc4444, 0x2e391700, 0x2e391717, 0x9357c400,
0x9357c4c4, 0x55f2a700, 0x55f2a7a7, 0xfc827e00, 0xfc827e7e, 0x7a473d00, 0x7a473d3d,
0xc8ac6400, 0xc8ac6464, 0xbae75d00, 0xbae75d5d, 0x322b1900, 0x322b1919, 0xe6957300,
0xe6957373, 0xc0a06000, 0xc0a06060, 0x19988100, 0x19988181, 0x9ed14f00, 0x9ed14f4f,
0xa37fdc00, 0xa37fdcdc, 0x44662200, 0x44662222, 0x547e2a00, 0x547e2a2a, 0x3bab9000,
0x3bab9090, 0x0b838800, 0x0b838888, 0x8cca4600, 0x8cca4646, 0xc729ee00, 0xc729eeee,
0x6bd3b800, 0x6bd3b8b8, 0x283c1400, 0x283c1414, 0xa779de00, 0xa779dede, 0xbce25e00,
0xbce25e5e, 0x161d0b00, 0x161d0b0b, 0xad76db00, 0xad76dbdb, 0xdb3be000, 0xdb3be0e0,
0x64563200, 0x64563232, 0x744e3a00, 0x744e3a3a, 0x141e0a00, 0x141e0a0a, 0x92db4900,
0x92db4949, 0x0c0a0600, 0x0c0a0606, 0x486c2400, 0x486c2424, 0xb8e45c00, 0xb8e45c5c,
0x9f5dc200, 0x9f5dc2c2, 0xbd6ed300, 0xbd6ed3d3, 0x43efac00, 0x43efacac, 0xc4a66200,
0xc4a66262, 0x39a89100, 0x39a89191, 0x31a49500, 0x31a49595, 0xd337e400, 0xd337e4e4,
0xf28b7900, 0xf28b7979, 0xd532e700, 0xd532e7e7, 0x8b43c800, 0x8b43c8c8, 0x6e593700,
0x6e593737, 0xdab76d00, 0xdab76d6d, 0x018c8d00, 0x018c8d8d, 0xb164d500, 0xb164d5d5,
0x9cd24e00, 0x9cd24e4e, 0x49e0a900, 0x49e0a9a9, 0xd8b46c00, 0xd8b46c6c, 0xacfa5600,
0xacfa5656, 0xf307f400, 0xf307f4f4, 0xcf25ea00, 0xcf25eaea, 0xcaaf6500, 0xcaaf6565,
0xf48e7a00, 0xf48e7a7a, 0x47e9ae00, 0x47e9aeae, 0x10180800, 0x10180808, 0x6fd5ba00,
0x6fd5baba, 0xf0887800, 0xf0887878, 0x4a6f2500, 0x4a6f2525, 0x5c722e00, 0x5c722e2e,
0x38241c00, 0x38241c1c, 0x57f1a600, 0x57f1a6a6, 0x73c7b400, 0x73c7b4b4, 0x9751c600,
0x9751c6c6, 0xcb23e800, 0xcb23e8e8, 0xa17cdd00, 0xa17cdddd, 0xe89c7400, 0xe89c7474,
0x3e211f00, 0x3e211f1f, 0x96dd4b00, 0x96dd4b4b, 0x61dcbd00, 0x61dcbdbd, 0x0d868b00,
0x0d868b8b, 0x0f858a00, 0x0f858a8a, 0xe0907000, 0xe0907070, 0x7c423e00, 0x7c423e3e,
0x71c4b500, 0x71c4b5b5, 0xccaa6600, 0xccaa6666, 0x90d84800, 0x90d84848, 0x06050300,
0x06050303, 0xf701f600, 0xf701f6f6, 0x1c120e00, 0x1c120e0e, 0xc2a36100, 0xc2a36161,
0x6a5f3500, 0x6a5f3535, 0xaef95700, 0xaef95757, 0x69d0b900, 0x69d0b9b9, 0x17918600,
0x17918686, 0x9958c100, 0x9958c1c1, 0x3a271d00, 0x3a271d1d, 0x27b99e00, 0x27b99e9e,
0xd938e100, 0xd938e1e1, 0xeb13f800, 0xeb13f8f8, 0x2bb39800, 0x2bb39898, 0x22331100,
0x22331111, 0xd2bb6900, 0xd2bb6969, 0xa970d900, 0xa970d9d9, 0x07898e00, 0x07898e8e,
0x33a79400, 0x33a79494, 0x2db69b00, 0x2db69b9b, 0x3c221e00, 0x3c221e1e, 0x15928700,
