



Universidade de Aveiro

Secure, multi-player online domino
game

João Génio - 88771

Joaquim Ramos - 88812

Ruben Menino - 89185

Daniel Correia - 90480

Resumo

O seguinte relatório apresenta o desenvolvimento de um jogo de dominó seguro. Tem como principal objectivo a comunicação entre múltiplos clientes e um servidor, que vai ser a mesa de jogo, para a realização de um jogo de dominó de modo seguro. O relatório irá explicar todos os passos realizados e a sua implementação para obter a versão final do jogo.

Conteúdo

1	Introdução	3
2	Set up das sessões entre jogadores e o servidor(table manager)	4
2.1	Ligação ao servidor	4
2.2	Troca de chaves para protecção das mensagens	5
3	Set up das sessões entre jogadores	6
3.1	Troca de chaves para protecção de mensagens	6
4	Autenticidade e integridade das mensagens	9
5	Protocolo de distribuição segura	11
5.1	Pseudonimização das peças	11
5.2	Randomização das peças	12
5.3	Seleção das peças	13
5.4	Compromisso das peças	14
5.5	Revelação das peças	15
5.6	Preparação da de-anonimização das peças	16
5.7	De-anonimização das peças	17
6	Jogar o Dominó	18
6.1	Início do jogo	18
6.2	Deteção de fraude	20
6.3	Tipos de fraude	21
6.3.1	Peça jogada encontra-se na mão de outro jogador . . .	21
6.3.2	Peça não pode ser jogada à esquerda	21

6.3.3	Peça não pode ser jogada à direita	21
6.3.4	Peça foi jogada previamente	21
6.4	Escolher uma peça do stock	21
7	Instruções de utilização	23
7.1	Como correr o jogo	23
7.2	Erros	24
8	Conclusão	25

Capítulo 1

Introdução

O seguinte trabalho foi desenvolvido para a unidade curricular de Segurança (4ºano, 1º semestre) do Mestrado Integrado Engenharia de Computadores e Telemática.

Foi proposto desenvolver um jogo de dominó que joga-se as suas peças e realiza-se determinadas acções de forma segura. Consiste então num jogo, em que as 28 peças são divididas pelo número de jogadores presentes na mesa de jogo. A cada jogada joga-se uma peça que tem de combinar com a peça já colocada na mesa de jogo. Neste projecto existe um table manager, que é o servidor, e ao se conectarem o número de jogadores definidos, o jogo é iniciado.

Capítulo 2

Set up das sessões entre jogadores e o servidor(table manager)

2.1 Ligação ao servidor

O jogo começa pelos jogadores ligarem-se ao servidor, é estabelecida uma sessão tcp por jogador. As mensagens têm campos, em que alguns só existem em mensagens de um certo tipo. Os campos fixos são: "message-type", que diz qual é o tipo de mensagem; "data", tem a informação que se quer enviar. Com o servidor ligado, os jogadores enviam a mensagem do tipo "join" e a data é uma string com o seguinte texto "Client wants to joint table manager.", a que o servidor não responde de imediato. Enquanto não estiverem ligados o número de jogadores definido previamente, este não envia nenhum pacote aos clientes. À medida que os jogadores se vão ligando é lançado uma thread para cada um deles, esta thread executa a função "handle", que lida com as mensagens que recebe dos clientes, e os jogadores recebem uma mensagem do tipo "join", data = "200", um HTTP status code com significado de OK, e um novo campo "nplayers", em que é enviado o número de jogadores que vão jogar. Após estarem os jogadores ligados, se algum outro jogador tentar ligar-se este não vai receber resposta do servidor, nem é lançada a thread da função "handle". Quando os jogadores recebem a resposta do servidor estes passam para a função "handle" que lida com todas as mensagens recebidas.

2.2 Troca de chaves para protecção das mensagens

Quando todos os jogadores estão ligados, passa-se para o estágio de troca de chaves para a protecção de mensagens entre o servidor e os jogadores. O método usado para a troca de chaves é o Diffie-Hellman, foi criada uma classe "DiffieHellman", que tem como atributos a chave privada gerada através do método `ec.generate_private_key(ec.SECP256R1(), default_backend())` e a chave pública gerada através do método `self.private_key.public_key()`, `self.private_key` é o atributo chave privada. Para além disso tem métodos ainda para: gerar a chave partilhada através da chave pública do parceiro no outro lado da sessão, `shared_key(self, publicKey)`; para cifrar, `encrypt(self, publicKey, secret)`, a `publicKey` é a chave pública do parceiro no outro lado da sessão e o `secret` é o conteúdo que se pretende cifrar; para decifrar, `decrypt(self, publicKey, secret, iv)`, a `publicKey` é a chave pública do parceiro no outro lado da sessão, o `secret` é o conteúdo que se pretende decifrar, o `iv` é o vetor de inicialização usada para cifrar; para obter o vetor de inicialização após a cifragem, `getIV(self)`; a função `hmac_sha512(self, msg, publicKey)`, que serve para calcular o hash usado no HMAC para autenticação, em que `msg` é a mensagem a ser autenticada, e a `publicKey` é a chave pública do parceiro no outro lado da sessão. A fase começa por o servidor percorrer todos os jogadores e criar uma instância de "DiffieHellman", para cada, e enviar uma mensagem do tipo "diffie" em que o "data" é igual à chave pública criada na instância de "DiffieHellman", esta mensagem tem ainda o campo "client" que representa para que jogador é que está a ir a mensagem. Após enviar para todos os jogadores, o servidor espera que estes respondam. Os jogadores quando recebem a mensagem do servidor do tipo "diffie" guardam a chave pública que este envia num dicionário em que a chave é o número de jogadores deste jogo (no caso de 4 jogadores a chave vai ser 4), e criam uma instância de "DiffieHellman", e enviam a chave pública, acabada de criar com a criação da instância, para o servidor com uma mensagem do tipo "diffie", em que "data" é a chave pública criada, e no campo "client" o valor que eles receberam na mensagem que o servidor enviou previamente, ou seja, o cliente que vai enviar a mensagem. Esta fase termina quando o servidor recebe as chaves de todos os jogadores.

