

Privagic benchmark

Eloi Besnard, Joshua Randria

Télécom SudParis, Institut Polytechnique de Paris, France

ASR 2022/23

Abstract

Notre PFE s'articule autour de la thèse de Subashiny Tanigassalame qui est encadrée par Gaël Thomas à Télécom SudParis. Notre travail consistait en la réalisation d'une banque distribuée et sécurisée, servant de test pour les benchmarks de la thèse de Subashiny. La technologie gRPC (google Remote Procedure Call) permet à cette banque d'être distribuée. La technologie Intel SGX (Software Guard Extension) permet à la banque de se loger dans une enclave sécurisée.

Contents

1	Introduction	3
1.1	Intel SGX	3
1.2	Privagic	3
1.3	But du PFE	3
1.3.1	Développement	4
1.3.2	Evaluation	4
2	Notre travail	5
2.1	Architecture globale	5
2.1.1	Client	5
2.1.2	Serveur	5
2.1.3	Enclave	6
2.1.4	Divers	6
2.2	Etapes	7
2.2.1	Code de départ:	7
2.2.2	Code avec Hashmap	8
2.2.3	Code avec gRPC	8
2.2.4	Code avec SGX	8
2.2.5	Fusion SGX et gRPC	8
2.2.6	Makefile SGX et gRPC	9
3	Partie technique	10
3.1	gRPC	10
3.2	Intel SGX	11
3.3	Hashmap	12
3.4	Makefile	13
3.5	Injecteur de charge	14
3.6	Graphiques	14
4	Résultats	16
4.1	Implémentation gRPC sans SGX	17
4.2	Implémentation gRPC avec SGX	18
4.3	Analyse des résultats	19
5	Conclusion	20
5.1	Objectifs atteints	20
5.2	Suite du projet	20
6	Bibliographie	21

1 Introduction

Dans un premier temps, une brève mise en contexte avec Intel SGX, Privagic et le but de ce PFE. Ce papier détaille ensuite notre travail, en mettant l'accent sur l'architecture globale et les différentes étapes. La partie technique est ensuite explorée. Enfin interviennent les résultats.

1.1 Intel SGX

La technologie Intel SGX permet au système d'exploitation de créer des enclaves, zones de mémoire protégées au sein de la RAM. Cette technologie répond à des besoins de sécurité et de confidentialité notamment dans les domaines du *cloud computing*. Néanmoins, pour distinguer quelle partie du code doit être exécuté depuis une enclave ou en-dehors, il faut séparer manuellement les données (rendant la tâche délicate, fastidieuse et susceptible d'être mal faite).

1.2 Privagic

Pour simplifier le processus, les travaux de Subashiny permettent à l'utilisateur de distinguer le code à l'aide d'annotations élémentaires.

```
1 struct bank_account {  
2     char* name[256] color (red);  
3     int val color (blue);  
4 }
```

Figure 1: Code avec les annotations

Dans la figure 1, le `bank_account` composé d'une chaîne de caractère `name` et d'un montant `val` sont séparés dans 2 enclaves différentes à l'aide des annotations `color`.

Ces annotations sont lues par Privagic, une surcouche du compilateur LLVM. A l'aide des annotations lues, Privagic peut discerner le code à exécuter dans ou en dehors de l'enclave.

Ici, notre programme *Bank.c* est compris dans le *Prog.c*. Le front-end du compilateur s'occupe de donner un fichier intermédiaire: *Prog.ll*. Privagic entre en jeu et sépare *Prog.ll* en d'autres fichiers en *.ll* (*untrusted.ll*, *blue.ll*, *red.ll*), cette fois-ci distingués en fonction de leur exécution ou non dans une enclave.

1.3 But du PFE

Le but du PFE était d'évaluer les performances de l'outil Privagic à l'aide de plusieurs implémentations d'une banque répartie, ainsi que d'outils pour mesurer certains critères de performance. Au début du PFE nous avons, avec l'équipe encadrante, défini trois objectifs:

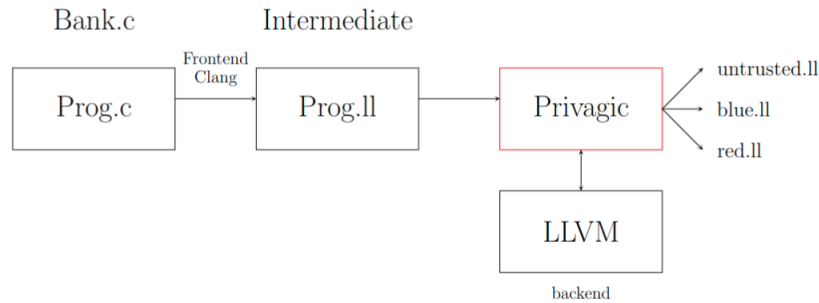


Figure 2: Privagic

1.3.1 Développement

Coder deux implémentations différentes:

- Baseline
Implémentation servant de point de départ pour coder les autres, il s'agit d'un programme simulant le fonctionnement d'une banque avec une architecture serveur client reposant sur la technologie gRPC et une hashmap "maison".
- Bank SGX
Implémentation utilisant le SDK SGX de Intel permettant de sceller la partie serveur responsable d'accéder à la hashmap.

Le développement de ces différentes implémentations a pour objectif de permettre une comparaison des performances et de la quantité de code.

1.3.2 Evaluation

Afin d'évaluer les différentes implémentations, l'obtention de graphiques était attendu. Pour cela nous avons besoin des outils suivants:

- Un injecteur de charge
Depuis le côté client, en s'inspirant de la spécification TCP-B [6], l'injecteur de charge consiste à effectuer un grand nombre de requêtes de type création de compte, crédit et débit de compte différents.
- Un script permettant d'obtenir des graphiques
Utilisant la librairie pandas de python une version graphique représentant l'évolution du débit (throughput) et de la latence (latency) permet d'illustrer de manière simple les performances des implémentations.

2 Notre travail

2.1 Architecture globale

L'architecture globale s'articule en trois parties: Premièrement le client qui effectue les requêtes au serveur, deuxièmement le serveur qui reçoit et traite les requêtes du client et pour finir l'enclave SGX qui contient la hashmap et les fonctions permettant lecture et écriture.

