Rede de mixes baseada em SGX

Pedro Miguel Nicolau Escaleira
escaleira@ua.pt

20/06/2021

Conteúdo

1 Introdução

O trabalho apresentado neste relatório foi realizado no âmbito da disciplina de Ambientes de Execução Seguros, integrada no Mestrado de Cibersegurança da Universidade de Aveiro. Nele, foram desenvolvidas competências sobre a criação de programas que fazem uso das tecnologias *Intel Software Guard Extensions (SGX)*, que permitem a anonimização das execuções desses programas ao nível do *CPU* [1].

O caso de estudo explorado foi o da criação de uma rede de *mixes* [2], que permitisse a permutação aleatória das mensagens enviadas para ela, com o intuito de não haver qualquer associação possível entre uma dada mensagem final e a ordem com que foi enviada para a rede. Sendo assim, toda a lógica associada à permutação das mensagens em cada *mix* (ou nó) foi implementada sob a forma de enclaves de *SGX*, de maneira a assegurar a privacidade da aleatoriedade das mensagens a todos os níveis. É de notar que todas as mensagens enviadas entre os vários nós da rede foram sempre cifradas e decifradas nos próprios enclaves, de forma a também evitar qualquer *eavesdropping* possível na rede.

A nossa rede de *mixes* pode-se considerar como sendo bastante básica, sendo que não existe, por exemplo, descoberta automática dos nós da rede ou resiliência na comunicação, uma vez que este trabalho é apenas uma prova de conceito de *SGX* e não propriamente um trabalho realizado no âmbito de computação distribuída. Desta forma, todos os nós são lançados por um *script* inicial, que informa cada nó do nó seguinte e anterior na rede, formando-se uma rede de *mixes* fechada em circulo. Para além disso, toda a rede é executada localmente, mas com mudanças triviais no código fonte poderá ser dada a funcionalidade de execução em qualquer máquina.

De maneira a implementarmos o código associado aos enclaves de *SGX*, usámos as bibliotecas do seu *SDK*, cuja documentação pode ser encontrada no manual [3].

Sublinhamos também que todos os resultados apresentados na secção 3 foram obtidos de execuções feitas num ambiente *SGX* simulado, disponibilizado no *SDX* do *SGX*, numa máquina virtual *Ubuntu*, ou seja, não executamos qualquer código em "modo *hardware*", por falta de recursos para tal. Neste tipo de execução, não há qualquer garantia de que a memória e a execução do enclave sejam visualizados por terceiros, mas o código produzido será, teoricamente, compatível com a execução em "modo *hardware*".

2 Implementação

2.1 Rede de mixes

A primeira parte da nossa solução passou pela implementação dos nós que constituem a rede de *mixes*. Cada um destes nós é constituído por duas componentes principais:

- O módulo **enclave**, que executa a porção confiável da aplicação. Desta forma, é nesta componente que é criado o par de chaves de cada nó, usadas para fazer a cifra e decifra das mensagens que irão circular na rede, a própria cifra duma nova mensagem a ser enviada, a decifra de cada mensagem recebida, o armazenamento das mensagens recebidas decifradas e a lógica para fazer as decisões de que tipo de mensagem será enviada e para onde será enviada.
- O módulo **aplicação**, que é responsável por fazer as comunicações de rede necessárias para a execução da rede de *mixes* e por fazer a ligação ao **enclave**, enviando para este as mensagens recebidas e obtendo do mesmo as mensagens que irá enviar e as correspondentes decisões de para quem as enviará.

2.1.1 Módulo enclave

Como já discutido, este módulo tem como propósito fazer toda a lógica do correspondente nó que está associada a secretismo. Desta forma, ele possui as seguintes funções, que podem ser chamadas pelo módulo aplicação:

- **Criação do par de chaves ("*create_keys*"):** pode ser vista como a primeira função do enclave. As razões para a necessidade desta função ser secreta são evidentes: a chave privada usada para a comunicação não deve ser, em qualquer circunstancia, pública. Desta forma, o próprio processo de criação da mesma terá de ser completamente oculto, através da execução no ambiente *SGX*. Para a criação das chaves, usamos as funções da biblioteca de criptografia do *SDK* do *SGX*[1]: "*sgx_create_rsa_key_pair*"e "*sgx_create_rsa_priv2_key*". A primeira permite, como o próprio nome indica, criar o par de chaves *RSA* própriamente dito, armazenando os valores dos respetivos parâmetros para os *buffers* passados como argumento. A segunda, permite a criação da chave privada correspondente aos parâmetros da mesma criados através da primeira função. É de notar que na nossa implementação são sempre criadas chaves de 256 *bytes* e o expoente publico é sempre 65537[2]. Por fim, a nossa função de criação do par de chaves armazena o módulo da chave pública, de 256 *bytes*, num *buffer* enviado por argumento, permitindo desta forma transmitir o módulo da chave pública para fora do enclave (para o módulo aplicação).
- **Definição da chave pública a ser usada para decifra ("*set_public_key*"):** esta função permite passar como argumento o valor do módulo duma chave pública, que irá ser usado para criar a mesma (através deste módulo e do expoente 65537). É através desta mesma função que é definida a chave pública a ser usada para cifrar as mensagens a serem extraídas do enclave (para serem enviadas para o nó seguinte da rede pelo módulo aplicação, como será apresentado na secção 2.1.2). Para obter a chave pública a partir dos parâmetros da mesma, foi usada a função "*sgx_create_rsa_pub1_key*"do *SDK* do *SGX*.
- **Importação de uma nova mensagem ("*import_message*"):** esta função, quando chamada pela aplicação externa, permite armazenar de forma segura, no enclave, uma mensagem recebida do nó anterior da rede. Como iremos apresentar na secção 2.1.2, cada nó recebe mensagens cifradas usando a sua chave pública, sendo que aquando da receção é necessário enviar a mesma para o enclave, de forma a decifrar o seu conteúdo, uma vez que a chave privada está armazenada de forma segura dentro do enclave, como já apresentado acima. Desta forma, o enclave recebe esta mensagem, decifra o conteúdo da mesma usando a sua chave privada e armazena o resultado num *buffer* global, caso a mensagem seja verídica. Caso contrário, descarta a mesma, não a armazenando, uma vez que