Capítulo 3

Set up das sessões entre jogadores

Os jogadores não comunicam diretamente, nas fases 3 e 6 do protocolo de distribuição segura é necessário que eles enviem o stock, na fase 3, e o array com as chaves públicas, na fase 6, para um jogador escolhido de forma aleatória e de forma secreta. Para isso como os jogadores não comunicam diretamente, comunicam só com o servidor, é adicionado um campo extra nas mensagens que os jogadores enviam nestas fases, mensagens estas que vão ser referidas mais à frente, e esse campo é "client" que indica ao servidor para que jogador é que a mensagem deve de ir. Como o conteúdo destas mensagens, isto é, o stock e o array não é suposto ser possível de interpretar pelo servidor, este conteúdo tem de ser cifrado. Para cifrar este conteúdo é necessário partilhar chaves entre os jogadores.

3.1 Troca de chaves para protecção de mensagens

Quando o servidor termina a fase da partilha de chaves entre jogador e servidor, este inicia então a fase da partilha de chaves entre jogadores. O método usado para a partilha de chaves entre jogadores é o mesmo usado para a partilha de chaves entre jogador e servidor, isto é, Diffie-Hellman. Os jogadores criam uma instância da classe "DiffieHellman" para cada uma das sessões entre os outros jogadores, isto é, o jogador 1 cria uma instância para

a sessão entre ele e o jogador 2, outra instância para a sessão entre ele e o jogador 3, e outra instância para a sessão entre ele e o jogador 4, no máximo só há quatro jogadores, e os outros jogadores fazem o mesmo de acordo as sessões que têm.

Para iniciar a fase de troca de chaves entre clientes, o servidor manda uma mensagem para todos os clientes menos o último, com a seguinte estrutura: "message-type": "diffie-clients", "data": , "client": i, "send": True, "save-back": False onde o campo "send": True é a flag que identifica este pacote em específico.

De seguida, todos os jogadores que receberam essa mensagem (todos menos o último), geram uma instância da classe "DiffieHellman", que é guardada num dicionário em que a chave desse dicionário é o id do cliente a quem vai ser mandada a chave. Esse dicionário criado vai ter todas as instâncias criadas para a troca de chaves deste cliente com outros. Depois mandam a sua chave pública gerada nessa instância para todos os jogadores "à sua frente", ou seja, se eu for o jogador 1, mando para o jogador 2, jogador 3 e jogador 4 (caso haja 4 jogadores). O jogador 2 manda apenas ao jogador 3 e ao jogador 4, ect ... a estrutura dessa mensagem é: "message-type": "diffie-clients", "data": serialized_public, "client": i, "send": False, "from": client, "save-back": False. O campo "data" contém a chave pública serializada, "client" é o cliente para qual vai a mensagem, "from" é o cliente que está a mandar a mensagem.

O pacote explicado em cima é então recebido pelos clientes para quem foi enviado onde é desserializada a chave recebida e guardada num dicionário em que a chave desse dicionário é o id do cliente que enviou a chave e o valor é a chave pública partilhada por esse cliente. De seguida o cliente que recebeu a chave cria uma instância da classe "DiffieHellman" que é a instância que corresponde ao agreement dos dois clientes. Essa instância é guardada no dicionário explicado em cima, que mapeia esta instância ao cliente que me enviou a chave. Por fim, a chave pública gerada na instância "DiffieHellman" criada vai ser enviada ao jogador que me acabou de enviar a sua chave. A estrutura dessa mensagem é: "message-type": "diffie-clients", "data": serialized_public, "client": package["from"], "send": False, "from": client, "save-back": True. O campo "data" contém a chave serializada, o "client" é o cliente correspondente do agreement para quem vou mandar a

chave pública, "from" é o cliente que está a mandar a chave e o "save-back" é a flag que indica ao cliente que vai receber este pacote que está neste estágio da troca de chaves (receber a chave pública do cliente para quem já mandei a minha).

