Instituto de Matemática e Estatística da USP

MAC0216 - Técnicas de Programação I - 2s2023

EP1
Entrega até 8:00 de 4/9/2023
(INDIVIDUAL)

Prof. Daniel Macêdo Batista

1 Problema

Funções de hash são muito úteis em diversos campos da computação, como por exemplo na criptografia. De forma bem resumida, funções de hash são funções que mapeiam uma entrada de qualquer tamanho em um valor de tamanho fixo. A saída de uma função de hash costuma ser chamada de *digest* ou *hash*. Além de utilidade em sistemas criptográficos, funções de hash também são usadas para indexar tabelas de tamanho fixo chamadas tabelas hash. Para ilustrar a utilidade de funções de hash, suponha que existe uma matriz com informações da comunidade da USP (essa matriz seria nossa tabela hash) e que é necessário encontrar as informações de pessoas com número USP igual a x nessa matriz. No caso, para encontrar essas informações, é necessário encontrar a linha da matriz com as informações da pessoa de número USP igual a x. Uma forma de fazer isso, supondo que a matriz esteja ordenada pelos números USP, seria usando busca binária. Outra forma seria usando uma função de hash. Se essa função fosse $f(x) = x\%16$, o valor de $f(x)$ seria a linha da matriz com as informações da pessoa. Supondo alguém com NUSP igual a 7410111, a posição seria 15. Para uma pessoa com NUSP igual a 1234567, a posição seria 7 (Note que, independente do tamanho da entrada, nesse exemplo o hash sempre será um valor de 4 bits). A depender do tamanho da matriz, encontrar a posição buscada pode ser muito mais rápido utilizando uma função de hash do que fazendo uma busca binária. Um problema que ocorre com funções de hash é a possibilidade de ocorrer colisões. Por exemplo, uma pessoa com NUSP igual a 2049503 também teria um hash igual a 15, o que é algo indesejável pois na posição 15 estariam as informações da pessoa de NUSP 7410111. Esse exemplo é suficiente para concluir que funções de hash precisam ser muito bem projetadas para reduzir a chance de ocorrer colisões. No cenário de criptografia, funções de hash podem ser usadas para garantir que uma mensagem não foi modificada. Nesse caso, Alice, que queira enviar uma mensagem para Bob, enviaria, além da mensagem, o hash da mesma. Quando Bob recebesse a mensagem ele rodaria a função de hash sobre a mesma e, caso esse hash não coincidisse com o hash enviado pela Alice, significaria que a mensagem foi modificada por alguém[1].

[1] Esse exemplo em criptografia está extremamente simplificado. Explicar os detalhes disso fogem do escopo da disciplina

Este EP vai explorar o uso de diversas operações da linguagem Assembly (transferência de dados, operações aritméticas, operações lógicas, saltos e *syscalls*) na implementação de uma função de hash. Adicionalmente, um programa em Python também terá que ser escrito para ser usado numa análise comparativa de desempenho.

2 Requisitos

2.1 Códigos em Python e em Assembly

Dois programas deverão ser escritos para serem executados no Linux, um em Python e um em Assembly. Os seguintes requisitos deverão ser cumpridos:

- Os programas receberão via entrada padrão uma string de no mínimo 1 e no máximo 100000 caracteres ASCII[2] e deverão imprimir na saída padrão o hash produzido pela função de hash que está explicada na próxima subseção. Seguem alguns exemplos de execução do programa (Obs.: não há quebra de linha nos textos escritos como entrada nos programas. As quebras aparecem abaixo pela limitação do tamanho da página):

```
lidenbrook@asteroid:~/usp/mac0216/ep1-Daniel$ python ep1P.py
Ola mundo! <------------------------ Isso foi digitado
7ea2319be0d038908161b4e8c26bfc7a <-- Essa foi a saída do programa

lidenbrook@asteroid:~/usp/mac0216/ep1-Daniel$ ./ep1S
Ola mundo!
7ea2319be0d038908161b4e8c26bfc7a

lidenbrook@asteroid:~/usp/mac0216/ep1-Daniel$ ./ep1S
Ole mundo!
c0900a10ded215a7297b69d86d81e25a

lidenbrook@asteroid:~/usp/mac0216/ep1-Daniel$ python ep1P.py
que tistreza
878561c1134d1fd53e9b36822e1914cc

lidenbrook@asteroid:~/usp/mac0216/ep1-Daniel$ ./ep1S
Transmita o que aprendeu. Forca, mestria. Mas fraqueza, insensatez, fra
casso tambem. Sim, fracasso acima de tudo. O maior professor, o fracass
o eh. Luke, nos somos o que eles crescem alem. Esse eh o verdadeiro far
do de todos os mestres.
9e064b26292cf88acab540ed6f904b6a

lidenbrook@asteroid:~/usp/mac0216/ep1-Daniel$ python ep1P.py
Bom, ia falando! questao, isso que me sovaca... Ah, formei aquela pergu
nta, para compadre meu Quelemem. Que me respondeu! que, por perto do Ce
```