[1] É possível encontrar informação sobre as mesmas na subsecção *Cryptography Library* da subsecção *Trusted Libraries* da secção *Library Functions and Type Reference* do manual do desenvolvedor [3].

[2] Número primo históricamente usado como expoente em chaves públicas *RSA* e cuja utilização é tida como criptográficamente segura.

mensagens falsas servem apenas para gerar confusão na comunicação. Além disso, caso a mensagem seja verídica, a porção da mesma que indíca o valor dos *hops* por onde a mensagem já passou é incrementado[3]. Para efetuar a decifra da mensagem, usamos uma vez mais uma função do *SDK* do *SGX*, desta vez a função "*sgx_rsa_priv_decrypt_sha256*".

- **Exportação de uma mensagem ("*dispatch*"):** esta função é responsável por efetuar toda a lógica de aleatoriedade que um *mix* é suposto apresentar i.e., é ela a responsável por "baralhar"as mensagens que chegaram ao nó correspondente, de forma a que a ordem com que elas são enviadas seja muito distinta da ordem com que foram recebidas. Esta computação teve de ser feita no interior do enclave uma vez que é necessário que todas estas decisões sejam feitas de forma anónima, de maneira a não se poder obter qualquer informação de qual é a correspondência entre a ordem de chegada de uma mensagem e a ordem do seu envio pelo *mix*. Para além deste aspeto, esta função é ainda responsável por decidir se, num determinado envio, será enviada uma mensagem "falsa", ou uma mensagem verídica. Esta estratégia foi seguida de forma a camuflar as mensagens reais quando existem ainda poucas no *buffer*.

  Posto isto, a função começa por fazer a decisão de se irá fazer ou não *fan out* de uma mensagem i.e., se a irá enviar ou não para o consumidor final. Esta decisão é feita de acordo com uma probabilidade de 1%, ou seja, cada vez que esta função é chamada, há 1% de possíbilidade que uma mensagem do *buffer* seja enviada para o consumidor final. De seguida, caso tenha sido feita a decisão de ser enviada a mensagem não para o consumidor, mas sim para o próximo nó da rede, é feita a decisão de se irá ser enviada uma mensagem falsa ou verdadeira, de acordo com uma probabilidade obtida a partir da equação:

$$P(enviar\ uma\ mensagem\ falsa) = \frac{H - N}{H} \tag{1}$$

  caso $H > N$, onde H representa um dado limite de mensagens pré-definido e N representa o número de mensagens nesse instante no *buffer*. Como é possível perceber, quanto mais elevado for o valor de H, maior será a probabilidade de ser enviada uma mensagem falsa quando esta função é chamada. Para além disso, quando maior for o valor do N, menor será o valor da probabilidade de enviar uma mensagem falsa, pelo que quantas mais mensagens são recebidas e, consequentemente, armazenadas no *buffer*, menor será o valor da probabilidade. Na nossa implementação foi usado um valor de $H = 100$.

  Por fim, caso a mensagem seja enviada para o consumidor, esta será enviada em claro, não sendo cifrada. Por outro lado, caso a mensagem, falsa ou verdadeira, seja enviada para o próximo *mix*, esta irá ser cifrada usando a chave pública do mesmo e o resultado, de 256 *bytes*, armazenado num *buffer* enviado como argumento da função. Neste caso foi uma vez mais usada uma função da biblioteca do *SDK* do *SGX*: a função "*sgx_rsa_pub_encrypt_sha256*", que permite cifrar uma mensagem usando uma dada chave pública.

  É também importante referir que é possível indicar à função para fazer a libertação de uma mensagem para o consumidor de forma incondicional, de maneira a permitir ao módulo aplicação fazer *fan out* de todas as mensagens, quando necessário.