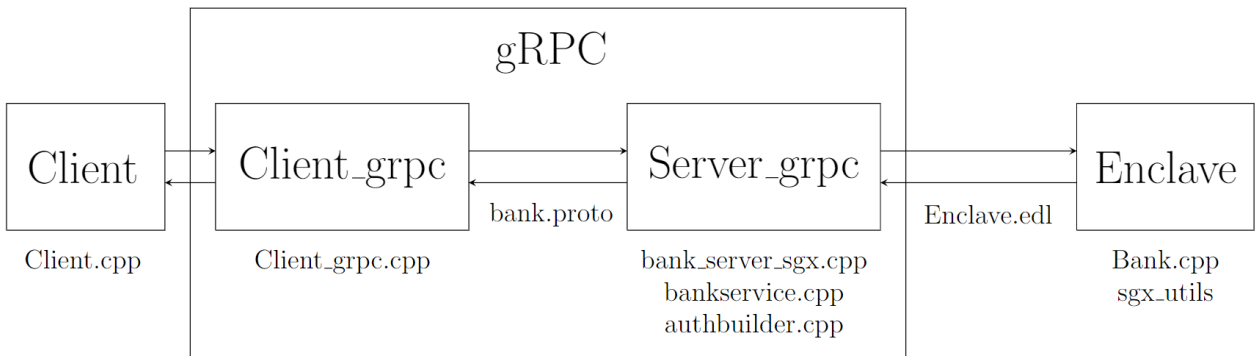


Figure 3: Architecture globale

2.1.1 Client

La partie client est constituée de trois fichiers:

- **Client.cpp**
Il contient la méthode `main` avec l'injecteur de charge.
- **Client_grpc.c** et **Client_grpc.h**
Les fichiers contenant les méthodes appelées dans le `main` sous la forme adaptée pour utiliser gRPC.

Pour faire le lien entre le client et le serveur avec gRPC, on doit spécifier les règles dans le fichier `bank.proto` qui contient les différents services et les messages de type requête et réponse.

2.1.2 Serveur

La partie serveur contient six fichiers:

- **bank_server_sgx.cpp** et **bank_server_sgx.h**
Contenant la méthode `main` avec l'initialisation du serveur et l'écoute des requêtes du client. On initialise l'enclave dans ces fichiers.

- `authbuilder.cpp` et `authbuilder.h`
Ces fichiers permettent d'initialiser le server de façon sécurisée, mais nous n'exploitons pas cette fonctionnalité dans notre implémentation.
- `bank_service.cpp` et `bank_service.h`
Ces fichiers contiennent les méthodes transmettant l'action à effectuer à l'enclave et retournant la réponse à renvoyer au client.

Les appels à l'enclave depuis le serveur sont traités grâce aux parties `untrusted` et `trusted`, représentées après compilation

2.1.3 Enclave

Dans l'enclave on a:

- `Enclave.edl`
C'est dans ce fichier qu'on définit les règles pour définir les règles d'accessibilité à l'enclave.
- `Bank.cpp` et `Bank.h`
On définit à la fois les méthodes pour manipuler la hashmap et également celles spécifiques à la banque, utilisant les précédentes.
- `sgx_utils.cpp` et `sgx_utils.h`
Ces fichiers contiennent les méthodes usuelles pour l'utilisation de SGX.

2.1.4 Divers

De plus on a quelques fichiers permettant le bon fonctionnement du code.

- `graph.py`
Script python permettant de tracer les graphiques de performances.
- `Makefile`
Fichier permettant la compilation automatisée et non triviale pour l'emploi simultané de gRPC et SGX.
- `tenkusernames.txt` et `names.txt`
Bases de données pour la création en masse de comptes fictifs.
- `Enclave_private.pem` et `Enclave.config.xml`
Fichiers nécessaires pour utiliser SGX.
- `README.md`

Sur la figure 4 on peut suivre les requêtes effectuées du client vers l'enclave. Une fois le client Cli créé on appelle la méthode `Create_Account()` qui va entraîner via le framework gRPC la requête `CreateAccount()` envoyée au serveur. Ce dernier va ensuite faire un appel à la méthode `account_create()` de l'enclave en spécifiant l'eid ou identité de l'enclave. L'accès à cette méthode depuis le