[2]Dica: para não ter que digitar uma entrada muito grande o tempo todo durante os testes do seu programa, você pode colocar o conteúdo dela em um arquivo e executar o programa desta forma no terminal: `cat arquivo | ./ep1S`

```
u, a gente se alimpou tanto, que todos os feios passados se exalaram de
 nao ser - feito sem-modez de tempo de crianca, mas-artes. Como a gente
 nao carece de ter remorso do que divulgou no latejo de seus pesadelos
de uma noite.
e7ebf435d398ed9c06fe27b9cecceb13

lidenbrook@asteroid:~/usp/mac0216/ep1-Daniel$ ./ep1S
Foi na esquina da rua que ele notou o primeiro indicio de que algo estr
anho ocorria - um gato lia um mapa. Por um instante o Sr. Dursley nao p
ercebeu o que vira - em seguida virou rapidamente a cabeca para dar uma
 segunda olhada. Havia um gato de listras amarelas sentado na esquina d
a rua dos Alfeneiros, mas nao havia nenhum mapa a vista. Em que estaria
 pensando naquela hora? Devia ter sido um efeito da luz. Ele piscou e a
rregalou os olhos para o gato. O gato o encarou. Enquanto virava a esqu
ina e subia a rua, espiou o gato pelo espelho retrovisor. Ele agora est
ava lendo a placa que dizia rua dos Alfeneiros - nao, estava olhando a
placa: gatos nao podiam ler mapas nem placas. O Sr. Dursley sacudiu a c
abeca e tirou o gato do pensamento. Durante o caminho para a cidade ele
nao pensou em mais nada exceto no grande pedido de brocas que tinha esp
erancas de receber naquele dia.
896a06f28a29a4352544dc231913740a
```

- Os programas não podem usar nenhuma biblioteca que forneça funções voltada para cálculos de funções de hash
- Os programas devem ser equivalentes. Isso é importante para que a comparação de desempenho seja justa (Recomenda-se fortemente escrever o programa em Python primeiro e o programa em Assembly depois)
- O programa em Assembly só pode usar instruções e construções da arquitetura x86_64 vistas em sala de aula. Ele será montado e ligado com os seguintes comandos na correção:

```
nasm -f elf64 fonte.s
ld -s fonte.o
```

- O programa em Assembly precisa ter comentários que expliquem o que está sendo feito. Programas em Assembly sem comentários não serão corrigidos (não reproduza no comentário, de forma literal, o que está sendo feito em uma dada linha. Por exemplo, dizer que um comando está movendo o número 4 para o registrador rdx não ajuda. Explique a intenção daquele mov. Recomenda-se que os comentários sejam feitos para cada bloco do programa e não linha a linha)
- Em ambos os programas, mas principalmente no caso do programa em Assembly, use nomes para rótulos e variáveis que auxiliem no entendimento do programa. Se a compreensão do programa ficar muito difícil por conta dos nomes usados, serão descontados pontos
- O programa em Python será corrigido com a versão 3 do interpretador.
- *Warnings* ou erros que apareçam na montagem, ligação e execução dos programas levarão a descontos na nota

Recomenda-se fortemente o uso do depurador `gdb` durante a implementação do programa em Assembly. Ainda sobre o programa em Assembly, vale lembrar que caracteres lidos da entrada padrão e escritos na saída padrão são representados por um código ASCII correspondente. No momento de depuração do programa pode ser útil conferir os códigos ASCII dos caracteres na manpage do ascii, que pode ser lida com o comando `man ascii` no terminal.