Para além dos detalhes acima mencionados, é também de notar que foi usada uma função do *SDK* do *SGX* para obter valores aleatórios, quando foi necessário fazer decisões baseadas em probabilidades e obter um valor de forma aleatória do *buffer*. A função usada foi a "*sgx_read_rand*", que armazena numa variável passada como argumento um valor obtido de forma aleatória, cujo tamanho em *bytes* é também passado por argumento.

Posto isto, foi possível criar um mecanismo que permite fazer todas as operações sensíveis do *mix* de forma secreta, usando as potencialidades do *SGX*. A aplicação externa e qualquer outra componente externa ao enclave, como o sistema operativo, nunca têm acesso à memória deste ou às operações que está a fazer no processador da máquina, permitindo neste caso que todas as decisões feitas no interior do enclave, bem como as mensagens armazenadas em claro no *buffer*, sejam secretos.

[3] Mais sobre este valor na secção 2.2.

2.1.2 Módulo aplicação

O módulo aplicação criado é responsável pelo lançamento (execução) do módulo enclave e por interagir com este quando necessário. Para além disso, é esta componente que faz toda a comunicação com as restantes entidades na rede, quer sejam outros *mixes*, quer sejam o produtor ou consumidor de dados.

Quando a aplicação é lançada, esta começa por fazer o lançamento do respetivo enclave. De seguida cria uma *thread* que irá criar uma *socket* para receber dados numa dada porta local. Posto isto, pede ao enclave que faça a criação do par de chaves, como apresentado na secção 2.1.1, obtendo desta interação o módulo da chave pública criada. Com esta aquisição, envia o módulo para o nó que se encontra antes de si na rede e para o produtor de dados, de forma a que estes possam enviar mensagens propriamente cifradas para este *mix* em especifico, que só poderão ser decifradas por ele. Na mensagem que envia ao produtor, envia também a porta de rede onde se encontra ligado, de forma a que o produtor possa associar cada chave pública recebida ao respetivo *mix*. Por fim, cria uma segunda *thread* responsável pelo envio intermitente das mensagens obtidas do *enclave* para o próximo *mix* ou para o consumidor.

O funcionamento desta ultima *thread* baseia-se no seguinte conjunto de passos iterativos:

1. Primeiro, começa por verificar, de segundo a segundo, se já recebeu o módulo da chave pública do nó que se encontra a seguir na rede, uma vez que só poderá fazer a cifra das mensagens com esta informação. Quando esta verificação se averigua como verdadeira, passa para o ponto seguinte.
2. De seguida, faz uma chamada à função do enclave que permite a extração duma nova mensagem, já descrita na secção 2.1.1. Desta chamada irá obter a mensagem que irá enviar de seguida e a informação de para quem a terá de enviar. Caso receba indicação para enviar a mensagem para o consumidor, irá abrir uma conexão, através da criação duma *socket*, com o consumidor final, e irá enviar a mensagem obtida do enclave, em *clear text*. Caso, por outro lado, receba indicação para enviar a mensagem para o próximo *mix*, irá abrir uma nova conexão com o mesmo, também através da criação duma *socket*, enviando-lhe a mensagem cifrada com a chave pública deste último.
3. Por fim, faz a verificação de se já pode terminar o seu trabalho. Esta decisão é feita com base em 3 parâmetros, sendo que todos têm de se verificar como sendo verdadeiros:
   - Se já recebeu "ordem"do produtor ou consumidor para fazer *fan out* de todas as mensagens;
   - Se já recebeu indicação do *mix* anterior na rede que este iniciou o seu processo de *fan out* de todas as mensagens, e não irá enviar mais mensagens para este *mix*;
   - Se o tamanho do *buffer* do enclave é igual a zero.

   Caso todas estas condições se verifiquem, a *thread* é terminada, sendo *joined* na *main thread* que a lançou em primeiro lugar. Caso contrário, volta a executar os passos a partir do ponto 2, inclusivé.

Nos passos apresentados acima, a *thread* é "informada"sobre se já recebeu informação para fazer "*fan out*"de todas as mensagens e indicação de que o *mix* anterior já iniciou o processo de *fan out* de todas as suas mensagens, através de duas *flags*, uma para cada caso. Estas são iniciadas com o valor "falso"no inicio da execução do módulo aplicação, sendo que passam a ter o valor "verdadeiro"assim que a primeira *thread* apresentada receba estas informações ou do consumidor/produtor, ou do *mix* anterior, como iremos de seguida apresentar. Para além disso, quando a *flag* que indica que é necessário fazer *fan out* passa a ter o valor de verdade, a função de exportação de mensagens do enclave passa a ser chamada com esta informação, o que, como já discutimos na secção 2.1.1, significa que irão ser sempre obtidas mensagens em claro do enclave, de forma a que sejam enviadas para o consumidor.

Para além destas características, é também de salientar que quando esta *thread* descrita é *joined*, a primeira *thread*, de receção de mensagens é terminada. Isto uma vez que esta última é separada da *main thread* (*detached*), sendo que quando a *main thread* termina a execução, como é o caso, a *thread* de receção de mensagens será terminada também. Desta forma, é possível terminar com sucesso a execução do módulo aplicação assim que a *thread* de envio de mensagens faça o conjunto de 3 verificações apresentadas, que corresponde ao momento em que a aplicação não irá receber mais nenhuma mensagem nem do produtor nem do *mix* anterior, nem irá enviar qualquer mensagem, uma vez que possui o *buffer* vazio.