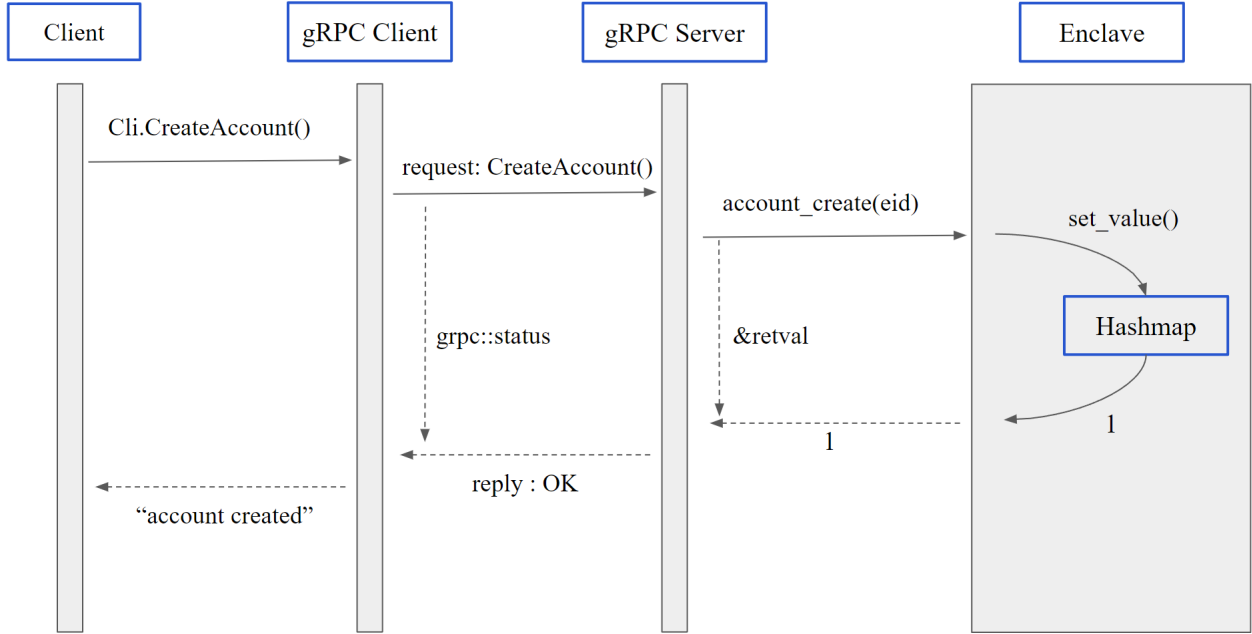






Figure 4: Diagramme de séquence du fonctionnement global

côté untrusted vers le côté trusted, l'enclave exécute alors en interne l'opération demandée à savoir `set_value(0)` pour un nouveau noeud de la hashmap. Elle écrit alors dans le pointeur soumis lors de la requête pour retourner la valeur 1 signifiant le succès de l'opération. Le status gRPC peut alors être mis à jour avec une réponse envoyée du serveur au client. Enfin on peut afficher que le compte a bien été créé.

2.2 Etapes

Cette section va nous permettre de comprendre comment nous avons construit le projet, étape par étape.

2.2.1 Code de départ:

Hashmap: , gRPC: , SGX: , MKFL: 

En premier lieu, nous avons reçu un squelette de code de nos encadrants, avec un service `Bank_create` et un service `create_account`. Une hashmap à adressage ouvert (*khash*) d'une bibliothèque *klib* [7] servait de structure à la banque. Mais

cette dernière étant trop complexe, notamment en créant des *shared objects*, Privagic aurait pu être inutilement ralenti à cause de cela. Un premier but a donc été de reconstruire une hashmap simple mais fonctionnelle.

2.2.2 Code avec Hashmap

Hashmap: ✓, gRPC: ☹, SGX: ☹, MKFL:✓

Nous avons donc repris le code existant et avons implémenté une hashmap standard. Voir la section Hashmap pour son implémentation et fonctionnement.

2.2.3 Code avec gRPC

Hashmap: ✓, gRPC: ✓, SGX: ☹, MKFL:✓

Une fois que la hashmap fût fonctionnelle, nous avons ajouté les méthodes manquantes au fonctionnement basique d'une banque: `add_amount`, `sub_amount` pour ajouter ou enlever de l'argent d'un compte et `list_account` pour avoir un retour sur le contenu de notre banque. Nous avons modifié les méthodes existant déjà (`Bank_create` et `account_create`) pour les rendre compatibles avec notre hashmap. Nous avons ajouté ces fonctions au `Bank.proto` de notre projet. Egalement, nous avons généré les méthodes gRPC faisant office de proxy entre le client et le server. Nous avons pu faire nos premiers tests de latence et de débit avec cette version fonctionnelle de notre banque distribuée. Voir la section gRPC pour son implémentation et fonctionnement.

2.2.4 Code avec SGX

Hashmap: ✓, gRPC: ☹, SGX: ✓, MKFL:✓

Pour cette partie, nous avons débarrassé notre code de tout élément gRPC pour se concentrer sur SGX. Nous avons lu de la documentation sur cette technologie [3] [4] et avons utilisé un `hello-world` pour débiter [5]. A la fin de cette étape, nous avons réussi à loger la banque dans une enclave et la faire fonctionner de la même manière qu'auparavant.

Voir la section Intel SGX pour plus de détails sur le fonctionnement général de l'enclave.

2.2.5 Fusion SGX et gRPC

Hashmap: ✓, gRPC: ✓, SGX: ✓, MKFL:☹

Cette étape fût assez alambiquée à achever. L'objectif était de fusionner notre code existant avec gRPC à celui de SGX. C'est à cette étape que nous avons repris les fonctions de la partie proxy gRPC et que nous y avons intégré les fonctions définies dans le `Enclave.edl`.

En résumé synthétique, nous sommes passés de :


```

1 grpc::CreateAccount()
2 {
3     account_create(account_name);
4     return grpc::Status::OK;
5 }

```

Figure 5: avant fusion

à:

```

1 grpc::CreateAccount()
2 {
3     account_create(global_eid, &retval, account_name);
4     return grpc::Status::OK;
5 }

```

Figure 6: après fusion

Ici, `CreateAccount` fait référence à la partie gRPC alors que `account_create` est la partie SGX. Evidemment, il a fallu modifier nos fonctions et notre architecture en profondeur pour que cette permutation fonctionne. Néanmoins, impossible de faire fonctionner la banque. En effet, nous étions incapables de compiler le projet. Nous menant à la dernière étape...

2.2.6 Makefile SGX et gRPC

Hashmap: ✓, gRPC: ✓, SGX: ✓, MKFL:✓

Cette partie fut la plus complexe, même arrivés à la fin du projet. Le code semblait opérationnel mais il nous manquait l'étape de compilation. Nous détaillons cette partie dans la section Makefile. A la fin de cette étape, nous avons pu lancer clients et server et réaliser avec succès nos premières transactions de manière distribuée à une banque sécurisée!

3 Partie technique

Cette partie décrit avec plus de détails les différentes technologies utilisées à travers ce PFE.

3.1 gRPC

gRPC est un framework open-source qui gèrent les appels entre client et server. Il s'inscrit en tant que middleware et supporte de nombreux langages de programmation.