Capítulo 4

Autenticidade e integridade das mensagens

Após a fase referida em cima todas as mensagens trocadas entre o servidor e os jogadores têm um campo adicional, este campo é o "hmac", em que é enviado o hash gerado a partir do conteúdo do campo "data", e a chave usada é a chave secreta da sessão entre o jogador e o servidor. Sempre que uma mensagem é enviada seja do servidor para o cliente ou do cliente para o servidor, o emissor calcula o hash com a chave partilhada, e a mensagem a ser autenticada é o conteúdo do campo data, e o recetor quando recebe calcula o hash usando a mesma chave, e verifica se o resultado é igual ao que recebeu, se não for quer dizer que a mensagem ou não veio do emissor esperado, ou os dados foram adulterados. Caso os hash não sejam iguais, o jogo é abortado. Este é o método escolhido para garantir a autenticidade das mensagens e a integridade dos dados. A função de hash usada é a SHA-512 executada através da função `"hmac_sha512(self, msg, publicKey)"` da classe "DiffieHelman", o argumento "publicKey" é a chave pública que o emissor partilhou com o recetor, e dentro da função é calculada a chave simétrica "shared_key", através da chave pública, é a chave simétrica que é usada na função de hashing da biblioteca "hmac" usada `"hmac.new(msg, self.shared_key(publicKey), hashlib.sha512)"`. Também é a chave simétrica "shared_key" que é usada nas funções de cifra e decifra referidas no ponto 2.2, para cifrar e decifrar e é calculada através da chave pública que o

emissor partilha com o recetor.

Capítulo 5

Protocolo de distribuição segura

Após a troca de chaves estar feita inicia-se o protocolo de distribuição segura.

5.1 Pseudonimização das peças

Um pseudônimo, é um valor que esconde o real valor, porém esse valor pode ser verificado quando necessário, conseguindo saber esse mesmo valor no momento inicial. Para conseguir criar pseudônimos nas peças do dominó seguimos a seguinte fórmula:

$$P_i = h(i, K_i, T_i)$$

Onde i representa o índice da peça T_i , e K_i a sua chave aleatória. São então gerados 28 pseudônimos diferentes. Essas chaves foram geradas a partir de uma função pseudo-aleatório, baseada em autenticação HMAC, onde o input com um valor salt gera e repete o processo várias vezes para produzir uma chave derivada, que vai ser então usada como uma chave criptográfica para as restantes operações efectuadas. Após criar as chaves e relacionar cada chave com um índice de uma peça, obtemos o 'dominoset' com as peças pseudonimizadas, onde só o servidor terá conhecimento dessas peças.

```
PSEUDONIMOS --> {0: b'\x19\xcb3\xd6\x00\xec-\xefp\xda\xad\xcc\x1b\x83\xa2\xd7\xcc\xeev\x00\x81\xe1\x13\xae]\xfc\xfdqF\xec\xac', 1: b'\xfa[\xe7)\x92\x9dxV\x05\x03\x00\x03'\x8f\x07\x08\x13\x01\x00\x04\x03\x0b1G\x0f6', 2: b''s'\x07\x16\xfb\x01\x0f1\x00\x0fA\x0cft\x06y\x01N\x0c\x06\x11\xcd\xcbCB?\x0c\x0c\x0c\x0c', 3: b'\x98|\xffx|n\x0a\x09\x02t\x0c\x04\x0cF\x04\x1e\x14\x19\x07\x07\x08\x07\x09fe<\xddq\x0fY2'
```

Figura 5.1: Pseudônimo para respectivo índice da peça

5.2 Randomização das peças

Após as peças estarem pseudonimizadas, e o servidor ter acesso a essas peças elas vão ser enviadas para cada cliente (do último cliente ao primeiro), onde cada um vai adicionar criptografia às peças. Para cada índice e para cada pseudônimo da peça, vai ser cifrado e vai ser guardado esses valores com essa nova cifragem. Para essa cifragem inicialmente é retornado uma string de bytes aleatórios que contém um determinado número de bytes. A esses bytes é aplicada uma cifra de bloco AES utilizando GCM (Galois Counter Mode)

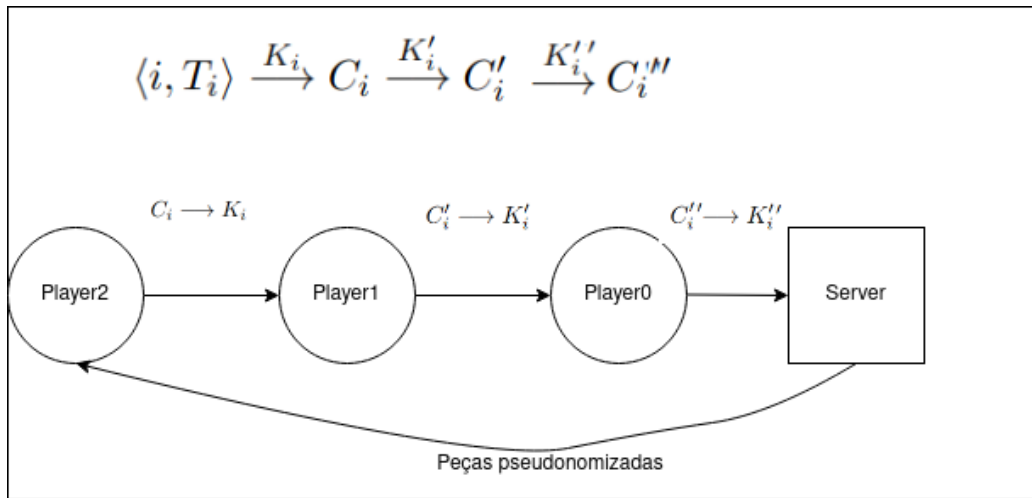


Figura 5.2: Processo de randomização das peças

Após os clientes randomizarem as peças, elas são enviadas de volta para o servidor. Neste ponto, ninguém sabe que peças é que são. É obtido então o stock, um conjunto baralhado e cifrado de peças pseudonimizadas