Apresentada a linha de execução da *thread* de envio das mensagens do enclave, é necessário explicar a execução da *thread* da receção das mensagens. Esta encontra-se em ciclo a escutar por novas comunicações que poderão ser feitas à *socket* a que está conectada. Quando recebe uma mensagem, esta poderá ser uma de 4 tipos distintos, sendo que o comportamento da *thread* perante cada um deles irá ser em concordância:

- Poderá receber uma mensagem com o módulo público do *mix* seguinte na rede. Neste caso, irá armazenar o mesmo no enclave, chamando a função deste que permite o armazenamento da chave pública, e irá indicar à *thread* de envio de mensagens mesma a receção desta informação, através duma *flag*. É através desta *flag* que a *thread* de envio de mensagens do enclave sabe quando começar o seu trabalho, como já apresentado acima.
- A mensagem poderá conter uma mensagem cifrada, ou do nó de rede anterior, ou do produtor de dados. Esta, como já discutido, terá sempre 256 *bytes*, e será enviada para o enclave, através da chamada da função de importação de mensagens do mesmo, como apresentado na secção 2.1.1. Desta maneira, a mensagem poderá ser decifrada com a chave privada deste *mix* em especifico e ser armazenada num *buffer*, tudo feito de forma segura dentro do enclave.
- Poderá ser uma mensagem com indicação para começar a fazer *fan out* de todas as mensagens. Esta mensagem será recebida apenas uma vez, e será enviada pelo consumidor/produtor de dados. Aquando da receção, é colocado o valor da *flag* de *fan out* a *true* e é, de seguida, enviada uma mensagem ao *mix* seguinte de que este *mix* em especifico não irá enviar mais mensagens, uma vez que irá começar o processo de *fan out* de todas as mensagens para o consumidor final.
- Por fim, aceita também mensagens com a informação de se o *mix* anterior já iniciou o processo de *fan out* de todas as suas mensagens. Desta feita, coloca a *flag* de *fan out* do *mix* anterior a "*true*", de forma a informar a *thread* de envio de mensagens que poderá terminar a sua execução.

Como podemos entender da execução das duas *threads*, há escrita de dados no *buffer* do enclave em dois lugares distintos de forma concorrente: quando é feito o pedido de exportação duma mensagem, do lado da *thread* de envio de mensagens, poderá ser feita a remoção duma mensagem de forma aleatória do *buffer*, que é uma operação de escrita, e quando é feita a importação de uma nova mensagem, do lado da *thread* de receção de mensagens, é também feita uma operação de escrita no *buffer*. Desta forma, pode haver uma *race condition* ao *buffer* do enclave, uma vez que a escrita no mesmo pode ser feita em duas *threads* concorrentes. Sendo assim, usámos um *mutex* para fazer o acesso coordenado ao enclave nestas situações. Quando uma das *threads* chama uma destas funções do enclave, antes faz o *lock* do *mutex*, sendo que ficará à espera caso este esteja em estado de *lock* nesse momento, e depois da chamada à função ser concluída, fazem *unlock*, de forma a libertar o *mutex* e permitir a próxima execução de uma destas funções.

É também usado um outro *mutex* de forma a coordenar a execução das duas *threads*, neste caso quando a *thread* de receção de mensagens recebe uma mensagem para fazer *fan out* e consequentemente envia uma mensagem a informar o *mix* seguinte deste facto. Uma vez que esta mensagem que irá enviar indica que não irá enviar qualquer mensagem para o *mix* seguinte, tem de haver a garantia de que a *thread* de envio de mensagens não está no processo de enviar uma mensagem ao *mix* seguinte, caso contrário, esta ficará perdida, uma vez que o *mix* seguinte poderá terminar a sua execução pelo facto de não esperar mais mensagens. Desta forma, é usado um *mutex* para coordenar o envio destes dois tipos de mensagens, de forma a impedir este tipo de incongruência.

Por fim, é de realçar que as mensagens enviadas possuem todas um identificador, que permite a assimilação do tipo da mesma quando é recebida por cada *mix*, ou pelo consumidor/produtor.

2.2 Produtor e consumidor de dados

De forma a que houvessem dados a circular entre os *mixes*, criamos um produtor/consumidor de dados, neste caso feito usando a linguagem de programação *Python*. Este programa funciona simultâneamente como produtor e consumidor, produzindo as mensagens iniciais que são enviadas para a rede de *mixes*, e consumindo as mensagens finais que são *fanned out* por estes.

Sendo assim, este programa começa por lançar também duas *threads*, uma para receção de mensagens (consumidor), e outra para o envio (produtor). Assim que ele é iniciado, fica em espera até receber os

módulos públicos dos *mixes* da rede, uma vez que só poderá enviar mensagens de forma cifrada. Quando recebe esta informação, calcula a correspondente chave pública *RSA*, associando-a à porta indicada pelo *mix* (como já discutido anteriormente, cada *mix* envia ao produtor/consumidor a sua chave pública associada à porta onde escutará por novas mensagens), e começa o envio das mensagens na *thread* de envio de mensagens.