Pour définir les méthodes appelées entre le client et le server, gRPC met en place un fichier dont l'extension est le *.proto*. Ce fichier sert de référence aux méthodes utilisées et c'est ici que nous les définissons.

```
service bank {  
  rpc CreateAccount(AccountCreateReq) returns (AccountCreateResp);  
  rpc AddAmount(AmountAddReq) returns (AmountAddResp);  
  rpc SubAmount(AmountSubReq) returns (AmountSubResp);  
  rpc ListAccount(Empty) returns (Empty);  
}
```

Figure 7: Proto service

Le regroupement de ces méthodes est fait dans un *service* que nous avons appelé *Bank*. La définition d'un *remote procedure call* se fait ainsi:

`rpc nom_methode(nom_methodeReq) returns (nom_methodeResp)`

On peut ensuite définir les éléments *Req* et *Resp* à l'aide de *messages*.

```
message AmountAddReq {  
  string account_name = 1;  
  int64 amount = 2;  
}  
  
message AmountAddResp {  
  int64 new_amount = 1;  
}
```

Figure 8: Proto message

Dans cet exemple, on définit la requête *AmountAddReq* qui comporte un *string* et un *int*. Ils correspondent respectivement au nom de la personne à qui on souhaite créditer de l'argent et au montant de cet argent.

En définitive, c'est ainsi que nous avons créé notre **Bank.proto**.

3.2 Intel SGX

Intel SGX est un ensemble d'instructions permettant au système d'exploitation de chiffrer des zones de mémoires RAM et d'exécuter du code avec. De plus, le déchiffrement se fait à la volée, à l'intérieur du CPU. Cela permet d'assurer qu'aucun autre programme ne puisse avoir connaissance du contenu de l'enclave.

Trusted / Untrusted: L'environnement d'exécution d'Intel SGX contient plusieurs parties, dont le Untrusted Run-Time System (uRTS) et le Trusted Run-Time System (tRTS). L'uRTS est la partie externe à l'enclave. Elle gère les appels à l'enclave appelés E-call (Enclave-call). La tRTS est la partie interne à l'enclave. Elle reçoit les E-call et envoie des O-call (Outside-call).

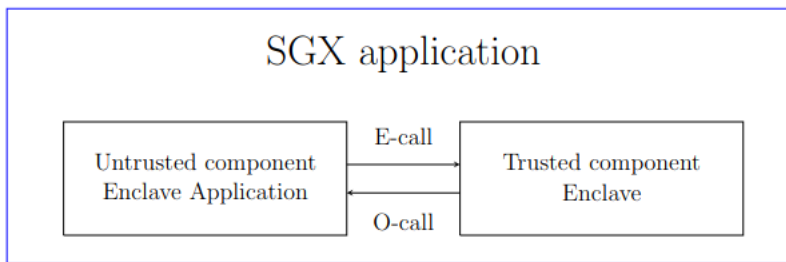


Figure 9: Enclave SGX

Les E-call et O-call sont définis dans un fichier dont l'extension est *.edl*, pour Enclave Definition Language.

```
enclave {
    include "Bank.h"
    trusted {
        public int bank_create([in] int* capacity);
        public int account_create([in, string] const char* account_name);
        public int amount_add([in, string] const char* account_name, [in] int* amount);
        public int amount_sub([in, string] const char* account_name, [in] int* amount);
        public int list_accounts();
    };
    untrusted {
        void ocall_print([in, string] const char* str);
    };
};
```

Figure 10: Enclave.edl

Enclave.edl: Ce fichier Enclave.edl contient les mêmes méthodes que le bank.proto vu précédemment. Néanmoins, il contient en plus la méthode *bank_create*. Cette méthode permet de créer la banque depuis l'extérieur de l'enclave, mais est seulement appelée par le serveur (*bank_server_sgx.cpp*). On voit distinctement les méthodes *trusted* et *untrusted*. Les arguments entre *[]* indiquent dans

quel sens sont copiées les données. Pour un E-call, *in* signifie que l'on copie de la partie *untrusted* à *trusted* (soit un *write*) et *out* l'inverse (un *read*). Cela va dans l'autre sens pour un O-call. Ainsi, puisque nous copions des données reçues depuis le serveur dans l'enclave, nous ajoutons l'argument *in* à nos méthodes *trusted*. Ensuite, la méthode *ocall_print*, appelée depuis l'enclave, possède l'argument *in* également car elle copie des données de la zone *trusted* vers la zone *untrusted*. *String* indique que la variable est une string en C.

Initialisation Enclave: L'enclave est initialisée à l'aide de la fonction nommée `initialize_enclave`. Cette étape est réalisée dans le `main` du serveur.

E-call: Les fonctions *trusted* ainsi que `initialize_enclave` comportent le nombre d'arguments choisi dans `Enclave.edl` plus deux autres arguments: un identifiant pour l'enclave `global_eid` et une valeur de retour `retval`. Ces nouvelles fonctions que l'on utilise sont générées par `Enclave_u.c` et `Enclave_t.c`. Ces fichiers sont générés lors de la compilation par le `makefile`, et s'appuient sur `Enclave.edl`.

3.3 Hashmap

La gestion des comptes de la banque se fait à l'aide d'une structure appelée *Hashmap*. A la fin de notre projet, cette dernière est entièrement contenue dans l'enclave et l'appel à ses méthodes est fait au sein de l'enclave.

Définition: une *hashmap* ou table de hachage est une structure de données qui permet une association clef / valeur. Sa capacité est son nombre d'entrées. Sa longueur est le nombre total de paires (clef/valeur) qu'elle contient.

Ainsi, on accède aux comptes par leur clef (le nom) et on peut ainsi trouver leur valeur (leur montant). Pour réduire la taille des entrées du tableau qui contient tous les comptes, on *hash* la clef du compte.

Définition: *hasher* [1] signifie passer d'un ensemble de données de longueur variable (ici le nombre de compte) à un ensemble de données de longueur fixe (longueur égale à la capacité de la banque). De plus, l'ensemble d'arrivée n'est censé ne rien avoir en commun avec l'ensemble de départ. Finalement, il est à noter qu'il est très facile à partir d'une clef d'obtenir son *hash*, mais très difficile de faire l'inverse.

Nous effectuons ce *hash* grâce à une fonction `hashcode` [2], que nous avons trouvé sur internet. Ainsi, on peut passer d'une banque avec 100 000 comptes à une table de hachage à la capacité que l'on désire.

Ensuite, pour naviguer et retrouver le compte voulu, nous utilisons une liste chaînée par entrée de la table.