Inicialmente o servidor vai enviar uma mensagem do tipo "shuffle" com o conteúdo das peças pseudonimizadas "self.domino.dominoset", e com um campo firstToEncrypt a True para o primeiro cliente. O cliente


```

I picked a random tile, this tile: b'H\xba\x19\x87rc\xbca3\xe5\x8eU\xc2\x8c\xea7\xaf\x9d\x1f\x92\xcfy%\x92\x06\x00)\x91\xdc\xfcw\xa3\xfa9\xbd\x
a2\x1d'\x9c\x86g0\xde;Z\x90\xe6\x9b\xfe^e\x77\x07}yq\x02[E\x0e5\x883\xe2g\xfdz\xdf\xcd\x06\xca\x0b\xbbL\x10\x1a\xfa6M\x1c\xed\x00\x0f\x00\xfa
4\t)\xe7\xb5\xf8\x06\x03\x08\x02\xca\x00\xbb\x80\xaf\x97\x03\x01\x1a>\x92\x09\x86\xad\x06\x8f\x16\r\b\xad\x84\xdc \x03\x06\xfd\r\x89\x0d1
\x03\x0b+\xb4\xda\x9f\xdag\xed\x07\x08K\x81\xce\x92\x08N\x84\xcf\x96;\x07\x04\x1d\xbc\xa9n.Uu\xab'
Did the data came from expected entity and is it ok ? --> True
I didn't mess up the stock.
Did the data came from expected entity and is it ok ? --> True
I swaped 1 pieces!

```

Figura 5.4: Casos possíveis na seleção das peças

- **"I swaped 1 pieces!"** - quando tem peças escolhidas, trocar por por alguma no stock.

Este processo é repetido para os restantes jogadores até todos terem as peças necessárias. Esta mensagem tem o seguinte formato:

```
{ "message-type": "selection", "data": data, "players-ready": False, "hmac":
hmac, "client": package["from"], "from": package["client"], "cyphered": True,
"iv": dh.IV }
```

O campo "cyphered" serve para indicar que os dados estão cifrados, isto é, o stock é cifrado pela função de cifra da classe "DiffieHellman", com chave partilhada entre o jogador que está a cifrar e o jogador que foi escolhido ao acaso para onde o stock vai. Isto permite que o servidor ou outras entidades externas não consigam perceber o que foi feito ao stock. O campo "from" indica de que jogador veio, para o jogador recetor saber que chave vai ter que usar para decifrar. O campo "iv" é o vetor de inicialização usado para cifrar pelo emissor. O campo "client" serve para indicar ao servidor para que jogador é que a mensagem vai.

Quando todos têm as peças necessárias o campo "players-ready" passa a True, e é enviado o stock sem ser cifrado, para o servidor. Este quando recebe a mensagem termina a fase de seleção de peças.

5.4 Compromisso das peças

Para conseguir que se saiba a veracidade de cada peça jogada, é necessário, que seja guardado um valor, para que num futuro se possa comprovar a realização desse valor. Para conseguir gerar esse 'bit commitment' foi necessário gerar dois valores aleatórios (bitCommitR1 e bitCommitR2), e utilizar uma função de hash(h), onde foi utilizado o algoritmo SHA256. T representa as peças na mão de cada jogador (já com as peças de cada jogador

cifradas dos estágios anteriores).

$$b = h(\text{bitCommitR1}, \text{bitCommitR2}, T)$$

Após todos os jogadores terem calculado o bitCommitment, com a seguinte mensagem, `{ "message-type": "commitment", "username": self.username, "save": False, "data": [self.bitCommitR1, self.bitCommit], "hmac": hmac, "client": package["client"] }` vai ser partilhado o bitCommitR1 e o $b(\text{bitCommit})$ tanto para os outros jogadores como para o server.

Cada um dos clientes e o server vai guardar esses valores num no formato `package[username]: [data[0], data[1]]` onde username é o nome do jogador, data[0] representa o número gerado aleatoriamente (bitCommitR1) e data[1] representa o bitcommitment gerado.

```
Mapa dos bitcommits no server: {'ucrvvxplur': ['909023', '85cf977d34fa41850629f130103dd69f19ea2fef6c4a10fe7224a34f68b00f56'], 'zutqhcztet': ['154093', '46e96b5302101c62a10589ccbc28a45bef3b01177d5973c57cc1e67b400a22e6'], 'idtugqcrzl': ['133156', 'df032c8900ae39497e96321d2d4983a8d6926d7ac56527c52b8fd4dc44318bf1']}
```