A mensagens enviadas pelo nosso *produtor/consumidor* são da forma "*TEST_<identificador>:<hop>*", onde o valor do "*<identificador>*"é obtido através dum contador, que é incrementado no envio de cada mensagem, permitindo relacionar uma mensagem à ordem com que foi enviada para a rede de *mixes*, e o valor do "*<hop>*"é usado para saber por quantos nós da rede a correspondente mensagem passou, sendo que é iniciado no produtor/consumidor a 0. No processo de envio de cada uma destas mensagens, o nosso programa faz a escolha do nó da rede para onde a enviar de forma aleatória, sendo que de seguida faz a cifra da mensagem usando a chave pública do *mix* escolhido, e enviando-lhe a mensagem. É necessário frisar que o envio quer do valor do identificador, quer do valor do *hop* são apenas para motivos de estudo, não podendo ser enviados numa situação real, onde é suposto não haver qualquer correlação possível entre uma mensagem final e a ordem com que foi inicialmente enviada.

Por fim, quando são enviadas todas as mensagens (cuja quantidade é definida aquando do inicio da execução), a *thread* de envio de mensagens irá fazer uma espera, no nosso caso de 2 minutos, antes de enviar a mensagem a todos os *mixes* para iniciarem o processo de *fan out* de todas as mensagens. Esta espera serve para permitir que as mensagens na rede de *mix* possam circular durante mais algum tempo, antes de ser feito o pedido para que todas sejam enviadas incondicionalmente para o consumidor.

Para além de receber a mensagem com o módulo da chave pública de cada um dos nós, a *thread* de receção de mensagens é também responsável por receber as mensagens em *clear text* enviadas pelos *mixes*. Uma vez que quando é feito o pedido para ser iniciado o *fan out* de mensagens, haverá uma grande quantidade destas a serem recebidas no consumidor, principalmente se houverem muitos nós na rede, o consumidor coloca as mensagens recebidas numa *queue*. Esta vai sendo esvaziada por um conjunto de 10 *threads workers*, que são inicialmente lançadas pelo consumidor. Desta forma, o trabalho associado ao tratamento de cada mensagem é feito por múltiplos *workers*, independentes do consumidor "original", impedindo o "entupimento"na receção de mensagens.

Quando todas as mensagens enviadas originalmente para a rede de *mixes* são recebidas no consumidor, este termina, permitindo que a execução do programa volte à *main thread* que o lançou. Esta irá armazenar as mensagens pela mesma ordem em que forem recebidas num ficheiro no sistema, permitindo o estudo posterior dos resultados.

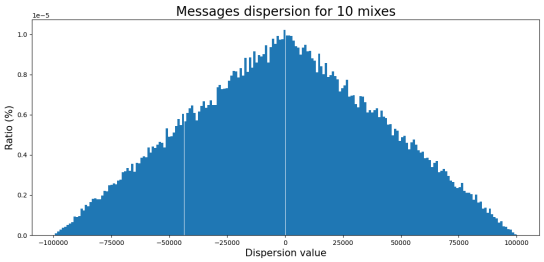
3 Resultados

Após termos feito a implementação da nossa solução, como descrito na secção 2, procedemos à execução da mesma, sob diferentes condições. Em cada teste, colocámos o produtor a enviar iterativamente 100000 mensagens para a rede de *mixes*, cujo tamanho variámos entre as múltiplas execuções, tendo sido usadas redes de 10, 100, 200 e 300 *mixes*. Para além disso, como já referido, foi usado um valor de $H = 100$ em cada *mix* e uma probabilidade de *fan out* de uma dada mensagem do *buffer* de cada nó de 1%.

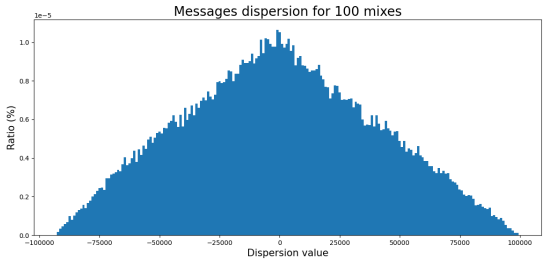
Tal como também foi referido na secção anterior, foi usado um identificador numérico em cada mensagem, indicativo da ordem com que cada uma foi enviada pelo produtor, de forma a podermos estudar qual a dispersão d, calculada para cada mensagem de acordo com a seguinte equação:

$$d = j - i \tag{2}$$

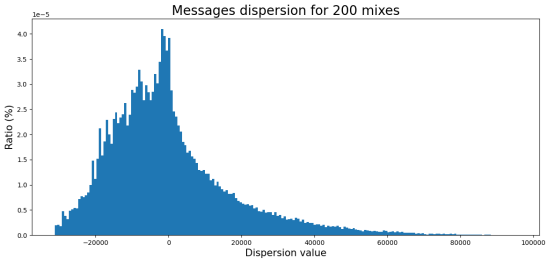
, onde j representa a ordem com que a mensagem chega ao consumidor e i representa a ordem com que ela foi enviada pelo produtor. Os resultados da dispersão obtidos podem ser analisádos através dos gráficos da figura 1.