Pour changer la valeur d'un compte ou ajouter un compte, nous utilisons la fonction `set_value`. Cette fonction parcourt la liste chaînée correspondant au *hash* du compte donné. Si le compte existe déjà, elle change sa valeur.

Sinon, elle l'ajoute à la fin de la liste chaînée.
 Pour obtenir la valeur d'un compte, nous utilisons `get_value`.

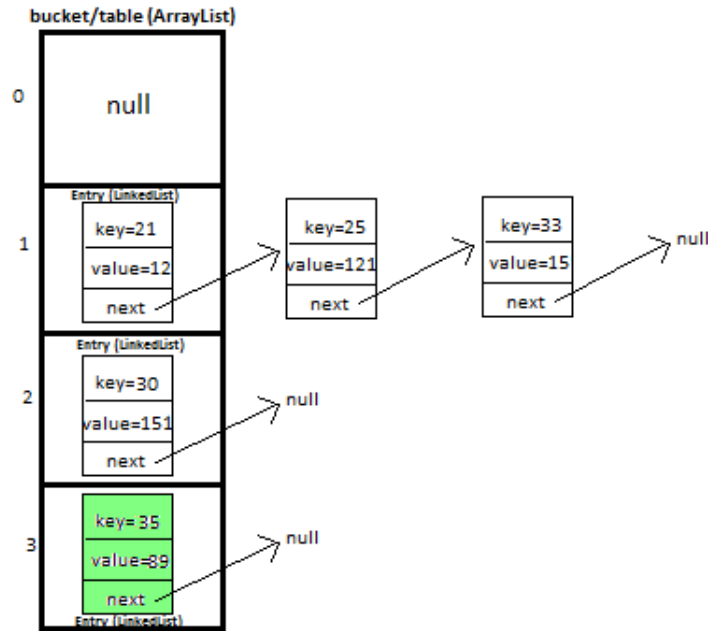


Figure 11: Hashmap

L'illustration 11 introduit les notions abordées dans cette partie. Le bucket comporte les entrées de la table de hachage. Ensuite, chaque entrée comporte une liste chaînée. La capacité de cette table de hachage est de 4. Sa longueur de 5. Expliquons cette illustration avec un exemple:

Exemple: Nous avons fixé la capacité de la banque à 4. Prenons la pair (Jean,100). Nous hashons Jean et cela donne 3. Alors, nous plaçons après la première pair dans la quatrième liste chaînée.

3.4 Makefile

Le Makefile occupe un rôle important dans le projet. En effet, utilisant de nombreuses technologies (gRPC, SGX,...), la compilation doit se faire de manière précise et centralisée. Tout d'abord, il faut voir le Makefile comme un arbre. En haut de cet arbre on retrouve les règles, qui sont premièrement les *main* que l'on exécutera à la fin. Le but pour créer un Makefile est tout d'abord de créer ces *règles*. Les *règles* peuvent aussi être des fichiers en `.o` qui sont indispensables à la compilation des *main*. Ainsi, les deux *main* que nous souhaitons exécuter à la fin étaient `bank_server_sgx` et `bank_client`. Ces

deux exécutables ont besoin du résultat de la compilation d'autres codes. Par exemple, `bank_server_sgx` nécessite le code relatif à gRPC, défini donc notamment dans le fichier `Bank.proto` et toute la partie SGX, liée à l'enclave.

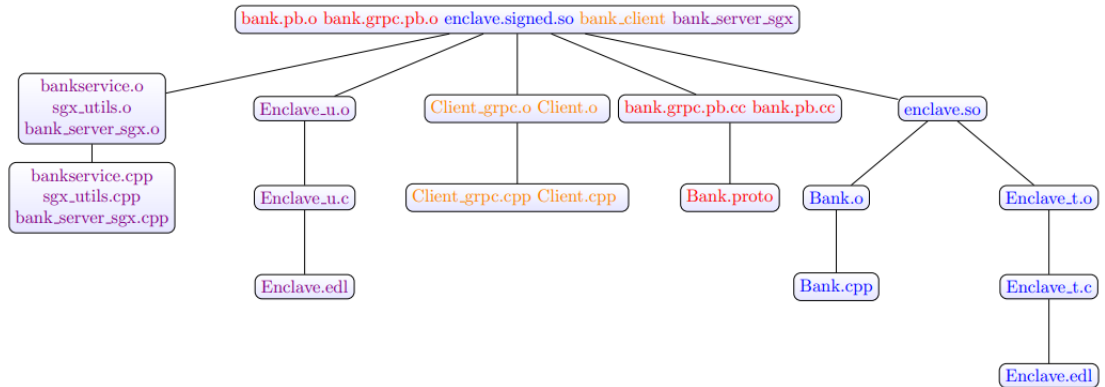


Figure 12: Makefile

La figure 12 donne un aperçu de la hiérarchie de notre makefile. Globalement, un exécutable nécessite des `.o` qui eux mêmes nécessitent des fichiers en `.cpp`, `.edl` ou `.proto` (dans notre exemple). Ce sont ces derniers que nous avons codés.

Ainsi, nous transformons les fichiers c/c++, EDL ou proto que nous avons écrits en objets. Les collections d'objets que l'on récupère servent ensuite à la création des exécutables que nous souhaitons détenir à la fin.

3.5 Injecteur de charge

L'injecteur de charge consiste à effectuer un très grand nombre d'opérations. Pour cela nous avons utilisé le fichier `tenkusernames.txt` contenant dix mille noms afin d'effectuer l'opération création de compte dix mille fois.

Ensuite en choisissant un montant et deux comptes de manière aléatoire nous avons effectué également dix mille fois les opérations débit d'un compte et crédit d'un autre compte avec un montant similaire.

Cela nous a permis de montrer que notre banque vérifie la spécification TCP-B [6] et également d'avoir un grand échantillon pour nos mesures.

3.6 Graphiques

Afin d'évaluer les performances des différentes implémentations, une représentation graphique du débit et de la latence était approprié.