2.2 Função de hash a ser implementada

A função de hash a ser implementada é composta de 4 passos descritos a seguir. Considere que x é a string passada como entrada para a função:

2.2.1 Passo 1 - Ajuste do tamanho

Nesse passo é necessário criar uma sequência de bytes, baseada na string original, mas que tenha um tamanho que seja múltiplo de 16 bytes. Como cada caracter da string tem 1 byte, isso significa que a sequência de bytes que será criada precisa ser modificada para ter $16 \times n$ caracteres, com $n \in \mathbb{N}$. Caso o tamanho da string original seja um múltiplo de 16, a sequência de bytes vai ser igual aos códigos ASCII da string original. Caso o tamanho da string original não seja um múltiplo de 16 bytes, ela precisará ser ampliada e teremos que adotar algum critério para definir quais valores serão usados nessa ampliação. Diversas decisões poderiam ser tomadas aqui mas vamos considerar que as novas posições serão preenchidas com $16 - tamanho(x)\%16$. Por exemplo, a string:

```
Ola mundo!
```

possui 10 bytes e, quando mantida em memória, tem o seguinte conteúdo em ASCII (Para facilitar a visualização, cada posição será mostrada ocupando 3 espaços e com um `|` separando as posições):

```
| 79|108| 97| 32|109|117|110|100|111| 33|
```

Após aplicar o passo 1, a sequência de bytes que deverá ser criada é:

```
| 79|108| 97| 32|109|117|110|100|111| 33|  6|  6|  6|  6|  6|  6|
```

Vamos chamar essa sequência acima de $saidaPassoUm$.

2.2.2 Passo 2 - Cálculo e concatenação dos XOR

No passo anterior o tamanho da string foi usado para definir uma primeira mudança nela, que foi a inclusão de novos valores para que a string ficasse com um tamanho múltiplo de 16. No passo 2, o conteúdo da string vai ser usado para influenciar o resultado do hash. Isso será feito realizando diversas operações com os valores de $saidaPassoUm$. Consideraremos que a $saidaPassoUm$ é formada por n blocos de 16 bytes e os valores de cada bloco serão usados para indexar um vetor de 256 inteiros. A ideia é que strings diferentes terão como resultado valores diferentes nesse vetor, aumentando a chance de termos hashes diferentes para strings diferentes, mesmo que a diferença seja apenas em 1 caracter. Como resultado deste passo, serão gerados 16 novos bytes que serão concatenados à $saidaPassoUm$, gerando uma nova sequência com $n + 1$ blocos de 16 bytes que vamos chamar de $saidaPassoDois$. Diversos algoritmos poderiam ser usados aqui. Vamos considerar o seguinte algoritmo para gerar os novos 16 bytes:

$novoBloco \leftarrow [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$
$novoValor \leftarrow 0$
for $i \leftarrow 0$ to $n-1$ **do**
  for $j \leftarrow 0$ to 15 **do**
    $novoValor \leftarrow vetorMagico[(saidaPassoUm[i{\times}16{+}j]$ XOR $novoValor)]$ XOR $novoBloco[j]$
    $novoBloco[j] \leftarrow novoValor$
  end for
end for
return $novoBloco$