0x15928787, 0xc920e900, 0xc920e9e9, 0x8749ce00, 0x8749cece, 0xaaff5500, 0xaaff5555,
0x50782800, 0x50782828, 0xa57adf00, 0xa57adfdf, 0x038f8c00, 0x038f8c8c, 0x59f8a100,
0x59f8a1a1, 0x09808900, 0x09808989, 0x1a170d00, 0x1a170d0d, 0x65dabf00, 0x65dabfbf,
0xd731e600, 0xd731e6e6, 0x84c64200, 0x84c64242, 0xd0b86800, 0xd0b86868, 0x82c34100,
0x82c34141, 0x29b09900, 0x29b09999, 0x5a772d00, 0x5a772d2d, 0x1e110f00, 0x1e110f0f,
0x7bcbb000, 0x7bcbb0b0, 0xa8fc5400, 0xa8fc5454, 0x6dd6bb00, 0x6dd6bbbb, 0x2c3a1600,
0x2c3a1616, 0x00630000, 0x63000000, 0x007c0000, 0x7c000000, 0x00770000, 0x77000000,
0x007b0000, 0x7b000000, 0x00f20000, 0xf2000000, 0x006b0000, 0x6b000000, 0x006f0000,
0x6f000000, 0x00c50000, 0xc5000000, 0x00300000, 0x30000000, 0x00010000, 0x01000000,
0x00670000, 0x67000000, 0x002b0000, 0x2b000000, 0x00fe0000, 0xfe000000, 0x00d70000,
0xd7000000, 0x00ab0000, 0xab000000, 0x00760000, 0x76000000, 0x00ca0000, 0xca000000,
0x00820000, 0x82000000, 0x00c90000, 0xc9000000, 0x007d0000, 0x7d000000, 0x00fa0000,
0xfa000000, 0x00590000, 0x59000000, 0x00470000, 0x47000000, 0x00f00000, 0xf0000000,
0x00ad0000, 0xad000000, 0x00d40000, 0xd4000000, 0x00a20000, 0xa2000000, 0x00af0000,
```

```
0xaf000000, 0x009c0000, 0x9c000000, 0x00a40000, 0xa4000000, 0x00720000, 0x72000000,
0x00c00000, 0xc0000000, 0x00b70000, 0xb7000000, 0x00fd0000, 0xfd000000, 0x00930000,
0x93000000, 0x00260000, 0x26000000, 0x00360000, 0x36000000, 0x003f0000, 0x3f000000,
0x00f70000, 0xf7000000, 0x00cc0000, 0xcc000000, 0x00340000, 0x34000000, 0x00a50000,
0xa5000000, 0x00e50000, 0xe5000000, 0x00f10000, 0xf1000000, 0x00710000, 0x71000000,
0x00d80000, 0xd8000000, 0x00310000, 0x31000000, 0x00150000, 0x15000000, 0x00040000,
0x04000000, 0x00c70000, 0xc7000000, 0x00230000, 0x23000000, 0x00c30000, 0xc3000000,
0x00180000, 0x18000000, 0x00960000, 0x96000000, 0x00050000, 0x05000000, 0x009a0000,
0x9a000000, 0x00070000, 0x07000000, 0x00120000, 0x12000000, 0x00800000, 0x80000000,
0x00e20000, 0xe2000000, 0x00eb0000, 0xeb000000, 0x00270000, 0x27000000, 0x00b20000,
0xb2000000, 0x00750000, 0x75000000, 0x00090000, 0x09000000, 0x00830000, 0x83000000,
0x002c0000, 0x2c000000, 0x001a0000, 0x1a000000, 0x001b0000, 0x1b000000, 0x006e0000,
0x6e000000, 0x005a0000, 0x5a000000, 0x00a00000, 0xa0000000, 0x00520000, 0x52000000,
0x003b0000, 0x3b000000, 0x00d60000, 0xd6000000, 0x00b30000, 0xb3000000, 0x00290000,