Figura 5.5: Mapa dos bitcommitments no server

```
Mapa dos bitcommits no cliente: {'ucrvvxplur': ['909023', '85cf977d34fa41850629f130103dd69f19ea2fef6c4a10fe7224a34f68b00f56'], 'zutqhcztet': ['154093', '46e96b5302101c62a10589ccbc28a45bef3b01177d5973c57cc1e67b400a22e6'], 'idtugqcrzl': ['133156', 'df032c8900ae39497e96321d2d4983a8d6926d7ac56527c52b8fd4dc44318bf1']}
```

Figura 5.6: Mapa dos bitcommitments no cliente

5.5 Revelação das peças

Neste estágio, o servidor envia para o jogador "0" a seguinte mensagem:

```
{ "message-type": "revelation", "data": , "firstToDecrypt": True, "client": 0, "hmac": hmac }
```

que indica que este é o primeiro jogador a efetuar a ação de decifrar o stock gerado na fase anterior. Ele começa por verificar quais pares index pseudónimo foram escolhidos na fase selection (isto é, os pares que não estão no stock) e decifra-os, enviando o resultado ao servidor:

```
{ "message-type": "revelation", "data": data, "client": package["client"], "hmac": hmac }
```

em que "data" é um dicionário chave (usada na cifra/decifra do pseudónimo

da peça): pseudônimo. O servidor encarrega-se então, de enviar esse dicionário a todos os clientes (exceto o que acabou de comunicar com ele):

```
{ "message-type": "revelation-save", "data": data, "client": actualclient, "hmac": hmac }
```

Para que possam (a seu tempo) efetuar as decifras. Resta-lhe então verificar se foi o último jogador a comunicar. Se tal acontecer, termina a fase, caso contrário, informa o próximo cliente que pode iniciar o processo de decifra. Esse mesmo cliente que entretanto já recebeu as chaves dos anteriores, através da mensagem do tipo "revelation-save", e que efetua a decifra.

5.6 Preparação da de-anonimização das peças

Neste estágio, o servidor começa por informar que a fase de revelação terminou, através da mensagem:

```
{ "message-type": "revelationdone", "data": , "youare": i, "hmac": hmac }
```

Ao receber esta informação, o cliente decifra sequencialmente, e recorrendo às chaves reveladas pelos outros jogadores, os indexes e pseudônimos que escolheu na fase da seleção. De seguida é enviada pelo servidor, uma mensagem que anuncia a preparação da fase seguinte, para um cliente selecionado aleatoriamente. Essa mensagem inclui um dicionário "keys" que os clientes irão preencher com chaves públicas correspondendo com os indexes selecionados por eles na fase de seleção. Essa mensagem tem o seguinte formato:

```
{ "message-type": "anonprep", "data": jsonify_values(keys.copy()), "players-ready": False, "hmac": hmac, "client": rplayer, "cyphered": False }
```

em que o campo "client" é um cliente random para quem o servidor manda o primeiro pacote apenas e o campo "cyphered" que indica se a mensagem está cifrada ou não.

Recebida essa mensagem, o cliente verifica quantas chaves já inseriu no dicionário. Se já tiver inserido um número de chaves igual ao número de peças por jogador, então envia o dicionário de volta imediatamente (iremos ver a estrutura da mensagem de seguida). Se existirem indexes (selecionados pelo cliente na fase 3) por preencher, o cliente decide, com probabilidade de 5%, inserir uma chave (que não tenha sido inserida) no dicionário. Se essa probabilidade falhar, o cliente não efetua mudanças no dicionário. A mensagem retornada ao servidor é:

```
{ "message-type": "anonprep", "data": self.tolist(keys.copy()), "players-
```

```
ready": True, "hmac": hmac, "client": rplayer, "from": client, "cyphered":  
True, "iv": dh.IV }
```

O campo "cyphered" serve para indicar que os dados estão cifrados, isto é, o stock é cifrado pela função de cifra da classe "DiffieHellman", com chave partilhada entre o jogador que está a cifrar e o jogador que foi escolhido ao acaso, pelo jogador que está a mandar esta mensagem. Isto permite que o servidor ou outras entidades externas não consigam perceber o que foi feito ao stock. O campo "from" indica de que jogador veio, para o jogador recetor saber que chave vai ter que usar para decifrar. O campo "iv" é o vetor de inicialização usado para cifrar pelo emissor. O campo "client" serve para indicar ao servidor para que jogador é que a mensagem vai.

Se o número de chaves inseridas corresponde a "número de jogadores"* "número de peças por jogador". Se isso não acontecer, o campo "players-ready" é enviado com o valor "False", a mensagem será cifrada neste caso em que será enviada para outro cliente tendo o campo "cyphered": True e será também enviado o IV para a decifragem. Se o servidor receber essa mensagem com o campo "players-ready" a "True", termina a fase, caso contrário, envia novamente o pacote do tipo "anonprep" visto anteriormente.

5.7 De-anonimização das peças

Nesta fase, o servidor cifra, usando as chaves públicas que os jogadores revelaram na fase anterior, o par peça chave (usada na criação do pseudónimo dessa peça) e guarda as cifras num dicionário que as mapeia ao seu index ("pseudMapEncrypted[index1] = (ti, ki)") e envia a todos os jogadores:

```
{ "message-type": "deanon", "data": self.stringify_tuples( pseudMapEncry-  
ted.copy(), "hmac": hmac ) }
```

Os clientes procedem, então, à decifra desses pares, usando as chaves privadas usadas na criação das públicas anteriormente mencionadas. A partir daqui, o jogo inicia o seu ciclo de execução.