Para que o algoritmo funcione, é necessário utilizar o seguinte vetor que possui valores de 0 a 255 distribuídos de forma pseudo-aleatória:

```
vetorMagico = [122, 77, 153, 59, 173, 107, 19, 104, 123, 183, 75, 10,
114, 236, 106, 83, 117, 16, 189, 211, 51, 231, 143, 118, 248, 148, 218,
245, 24, 61, 66, 73, 205, 185, 134, 215, 35, 213, 41, 0, 174, 240, 177,
195, 193, 39, 50, 138, 161, 151, 89, 38, 176, 45, 42, 27, 159, 225, 36,
64, 133, 168, 22, 247, 52, 216, 142, 100, 207, 234, 125, 229, 175, 79,
220, 156, 91, 110, 30, 147, 95, 191, 96, 78, 34, 251, 255, 181, 33, 221,
139, 119, 197, 63, 40, 121, 204, 4, 246, 109, 88, 146, 102, 235, 223,
214, 92, 224, 242, 170, 243, 154, 101, 239, 190, 15, 249, 203, 162, 164,
199, 113, 179, 8, 90, 141, 62, 171, 232, 163, 26, 67, 167, 222, 86, 87,
71, 11, 226, 165, 209, 144, 94, 20, 219, 53, 49, 21, 160, 115, 145, 17,
187, 244, 13, 29, 25, 57, 217, 194, 74, 200, 23, 182, 238, 128, 103,
140, 56, 252, 12, 135, 178, 152, 84, 111, 126, 47, 132, 99, 105, 237,
186, 37, 130, 72, 210, 157, 184, 3, 1, 44, 69, 172, 65, 7, 198, 206,
212, 166, 98, 192, 28, 5, 155, 136, 241, 208, 131, 124, 80, 116, 127,
202, 201, 58, 149, 108, 97, 60, 48, 14, 93, 81, 158, 137, 2, 227, 253,
68, 43, 120, 228, 169, 112, 54, 250, 129, 46, 188, 196, 85, 150, 6, 254,
180, 233, 230, 31, 76, 55, 18, 9, 32, 82, 70]
```

No caso do exemplo que estamos considerando, o novo bloco que terá que ser concatenado é:

```
|147| 70|  0|205| 74|247|244|219|105|175|252| 55|151| 53| 38|205|
```

E a $saidaPassoDois$ é:

```
| 79|108| 97| 32|109|117|110|100|111| 33|  6|  6|  6|  6|  6|  6|
|147| 70|  0|205| 74|247|244|219|105|175|252| 55|151| 53| 38|205|
```

(Para facilitar a visualização, o conteúdo das sequências de bytes acima foi mostrado em linhas de 16 bytes)

2.2.3 Passo 3: Transformação dos n+1 blocos em apenas 3 blocos

Como informado no início do enunciado, funções de hash transformam entradas de tamanhos arbitrários em uma saída de tamanho fixo. Este passo 3 tem o objetivo de transformar os $n+1$ blocos de 16 bytes da $saidaPassoDois$ em uma sequência chamada $saidaPassoTres$ com 3 blocos de 16 bytes. Diversos algoritmos poderiam ser usados para este fim. Vamos considerar o seguinte algoritmo:

$saidaPassoTres \leftarrow [0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \ldots, 0]$ ▷ Inicializa o vetor com 48 zeros
for $i \leftarrow 0$ to n **do**
  for $j \leftarrow 0$ to 15 **do**
    $saidaPassoTres[16 + j] \leftarrow saidaPassoDois[i \times 16 + j]$
    $saidaPassoTres[2 \times 16 + j] \leftarrow (saidaPassoTres[16 + j]$ XOR $saidaPassoTres[j])$
  end for
  $temp \leftarrow 0$
  for $j \leftarrow 0$ to 17 **do**
    for $k \leftarrow 0$ to 47 **do**
      $temp \leftarrow saidaPassoTres[k]$ XOR $vetorMagico[temp]$
      $saidaPassoTres[k] \leftarrow temp$
    end for
    $temp \leftarrow (temp + j)\%256$
  end for
end for
return $saidaPassoTres$

No caso do exemplo que estamos considerando, a saída deste passo será esta sequência de 48 bytes:

```
|126|162| 49|155|224|208| 56|144|129| 97|180|232|194|107|252|122|
|194| 53| 21| 66|190|179|174|146| 84|143|170|129|200| 98| 35| 92|
|209|123| 47|192| 63|184|225| 80| 61| 23|150|202|180| 27| 57|216|
```

2.2.4 Passo 4: Definição do hash como um valor em hexadecimal

O hash final são os 16 primeiros bytes de $saidaPassoTres$ mas, tradicionalmente, hashes são informados como valores em hexadecimal, portanto, os 16 primeiros bytes da saída do passo anterior precisam ser impressos na tela como uma sequência de valores em hexadecimal. No caso do exemplo que estamos considerando, o hash é:

```
7ea2319be0d038908161b4e8c26bfc7a
```

2.3 Relatório

O desempenho, em termos de tamanho e de tempo de execução, dos programas em Assembly e em Python deverá ser avaliado com os conteúdos dos 5 arquivos disponibilizados junto com este enunciado (`texto10.txt`, `texto100.txt`, `texto1000.txt`, `texto10000.txt` e `texto100000.txt`.