Lors de l'exécution de l'injecteur de charge nous avons relevé et stocké le débit et la latence des différentes opérations dans des fichiers csv grâce à la fonction

fprintf. La lecture des fichiers csv afin de tracer les différents graphiques peut ainsi être effectuée avec un script python en utilisant la bibliothèque pandas. En effet, cela permet une lecture facile des dataframes et de tracer simplement avec matplotlib. Afin d’avoir une représentation plus parlante nous avons également fait des moyennes par tranches de temps car les valeurs obtenus pouvaient être volatiles.

4 Résultats

Pour faire nos mesures on demande en tant que client la création de dix mille compte, suivi de l'opération de débit et crédit décrit précédemment répétées également dix mille fois. Voici ce que l'on peut observer dans la console:

```
Get 10000 names from file

Create 10000 accounts from names

Make 10000 operations in the different accounts from file
 200  400  600  800 1000 1200 1400 1600 1800 2000
2200 2400 2600 2800 3000 3200 3400 3600 3800 4000
4200 4400 4600 4800 5000 5200 5400 5600 5800 6000
6200 6400 6600 6800 7000 7200 7400 7600 7800 8000
8200 8400 8600 8800 9000 9200 9400 9600 9800 10000
*****          STATS          *****
10000 accounts = 0.89 secs
10000 operations = 1.83 secs
```

Figure 13: Résultats implémentation sans SGX

```
[BankCreate]
hm capacity = 10
[bank_create] Bank created
Server listening on localhost:50051
 200  400  600  800 1000 1200 1400 1600 1800 2000
2200 2400 2600 2800 3000 3200 3400 3600 3800 4000
4200 4400 4600 4800 5000 5200 5400 5600 5800 6000
6200 6400 6600 6800 7000 7200 7400 7600 7800 8000
8200 8400 8600 8800 9000 9200 9400 9600 9800 10000
```

Figure 14: Résultats implémentation sans SGX

Ensuite on execute le fichier graph.py pour obtenir les courbes représentant les données récoltées dans les fichiers csv au cours des opérations précédentes. Voici les résultats après exécution des benchmarks:

4.1 Implémentation gRPC sans SGX

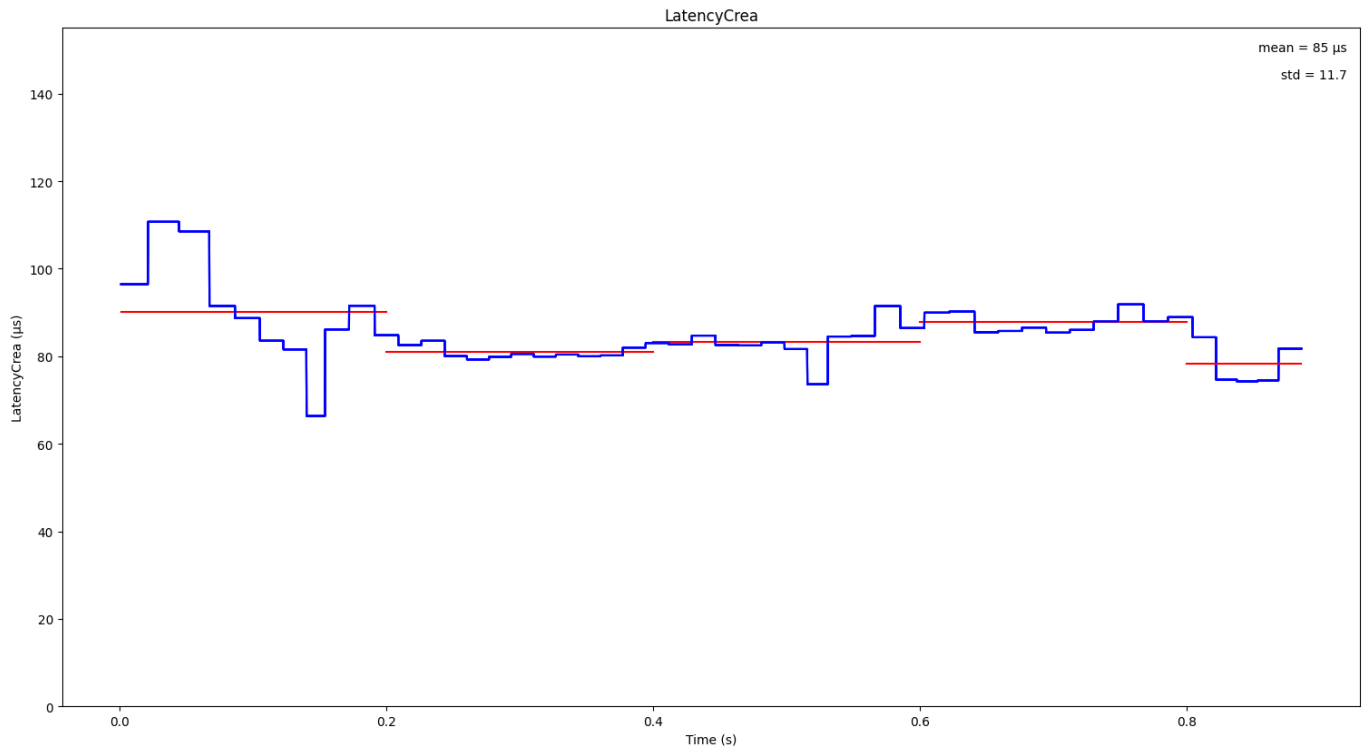


Figure 15: Latence pour la création de compte gRPC seul

De la même manière on obtient le débit pour la création ainsi que la latence et le débit pour les opérations de débit et crédit.