Capítulo 6

Jogar o Dominó

6.1 Início do jogo

Após o estágio de de-anonimização das peças pelas estar terminado, o servidor inicia os preparativos para iniciar todo o processo envolvente do jogo. Começa por determinar uma ordem aleatória de turnos, depois inicializa uma variável que guarda o id jogador do turno a ser jogado e por fim envia, a esse mesmo jogador, uma mensagem que indica que é o seu turno de jogar e o estado atual do "tabuleiro", que neste momento se encontra vazio. Essa mensagem tem a seguinte estrutura:

```
{ "message-type": "yourturn", "data": self.domino.board, "hmac": hmac }
```

Uma vez executado este arranque de jogo, o cliente que receber a mensagem, segue uma lógica muito específica. Começa por verificar se a mensagem inclui um campo "stockpickresult", que como iremos ver e aprofundar na secção 6.4 deste relatório, representa a peça que o jogador requisitou (previamente) ao servidor. Não existindo esse campo (que é o que acontece no caso que estamos a explorar), o cliente inicializa uma variável que representa a peça que vai jogar (representada pelo tuplo "(-1,-1)") e outra que representa o lado do tabuleiro onde irá colocar essa mesma peça (inicializada a "None"). De seguida verifica se o tabuleiro se encontra vazio, pois se estiver, a sua decisão é de escolher uma peça qualquer da sua mão (variá-

vel `"self.player.pseudodecryptedhand"`) e acrescentá-la a uma lista de peças (`"self.player.tilesplayed"`) já jogadas (que irá ter dimensão "1" no final deste turno). O lado que ele escolhe é trivial, porém, este cliente escolhe sempre o lado esquerdo nesta situação. No caso em que o tabuleiro não se encontra vazio, o cliente percorre todas as peças que lhe pertencem (no dicionário `"self.player.pseudodecryptedhand"`) e verifica se essa peça já foi jogada (na lista `"self.player.tilesplayed"`). Não tendo sido jogada, o cliente verifica se essa peça pode ser colocada no tabuleiro, testando o valor das faces exteriores das peças que se encontram nas extremidades do tabuleiro e os valores das faces da peça em questão. Se o jogador chegar ao fim da iteração do dicionário sem atualizar as variáveis `"play"` e `"side"`, ele percebe que a peça não pode ser jogada. Nesta situação ele verifica se a dimensão do stock atual (que é atualizado a cada jogada de um jogador, na variável `"self.player.stock"`), se não estiver vazio, escolhe uma peça do stock (que se encontra cifrada) e informa o servidor (iremos explorar este processo na seção 6.4 deste relatório). Se o jogador chegar ao fim deste processo e as suas variáveis correspondentes à peça e ao lado que ele irá jogar não tenham sido atualizadas desde a sua inicialização, ele envia-as tal como estão, cabendo ao servidor interpretar essa decisão como não é possível efetuar uma jogada, porém, continuando o ciclo do jogo.

A estrutura da mensagem que o jogador envia ao servidor para jogar uma peça é a seguinte:

```
{ "message-type": "play", "data": (play, side), "hmac": hmac }
```

De volta ao servidor, este recolhe informação do jogador que efetuou a jogada, tal como a peça e o lado onde ele a colocou. Se ele verificar que a peça tem o valor `"(-1,-1)"` ou que o lado é `"None"`, ele assume que o jogador não pôde jogar nenhuma peça e incrementa uma variável chamada `"self.didntplaycount"` (que controla o número de turnos seguidos em que não foram jogadas peças). Se a peça e o lado forem válidos, procede a atualizar o tabuleiro com essa peça no lado pedido, coloca a variável `"self.didntplaycount"` a zero e decrementa o número de peças disponíveis desse jogador (`"self.numtiles[self.domino.nextplayer] -= 1"`, de notar que esta informação é fruto do que o servidor consegue observar no jogo. Não são os jogadores que o informam que "gastaram" uma peça). O próximo passo é enviar uma mensagem que informa todos os jogadores do estado atualizado do jogo, que irá conter o tabuleiro atual (`"self.domino.board"`), o jogador que

efetuou a jogada ("self.domino.nextplayer", de notar que esta variável apesar de representar o próximo jogador a jogar, ainda não foi atualizada, sendo que representa neste momento, o jogador que terminou o seu turno), a peça e o lado jogados, e por fim o stock ("self.domino.stock"). Esta mensagem segue a seguinte estrutura:

```
{ "message-type": "update", "data": self.domino.board, "player": self.domino.nextplayer, "tile": tile, "side": side, "stock": self.domino.stock.copy(), "hmac": hmac }
```

Uma vez enviada essa informação, o servidor verifica as condições de término do jogo, sendo elas a variável "self.didntplaycount" ser igual ao número de jogadores ("self.nplayers") ou o número de peças do jogador que efetuou o seu turno ser igual a zero (não tem mais peças para jogar, é o vencedor).