0x29000000, 0x00e30000, 0xe3000000, 0x002f0000, 0x2f000000, 0x00840000, 0x84000000,
0x00530000, 0x53000000, 0x00d10000, 0xd1000000, 0x00000000, 0x00000000, 0x00ed0000,
0xed000000, 0x00200000, 0x20000000, 0x00fc0000, 0xfc000000, 0x00b10000, 0xb1000000,
0x005b0000, 0x5b000000, 0x006a0000, 0x6a000000, 0x00cb0000, 0xcb000000, 0x00be0000,
0xbe000000, 0x00390000, 0x39000000, 0x004a0000, 0x4a000000, 0x004c0000, 0x4c000000,
0x00580000, 0x58000000, 0x00cf0000, 0xcf000000, 0x00d00000, 0xd0000000, 0x00ef0000,
0xef000000, 0x00aa0000, 0xaa000000, 0x00fb0000, 0xfb000000, 0x00430000, 0x43000000,
0x004d0000, 0x4d000000, 0x00330000, 0x33000000, 0x00850000, 0x85000000, 0x00450000,
0x45000000, 0x00f90000, 0xf9000000, 0x00020000, 0x02000000, 0x007f0000, 0x7f000000,
0x00500000, 0x50000000, 0x003c0000, 0x3c000000, 0x009f0000, 0x9f000000, 0x00a80000,
0xa8000000, 0x00510000, 0x51000000, 0x00a30000, 0xa3000000, 0x00400000, 0x40000000,
0x008f0000, 0x8f000000, 0x00920000, 0x92000000, 0x009d0000, 0x9d000000, 0x00380000,
0x38000000, 0x00f50000, 0xf5000000, 0x00bc0000, 0xbc000000, 0x00b60000, 0xb6000000,
0x00da0000, 0xda000000, 0x00210000, 0x21000000, 0x00100000, 0x10000000, 0x00ff0000,
0xff000000, 0x00f30000, 0xf3000000, 0x00d20000, 0xd2000000, 0x00cd0000, 0xcd000000,
0x000c0000, 0x0c000000, 0x00130000, 0x13000000, 0x00ec0000, 0xec000000, 0x005f0000,
0x5f000000, 0x00970000, 0x97000000, 0x00440000, 0x44000000, 0x00170000, 0x17000000,
0x00c40000, 0xc4000000, 0x00a70000, 0xa7000000, 0x007e0000, 0x7e000000, 0x003d0000,
0x3d000000, 0x00640000, 0x64000000, 0x005d0000, 0x5d000000, 0x00190000, 0x19000000,
0x00730000, 0x73000000, 0x00600000, 0x60000000, 0x00810000, 0x81000000, 0x004f0000,
0x4f000000, 0x00dc0000, 0xdc000000, 0x00220000, 0x22000000, 0x002a0000, 0x2a000000,
0x00900000, 0x90000000, 0x00880000, 0x88000000, 0x00460000, 0x46000000, 0x00ee0000,
0xee000000, 0x00b80000, 0xb8000000, 0x00140000, 0x14000000, 0x00de0000, 0xde000000,
0x005e0000, 0x5e000000, 0x000b0000, 0x0b000000, 0x00db0000, 0xdb000000, 0x00e00000,
0xe0000000, 0x00320000, 0x32000000, 0x003a0000, 0x3a000000, 0x000a0000, 0x0a000000,
0x00490000, 0x49000000, 0x00060000, 0x06000000, 0x00240000, 0x24000000, 0x005c0000,
0x5c000000, 0x00c20000, 0xc2000000, 0x00d30000, 0xd3000000, 0x00ac0000, 0xac000000,
0x00620000, 0x62000000, 0x00910000, 0x91000000, 0x00950000, 0x95000000, 0x00e40000,