Para cada arquivo você deverá rodar o programa em python dez vezes e o programa em Assembly dez vezes. Ou seja, no total, o programa em python será executado cinquenta vezes e o programa em Assembly será executado cinquenta vezes. O tempo de execução em segundos deverá ser medido para cada entrada e a média das dez execuções para cada entrada deverá ser colocada em uma tabela no seu relatório conforme apresentado na Tabela 1.

Para descobrir o tempo de execução recomenda-se usar o utilitário `/usr/bin/time` da seguinte forma ao invocar o programa para uma dada entrada (por exemplo, para o primeiro arquivo da tabela):

```
lidenbrook@asteroid:~/usp/mac0216/ep1-Daniel$ cat texto100000.txt | /usr/bin/time -f "%e" ./ep1S
....
0.29 <--- Esse é o tempo em segundos retornado pelo /usr/bin/time
```

Tabela 1: Tempos de execução em segundos (média de dez execuções para cada entrada)

Entrada	**Média (Python)**	**Média (Assembly)**
`texto100000.txt`	Média 1 do programa Python	Média 1 do programa ASM
`texto10000.txt`	Média 2 do programa Python	Média 2 do programa ASM
`texto1000.txt`	Média 3 do programa Python	Média 3 do programa ASM
`texto100.txt`	Média 4 do programa Python	Média 4 do programa ASM
`texto10.txt`	Média 5 do programa Python	Média 5 do programa ASM

Tabela 2: Tamanhos dos programas em bytes

Python	**Assembly**
Tamanho do fonte Python	Tamanho do executável Assembly

Os tamanhos em bytes do fonte em Python e do executável em Assembly devem ser colocados em uma tabela no seu relatório conforme apresentado na Tabela 2.

Para descobrir o tamanho em bytes recomenda-se usar o comando `ls -l`.

Além das tabelas, o seu relatório precisa ter um cabeçalho com seu nome e NUSP e uma conclusão explicando se os resultados obtidos em termos de tempo de execução e de tamanho estão dentro do esperado ou não. As conclusões precisam ser justificadas. Também informe a configuração do computador (processador, memória e sistema operacional) onde todas as execuções foram realizadas. Todas as execuções devem ser feitas no mesmo computador para que a comparação tenha validade. Além disso, certifique-se de que, durante as execuções dos programas, apenas o terminal esteja aberto (Feche navegadores web, players de música, etc...)

2.4 LEIAME

Junto com os programas e com o relatório você terá que entregar também um arquivo LEIAME em texto puro (Ele pode ser nomeado apenas como LEIAME ou pode ser LEIAME.txt). Arquivos LEIAME (ou README) são extremamente importantes para que outras pessoas saibam quem fez o programa, qual o objetivo do programa, como compilar, como executar e quais as dependências, caso haja alguma. Portanto, crie o seu arquivo com no mínimo estas seis seções dentro dele:

```
AUTOR:
<Seu nome, NUSP e endereço de e-mail>

DESCRIÇÃO:
<Explique o conteúdo desse projeto e o que os programas fazem (Não diga
apenas que ele é o EP1 da disciplina tal, explique o que de fato os
programas fazem)>

COMO GERAR O EXECUTÁVEL:
<Informe o passo a passo de como gerar o executável do programa em
Assembly. De preferência com todas as sequências de linha de comando
necessárias>
```

```
COMO EXECUTAR:
<Informe como rodar os programas, como passar informações de entrada
para eles e como interpretar a saída>

TESTES:
<Dê exemplos de alguns pares de entradas e saídas que sejam suficientes
para que alguém confirme que os programas estão funcionando
corretamente>

DEPENDÊNCIAS:
<Informe o que é necessário para compilar e rodar os programas.
Informações como: versão do montador, versão do linker, versão do
interpretador, informações do Sistema Operacional onde você executou e
sabe que ele funciona são importantes de serem colocadas aqui>
```

Um bom arquivo de LEIAME faz parte de uma importante Técnica de Programação que deve ser exercitada sempre e que será vista com mais detalhes no decorrer do curso: **Documentação**. Os **Testes**, por si só, também representam uma outra importante Técnica de Programação que será mais exercitada adiante no curso.