Se o jogo não tiver condições de terminar, o servidor atualiza a sua variável "self.domino.nextplayer" para o próximo jogador que está na ordem de jogadores e informa o mesmo que é a sua vez de jogar, através da seguinte mensagem:

```
{ "message-type": "yourturn", "data": self.domino.board, "hmac": hmac }
```

6.2 Detecção de fraude

Um cliente ao receber uma mensagem do tipo "update", se esta não for resultante de um turno seu, inicia um processo de análise do tabuleiro. Listemos primeiro os erros detetáveis pelo cliente, entrando posteriormente em maior detalhe:

- Erro 1: Peça jogada encontra-se na minha mão / pertence-me;
- Erro 2/3: Peça não pode ser jogada neste local (2 - Esquerda, 3 - Direita);
- Erro 4: Peça já foi jogada previamente.

6.3 Tipos de fraude

6.3.1 Peça jogada encontra-se na mão de outro jogador

O cliente ao verificar que, num turno de outro jogador, a peça jogada encontra-se no seu registo de peças, infere que esse jogador tentou jogar uma peça que não lhe pertence.

6.3.2 Peça não pode ser jogada à esquerda

Este teste é feito para peças que são jogadas à esquerda do tabuleiro. Se a metade à direita da peça em questão tiver um valor diferente da metade esquerda da peça mais à esquerda no tabuleiro, então essa jogada é considerada uma tentativa de fraude.

6.3.3 Peça não pode ser jogada à direita

À semelhança do teste anterior, se a peça que foi jogada tiver, na sua metade esquerda, um valor diferente da metade direita da peça mais à direita no tabuleiro, então houve uma tentativa de fraude.

6.3.4 Peça foi jogada previamente

Este teste conta o número de ocorrências de uma dada peça. Se for superior a 1, então é seguro assumir que houve tentativa de fraude.

6.4 Escolher uma peça do stock

Nesta secção iremos explorar o processo, anteriormente mencionado, de "retirar" uma peça do stock. Relembrando, a necessidade de retirar uma peça do stock provém da inexistência de peças válidas (i.e. que possam ser colocadas no tabuleiro) na mão de um jogador. Neste cenário, o jogador seleciona o primeiro pseudónimo de peça (e index correspondente) do stock (que é atualizado em todas as mensagens "update" na variável "self.player.stock") e apaga-a do mesmo. Neste ponto o index/pseudónimo de peça estão cifrados tantas vezes quanto o número de jogadores. De seguida, o cliente faz um pedido ao servidor, no formato:

```
{ "message-type": "stockpick", "data": self.stringify_keys( c_index: c_pse
```

```
ud ), "iam": self.player.pid, "stock": self.player.stock.copy(), "hmac": hmac
}
```

O servidor ao receber esta mensagem inicia o processo de "stock picking", começando por atualizar o seu stock, e de seguida, enviando para o jogador "0" uma ordem de decifra de um par index e pseudónimo de peça, no formato:

```
{ "message-type": "stockpickdecrypt", "data": data }
```

Onde "data" é o campo recebido anteriormente, que contém o index e pseudónimo de peça cifrados. O jogador "0", ao receber esta ordem, efetua a decifra usando as chaves que usou na fase de randomização e retorna ao servidor, os resultados, na seguinte mensagem:

```
{ "message-type": "stockpickresponse", "data": self.stringify_keys( c_index
: c_pseud ), "iam": self.player.pid, "hmac": hmac }
```

No servidor, é verificado se este pacote vem do jogador N-1 (último jogador a conectar-se/primeiro a efetuar a cifração na fase de randomização). Se não acontecer, repete-se a ordem de decifra ao jogador seguinte (Se o jogador "0" efetuou a decifra, então o próximo é o "1"). Se o contrário for verificado, o index e pseudónimo de peça recebidos correspondem, respetivamente, ao index não cifrado (ou seja, um número inteiro) e o pseudónimo da peça não cifrado. Resta apenas ao servidor verificar na sua variável "self.domino.pseudonymMap" o valor da peça correspondente a esse pseudónimo e enviar esses dados ao jogador que efetuou o pedido:

```
{ "message-type": "yourturn", "data": self.domino.board, "stockpickresult":
index: tile , "hmac": hmac }
```

E o jogo continua o seu ciclo.

Capítulo 7

Instruções de utilização

7.1 Como correr o jogo

Inicialmente é necessário ter uma versão instalada de python. Para correr o jogo vai ser necessário instalar vários módulos utilizados:

- `pip3 install cryptography`
- `pip3 install pycryptodome`

Para alterar o número de jogadores é necessário alterar no ficheiro `server.py`, linha 873, o último argumento, que corresponde ao número de jogadores (`nplayers`).