4.2 Implémentation gRPC avec SGX

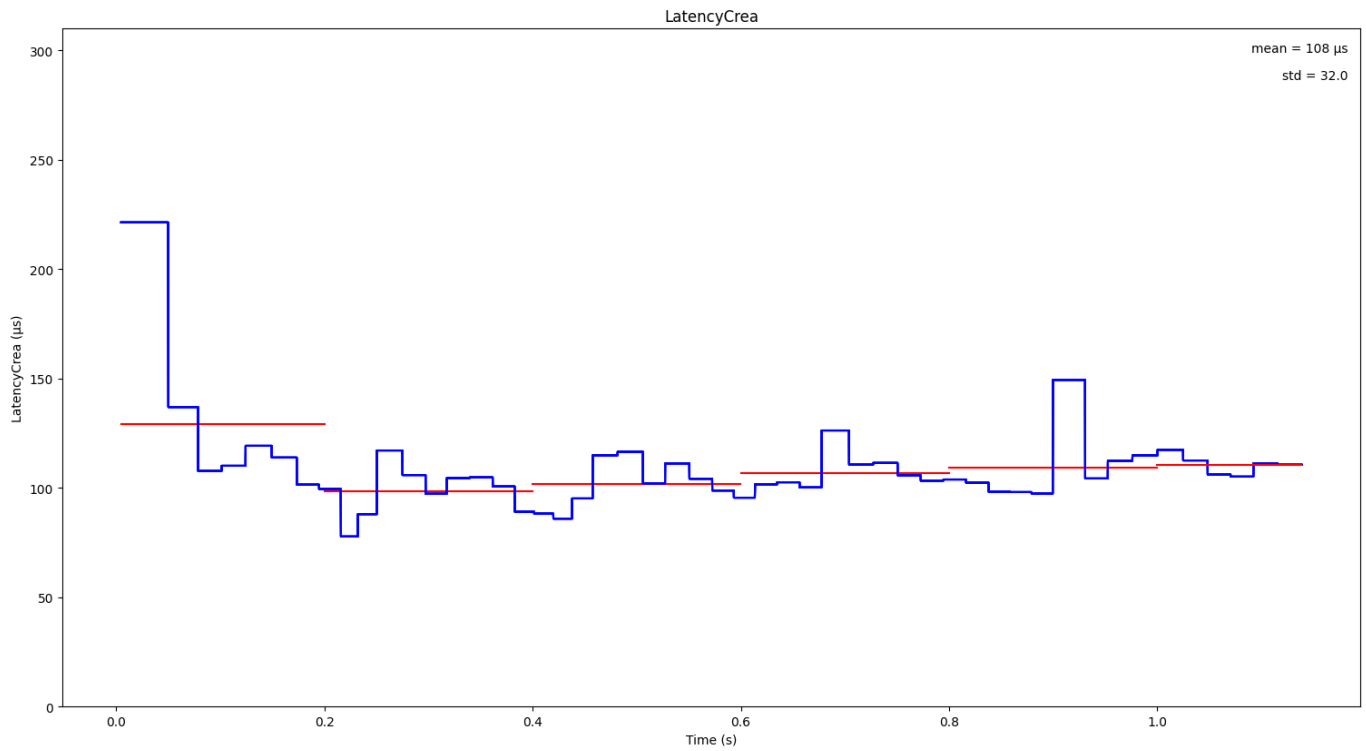


Figure 16: Latence pour la création de compte avec SGX

4.3 Analyse des résultats

	Création de comptes		Débit et Crédit	
	Latence (μ s)	Débit (ope/ms)	Latence (μ s)	Débit (ope/ms)
gRPC	85	11	180	6
gRPC + SGX	108	9	283	4

Figure 17: Résultats comparés

Comme attendu nous observons que l'emploi de l'enclave SGX rend les opérations légèrement plus lente, cependant la différence de temps reste faible car c'est le réseau qui domine.

5 Conclusion

5.1 Objectifs atteints

Nous avons donc réussi à

- Créer une banque distribuée et sécurisée dans une enclave
- Compiler cette banque et la faire fonctionner
- Effectuer le benchmark sur les différentes versions de cette banque

5.2 Suite du projet

- Propriétés ACID

En informatique, notamment dans le domaine bancaire, les propriétés ACID (A signifie atomicité, C cohérence, I Isolation et D Durabilité) assurent un panel de garanties censé rendre la transaction fiable.

L'ajout de différents *locks* nous aurait permis de garantir A,C et I. Quant à la durabilité, notre système bancaire perd ses données dès que le système s'éteint. Les bases de données pérennes en relation avec des enclaves sont un sujet encore très porteur en informatique et qui n'a pas fourni de solutions universelle (problème du replay).

- Implémentation "colorée" pour compilateur Privagic

Ajouter la coloration dans les fichiers et compiler en utilisant la solution Privagic afin d'obtenir une version gRPC + SGX.

- Mesurer le coût en terme de nombre de ligne de code et les performances
- Observer le bon fonctionnement du compilateur et la différence par rapport à l'implémentation "faite main" gRPC + SGX

- Surcouche d'authentification

Implémenter avec libssl une surcouche d'authentification des requêtes clients avec des enclaves SGX. L'idée est de faire du client-to-enclave encryption afin d'éviter qu'un serveur compromis puisse faire du MITM avec les enclaves.

Idée de la fonctionnalité:

- client::request::account::name@blue → authenticate with → server::enclave::@green
- client::request::account::amount@red → authenticate with → server::enclave::@orange

6 Bibliographie

References

- [1] <https://www.lesassisesdelacybersecurite.com/Le-blog/Glossaire/Hash>
- [2] <http://www.cse.yorku.ca/~oz/hash.html>
- [3] EDL Input and Bound Check - Intel <https://cdrdv2-public.intel.com>
- [4] Intel(R) Software Guard Extensions SDK Developer Reference for Linux* OS
- [5] <https://github.com/digawp/hello-enclave>
- [6] <https://www.tpc.org/tpcb/>
- [7] <https://github.com/attractivechaos/klib/blob/master/khash.h> ...