0xe4000000, 0x00790000, 0x79000000, 0x00e70000, 0xe7000000, 0x00c80000, 0xc8000000,
0x00370000, 0x37000000, 0x006d0000, 0x6d000000, 0x008d0000, 0x8d000000, 0x00d50000,
0xd5000000, 0x004e0000, 0x4e000000, 0x00a90000, 0xa9000000, 0x006c0000, 0x6c000000,
0x00560000, 0x56000000, 0x00f40000, 0xf4000000, 0x00ea0000, 0xea000000, 0x00650000,
0x65000000, 0x007a0000, 0x7a000000, 0x00ae0000, 0xae000000, 0x00080000, 0x08000000,
0x00ba0000, 0xba000000, 0x00780000, 0x78000000, 0x00250000, 0x25000000, 0x002e0000,
0x2e000000, 0x001c0000, 0x1c000000, 0x00a60000, 0xa6000000, 0x00b40000, 0xb4000000,
0x00c60000, 0xc6000000, 0x00e80000, 0xe8000000, 0x00dd0000, 0xdd000000, 0x00740000,
```

```
0x74000000, 0x001f0000, 0x1f000000, 0x004b0000, 0x4b000000, 0x00bd0000, 0xbd000000,
0x008b0000, 0x8b000000, 0x008a0000, 0x8a000000, 0x00700000, 0x70000000, 0x003e0000,
0x3e000000, 0x00b50000, 0xb5000000, 0x00660000, 0x66000000, 0x00480000, 0x48000000,
0x00030000, 0x03000000, 0x00f60000, 0xf6000000, 0x000e0000, 0x0e000000, 0x00610000,
0x61000000, 0x00350000, 0x35000000, 0x00570000, 0x57000000, 0x00b90000, 0xb9000000,
0x00860000, 0x86000000, 0x00c10000, 0xc1000000, 0x001d0000, 0x1d000000, 0x009e0000,
0x9e000000, 0x00e10000, 0xe1000000, 0x00f80000, 0xf8000000, 0x00980000, 0x98000000,
0x00110000, 0x11000000, 0x00690000, 0x69000000, 0x00d90000, 0xd9000000, 0x008e0000,
0x8e000000, 0x00940000, 0x94000000, 0x009b0000, 0x9b000000, 0x001e0000, 0x1e000000,
0x00870000, 0x87000000, 0x00e90000, 0xe9000000, 0x00ce0000, 0xce000000, 0x00550000,
0x55000000, 0x00280000, 0x28000000, 0x00df0000, 0xdf000000, 0x008c0000, 0x8c000000,
0x00a10000, 0xa1000000, 0x00890000, 0x89000000, 0x000d0000, 0x0d000000, 0x00bf0000,
0xbf000000, 0x00e60000, 0xe6000000, 0x00420000, 0x42000000, 0x00680000, 0x68000000,
0x00410000, 0x41000000, 0x00990000, 0x99000000, 0x002d0000, 0x2d000000, 0x000f0000,
0x0f000000, 0x00b00000, 0xb0000000, 0x00540000, 0x54000000, 0x00bb0000, 0xbb000000,
0x00160000, 0x16000000};

unsigned char* aes_tablex = ((unsigned char*)aes_full);
unsigned char* aes_table2 = ((unsigned char*)aes_full) + 1;
unsigned char* aes_table1 = ((unsigned char*)aes_full) + 2;
unsigned char* aes_table0 = ((unsigned char*)aes_full) + 3;
unsigned int* aes_table3  = aes_full + 1;
unsigned int* lr_table0   = aes_full + 512;
unsigned int* lr_table1   = aes_full + 513;
```