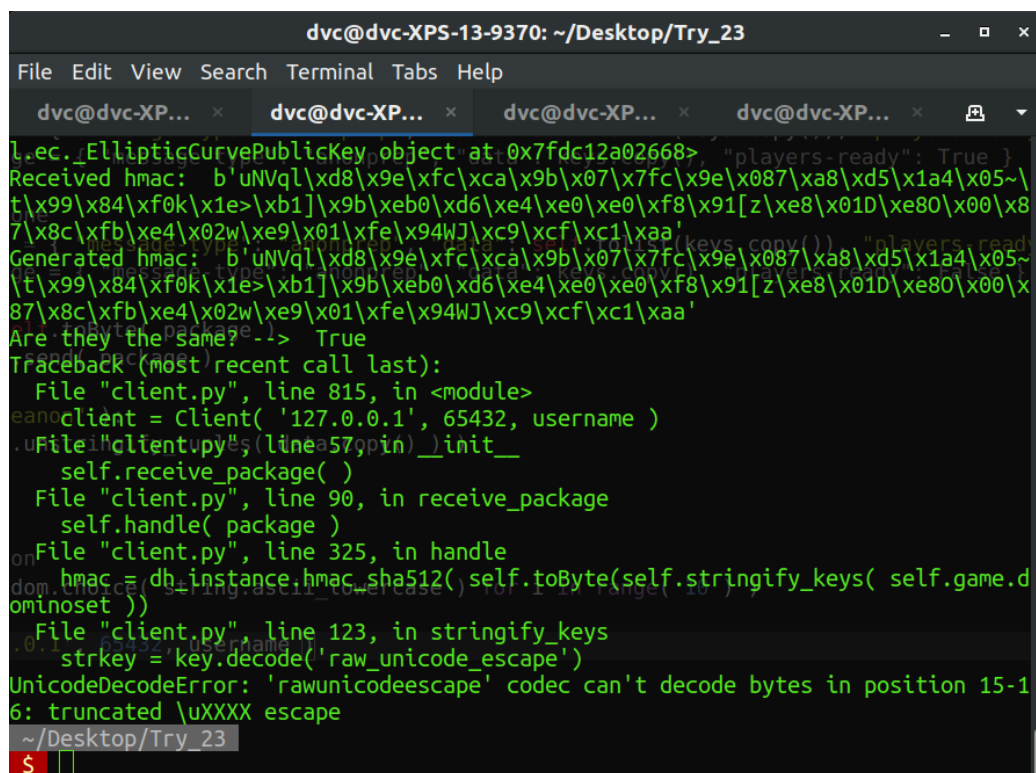
Para correr o programa teremos de iniciar o servidor, e cada um dos clientes.

No terminal do servidor: `python3 server.py` Nos restantes terminais: `python3 client.py`

7.2 Erros

No decorrer da implementação foi-nos gerado um erro ao qual não conseguimos resolver. Esse erro é gerado somente algumas vezes quando iniciamos um cliente. Provavelmente deve-se ao facto de algumas sequências de bytes não conseguirem ser processadas e não conseguirem serem enviadas nos pacotes.

Se o erro aparecer, é só correr o programa novamente.



```
dvc@dvc-XPS-13-9370: ~/Desktop/Try_23
File Edit View Search Terminal Tabs Help
dvc@dvc-XP... x dvc@dvc-XP... x dvc@dvc-XP... x dvc@dvc-XP... x
l.ec._EllipticCurvePublicKey object at 0x7fdc12a02668> "players-ready": True }
Received hmac: b'uNVql\xd8\x9e\xfc\xca\x9b\x07\x7fc\x9e\x087\xa8\xd5\x1a4\x05~\
t\x99\x84\xf0k\x1e>\xb1]\x9b\xeb0\xd6\xe4\xe0\xe0\xf8\x91[z\x9e8\x01D\xe80\x00\x8
7\x8c\xfb\xe4\x02w\xe9\x01\xfe\x94WJ\xc9\xcf\xc1\xaa'
Generated hmac: b'uNVql\xd8\x9e\xfc\xca\x9b\x07\x7fc\x9e\x087\xa8\xd5\x1a4\x05~\
t\x99\x84\xf0k\x1e>\xb1]\x9b\xeb0\xd6\xe4\xe0\xe0\xf8\x91[z\x9e8\x01D\xe80\x00\x
87\x8c\xfb\xe4\x02w\xe9\x01\xfe\x94WJ\xc9\xcf\xc1\xaa'
Are they the same? -> True
Traceback (most recent call last):
  File "client.py", line 815, in <module>
    client = Client( '127.0.0.1', 65432, username )
  File "client.py", line 57, in __init__
    self.receive_package( )
  File "client.py", line 90, in receive_package
    self.handle( package )
  File "client.py", line 325, in handle
    hmac = dh_instance.hmac_sha512( self.toByte(self.stringify_keys( self.game.d
    dom.choice( string.ascii_lowercase ) for i in range( 10 ) )
    ominoset ))
  File "client.py", line 123, in stringify_keys
    strkey = key.decode('raw_unicode_escape')
UnicodeDecodeError: 'rawunicodeescape' codec can't decode bytes in position 15-1
6: truncated \uXXXX escape
~/Desktop/Try_23
$
```

Figura 7.1: Erro de execução 1

Capítulo 8

Conclusão

Com a realização do projeto conseguimos implementar um jogo seguro, utilizando todo o conhecimento aprendido nas aulas teóricas e práticas. Relativamente à inclusão da utilização do Cartão de Cidadão não foi conseguida na versão final do projeto, devido a alguns erros durante a implementação da mesma.