3 Entrega

Você deverá entregar um arquivo tarball comprimido (.tar.gz) contendo os seguintes itens:

- 1 único arquivo .py com o código fonte do programa em Python;
- 1 único arquivo .s com o código fonte do programa em Assembly;
- 1 arquivo LEIAME (pode ser LEIAME.txt) em texto puro;
- 1 arquivo .pdf com o relatório.

Todos esses arquivos devem estar presentes. Caso algum arquivo esteja faltando, o EP não será corrigido e a nota dele será zero. O desempacotamento do tarball deve produzir um diretório contendo os itens. O nome do diretório deve ser ep1-seu_nome. Por exemplo: ep1-joao_dos_santos. Não envie arquivos que não foram pedidos (arquivos de entrada usados para testes, arquivos objeto e arquivos executáveis por exemplo não devem ser colocados no .tar.gz)

A entrega do tarball deve ser feita no e-Disciplinas. Se você nunca criou um tarball, segue abaixo um passo a passo de como você pode fazer isso supondo que todos os seus arquivos estejam no diretório `~/usp/mac0216/ep1-Daniel`:

```
# Rodando um ls para confirmar que realmente está tudo no lugar certo
lidenbrook@asteroid:~/usp/mac0216/ep1-Daniel$ ls
LEIAME        ep1.py         ep1.s         relatorio.pdf

# Descendo um nível para que o tarball seja criado com o diretório
lidenbrook@asteroid:~/usp/mac0216/ep1-Daniel$ cd ..
```

```
# Gerando o arquivo
lidenbrook@asteroid:~/usp/mac0216$ tar zcvf ep1-Daniel.tar.gz ep1-Daniel
a ep1-Daniel
a ep1-Daniel/ep1.s
a ep1-Daniel/LEIAME
a ep1-Daniel/ep1.py
a ep1-Daniel/relatorio.pdf

# Confirmando que ele foi criado sem problemas
lidenbrook@asteroid:~/usp/mac0216$ tar ztf ep1-Daniel.tar.gz
ep1-Daniel/
ep1-Daniel/ep1.s
ep1-Daniel/LEIAME
ep1-Daniel/ep1.py
ep1-Daniel/relatorio.pdf

# Confirmando mais ainda (descompactando no /tmp e indo lá ver)
lidenbrook@asteroid:~/usp/mac0216$ tar zxvf ep1-Daniel.tar.gz -C /tmp/
x ep1-Daniel/
x ep1-Daniel/ep1.s
x ep1-Daniel/LEIAME
x ep1-Daniel/ep1.py
x ep1-Daniel/relatorio.pdf
lidenbrook@asteroid:~/usp/mac0216$ cd /tmp/ep1-Daniel/
lidenbrook@asteroid:/tmp/ep1-Daniel$ ls
LEIAME        ep1.py         ep1.s         relatorio.pdf
```

O EP deve ser feito individualmente.

Obs.1: Serão descontados 2,0 pontos de EPs com tarballs que não estejam nomeados como solicitado ou que não criem o diretório com o nome correto após serem descompactados. Confirme que o seu tarball está correto, descompactando ele no shell (não confie em interfaces gráficas na hora de verificar o seu tarball pois alguns gerenciadores de arquivos criam o diretório automaticamente mesmo quando esse diretório não existe).

Obs.2: A depender da qualidade do conteúdo que for entregue, o EP pode ser considerado como não entregue, implicando em MF=0,0. Isso acontecerá também se for enviado um tarball corrompido, códigos fonte vazios ou códigos fonte que resolvam um problema completamente diferente do que foi solicitado.

Obs.3: O prazo de entrega expira às 8:00:00 do dia 4/9/2023.

4 Avaliação

80% da nota será dada pela implementação, 10% pelo LEIAME e 10% pelo relatório. Os critérios detalhados da correção serão disponibilizados apenas quando as notas forem liberadas.