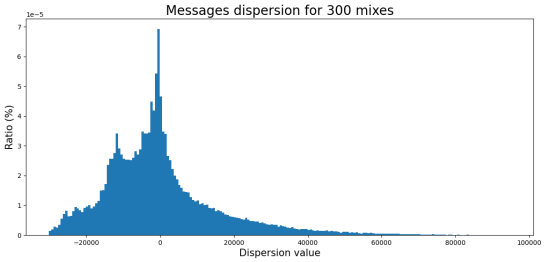
(a) Gráfico da dispersão das mensagens numa rede de 10 *mixes*.



(b) Gráfico da dispersão das mensagens numa rede de 100 *mixes*.



(c) Gráfico da dispersão das mensagens numa rede de 200 *mixes*.



(d) Gráfico da dispersão das mensagens numa rede de 300 *mixes*.

Figura 1: Gráficos da dispersão das mensagens em redes de diferentes tamanhos.

Numa permutação criada de forma aleatória, podemos observar que a probabilidade de um elemento ficar numa dada posição é igual à probabilidade de ficar em qualquer outra posição. Desta observação, poderíamos pensar numa primeira abordagem que a dispersão entre a posição final e inicial de cada elemento iria resultar em valores parecidos para cada um deles. Contudo, se compararmos a dispersão que um elemento que inicialmente esteja num dos extremos (por exemplo, na posição final ou na posição inicial) pode possuir com a dispersão que um elemento que esteja inicialmente numa posição mais central, podemos concluir que este último terá um conjunto de valores de dispersão possíveis diferente dos elementos que inicialmente se encontravam em extremos. Um elemento que se encontre, por exemplo, na posição inicial antes de ser permutado não poderá, de qualquer forma, ter um valor de dispersão menor que 0, nem maior que 100000, no nosso caso (já que é o número de elementos i.e., o número de mensagens que possuímos). Por esta linha de pensamento, um valor que esteja na posição final antes de ser permutado, irá ter uma dispersão entre -100000 e 0. Desta forma, é fácil de entender que a variação da posição inicial dum dado elemento faz variar o intervalo possível da sua dispersão final, sendo que para qualquer elemento, o seu intervalo de dispersões

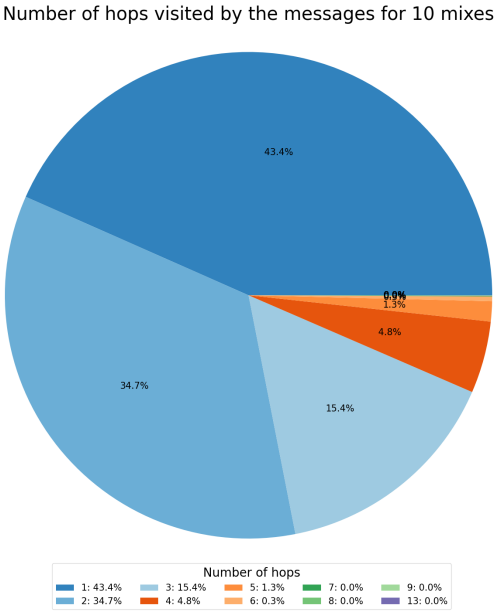
possíveis irá ter mais vezes valores mais próximos de 0 e menos vezes valores próximos de 100000 ou -100000, que são os valores máximo e mínimo que a dispersão poderá ter. Juntando todas estas inferências, é expectável que haja uma maior tendência que a dispersão de todas as mensagens possua mais valores próximos de 0 e que esta tendência dimínua à medida que nos vamos apróximando dos valores extremos da dispersão. É por esta razão que os gráficos da dispersão 1a e 1b apresentem a forma dum "triângulo", em que há mais mensagens com uma dispersão em torno do valor central 0 e menos nos extremos.

Contudo, esperávamos também que este comportamento se verificasse qualquer que fosse o tamanho da rede de *mixes*, uma vez que, como explicado, era suposto haver uma permutação quase aleatória das mensagens, mas como é possível observar nos gráficos 1c e 1d, para redes de maior dimensão, este comportamento deixa de ser a norma, sendo que não conseguimos concluir com exatidão o porquê deste novo comportamento observado. Contudo, temos algumas hipóteses para o sucedido:

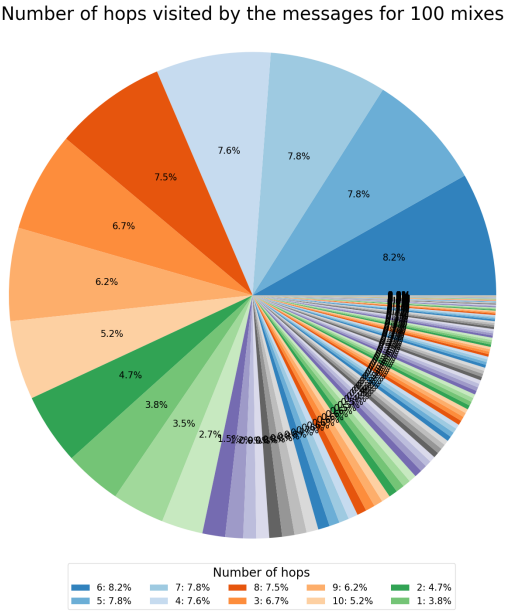
- Uma das possíveis explicações é que, pelo facto de com o aumento do tamanho da rede, haver um maior número de processos na máquina virtual usada para efetuar as medições, havendo mais processos a usufruir dos mesmos recursos comparativamente a redes de menor dimensão, tendo isto como consequência que alguns dos processos tenham mais "vantagem"do que outros no acesso a estes recursos computacionais. Há, inclusivé, uma alteração notável na performance do consumidor/produtor implementados, que para redes com tamanho maior, demoram substâncialmente mais tempo a enviar todas as mensagens para a rede. Notou-se também que, pelo facto de haver uma diminuição da performance do consumidor/produtor, quando o produtor terminava o processo de envio de todas as mensagens, havia muitas mais mensagens recebidas pelo consumidor do que quando a rede era mais pequena. Por exemplo, numa rede de 10 *mixes*, quando o produtor terminava a sua execução, o consumidor tinha apenas recebido por volta de 300 mensagens da rede de *mixes*, enquanto que quando o tamanho era 300 nós, quando o produtor terminava, já haviam cerca de 80000 mensagens recebidas pelo consumidor. Esta mudança na performance da rede foi mais notável quando a rede de *mixes* tinha 200 e 300 nós, não tendo sido notável uma diferença substancial entre as execuções de 10 e 100 nós.
- Outra possíbilidade poderá ter sido pelo facto de não termos diminuído o valor de H quando aumentamos o tamanho da rede. Quando há mais nós na rede, para um mesmo número de mensagens enviadas pelo produtor, os *mixes* irão receber, cada um, menos mensagens, possuindo, no geral, menos mensagens no *buffer* ao longo do tempo em comparação com uma rede com menos nós. Desta forma, não havendo uma diminuição do valor de H quando foi aumentado o tamanho da rede, aumentou a probabilidade de serem enviadas mensagens falsas entre os vários nós, o que pode, de alguma forma, ter induzido um resultado diferente do esperado no que toca à dispersão das mensagens recebidas pelo consumidor final.

Claro que não podemos indicar com certeza o que provocou estes resultados sem efetuar mais testes, algo que não nos foi possível dado o tempo disponível para a realização deste estudo. O desvio dos resultados para redes de *mixes* de maior dimensão pode ser um misto das hipóteses colocadas, ou nenhuma destas em particular. Futuros estudos relacionados a este trabalho poderiam passar pela execução das redes de tamanho superior com os nós espalhados por diferentes máquinas, ou em máquinas com mais poder computacional, de forma a verificar se uma melhoria da performance computacional teria como efeito a obtenção de resultados de acordo com os esperados. Seria também necessário fazer este mesmo estudo variando-se o valor de H, de forma a verificar-se o efeito do mesmo no comportamento da rede.

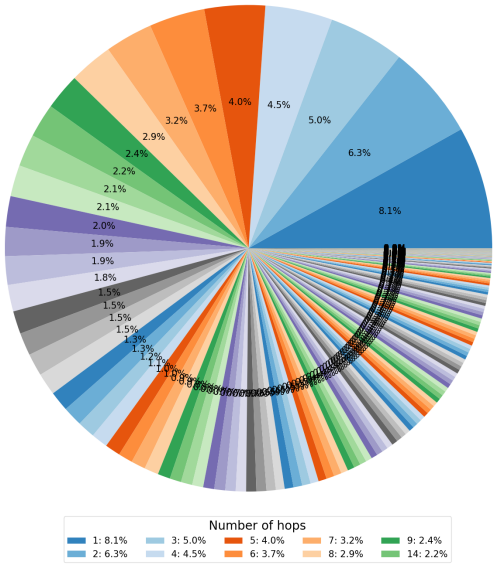
Foi também, como referido, feito o acompanhamento do número de *mixes* (*hops*) por onde cada mensagem passou, sendo que os resultados obtidos podem ser analisádos gráficamente na figura 2. Como é possível perceber, duma forma geral, com o aumento do tamanho da rede, há também um aumento do número de *hops* que as mensagens visitam antes de serem enviadas para o consumidor. Contudo, para as redes de 200 e 300 *mixes* parece haver uma atenuação do número de *hops* por onde as mensagens passam: enquanto que para uma rede de 100, houve mais mensagens a passar por exatamente 6 *hops* do que por qualquer outro número de *hops*, nas redes de 200 e 300, este valor volta a descer para apenas um *hop*. Este resultado indica uma vez mais que possívelmente o aumento do número de *mixes* na rede teve um efeito nefasto na performance geral da rede.



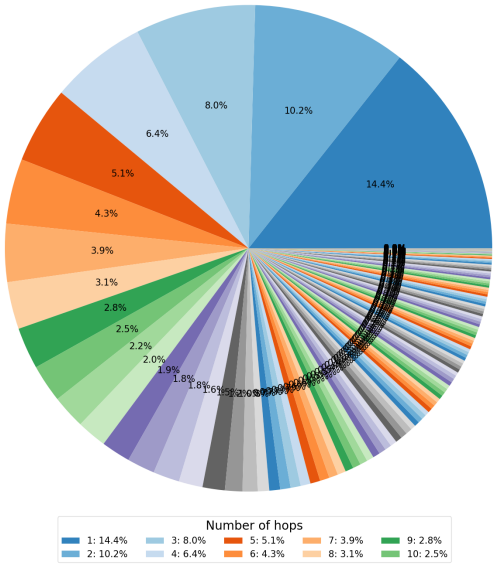
(a) Gráfico do número de *hops* visitados pelas mensagens numa rede de 10 *mixes*.



(b) Gráfico do número de *hops* visitados pelas mensagens numa rede de 100 *mixes*.



(c) Gráfico do número de *hops* visitados pelas mensagens numa rede de 200 *mixes*.

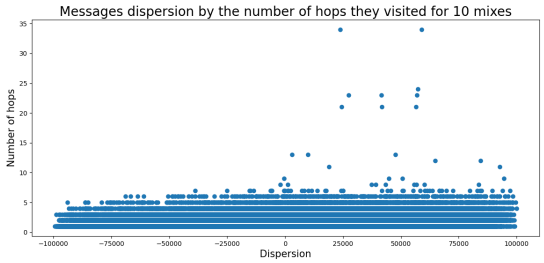


(d) Gráfico do número de *hops* visitados pelas mensagens numa rede de 300 *mixes*.

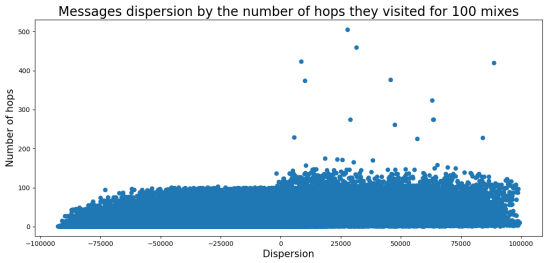
Figura 2: Gráficos do número de *hops* visitados pelas mensagens em redes de diferentes tamanhos.

Foram também obtidos gráficos da relação entre os valores da dispersão das mensagens e o número de *hops* onde as mesmas passaram, apresentados na figura 3. Nos gráficos 3c e 3d pode-se notar uma

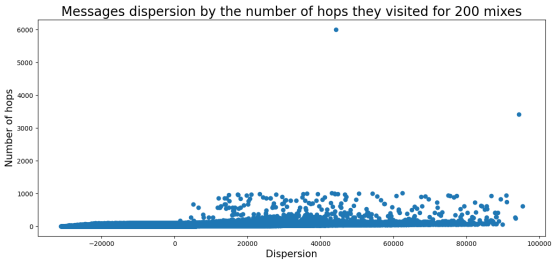
diminuição do número máximo de *hops* obtido da rede de 200 para a de 300 *mixes* que poderá ser explicada pela diminuição de performance já reparada previamente.



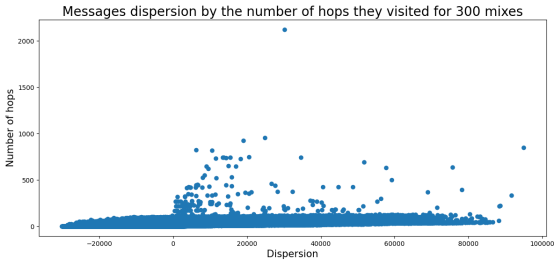
(a) Gráfico da relação entre a dispersão e o número de *hops* visitados pelas mensagens numa rede de 10 *mixes*.



(b) Gráfico da relação entre a dispersão e o número de *hops* visitados pelas mensagens numa rede de 100 *mixes*.



(c) Gráfico da relação entre a dispersão e o número de *hops* visitados pelas mensagens numa rede de 200 *mixes*.



(d) Gráfico da relação entre a dispersão e o número de *hops* visitados pelas mensagens numa rede de 300 *mixes*.

Figura 3: Gráficos da relação entre a dispersão e número de *hops* visitados pelas mensagens em redes de diferentes tamanhos.

4 Referências

[1] “Intel® software guard extensions (intel® sgx),” [Acedido em 19-06-2021]. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html

[2] “Introduction to mix networks and anonymous communication networks,” Mar 2021, [Acedido em 19-06-2021]. [Online]. Available: https://leastauthority.com/blog/introduction-to-mix-networks-and-anonymous-communication-networks/

[3] *Intel® Software Guard Extensions (Intel® SGX) SDK for Linux* OS*, 2019, ch. Library Functions and TypeReference, pp. 100–329, [Acedido em 12-06-2021]. [Online]. Available: https://download.01.org/intel-sgx/linux-2.1.3/docs/Intel_SGX_Developer_Reference_Linux_2.1.3_Open_Source.pdf