Annexe

Implémentation gRPC sans SGX

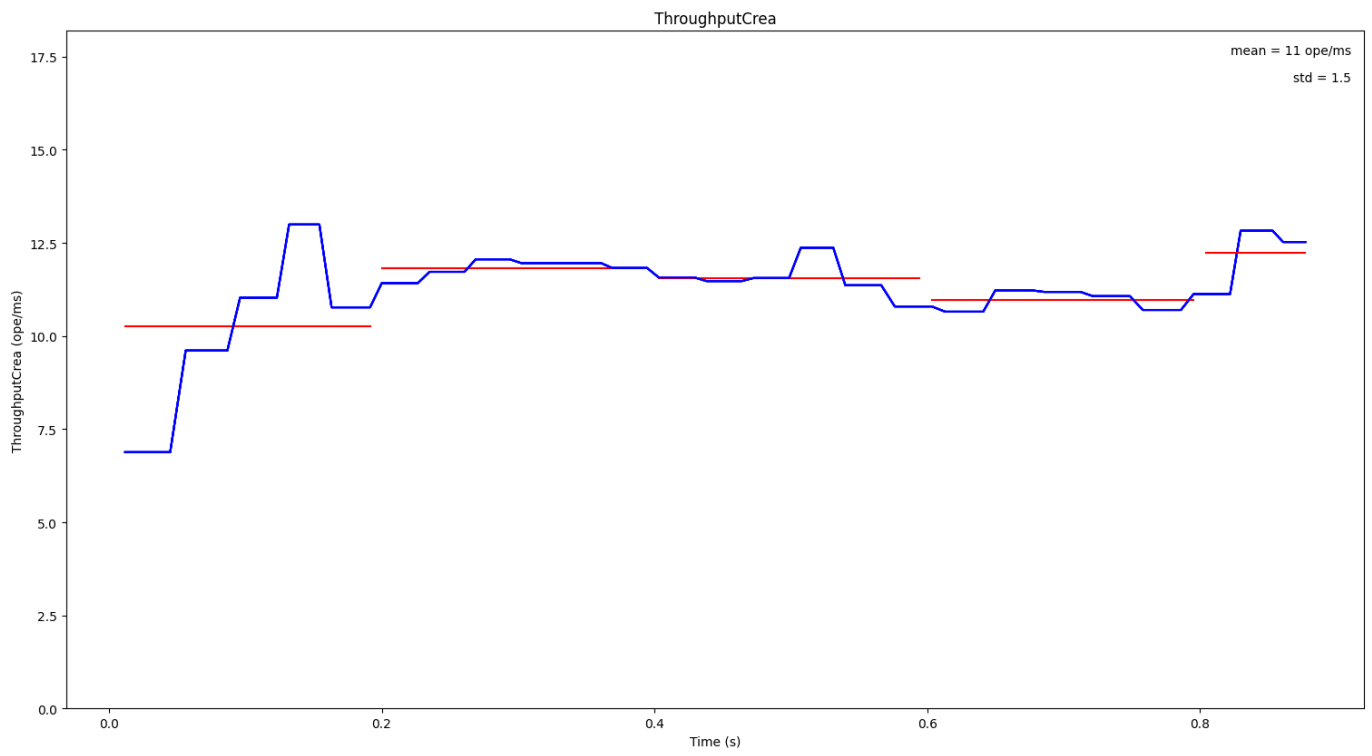


Figure 18: Débit pour la création de compte gRPC seul

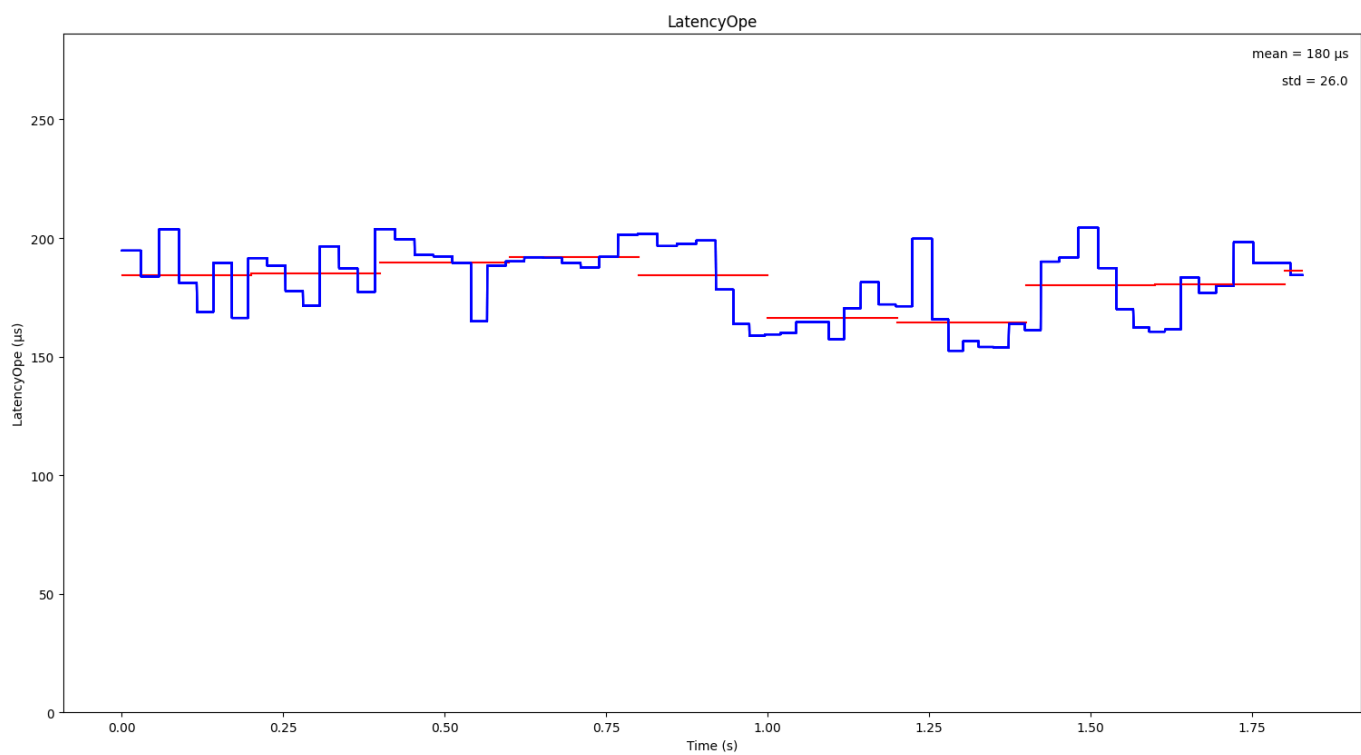


Figure 19: Latence pour l'opération de débit et crédit gRPC seul

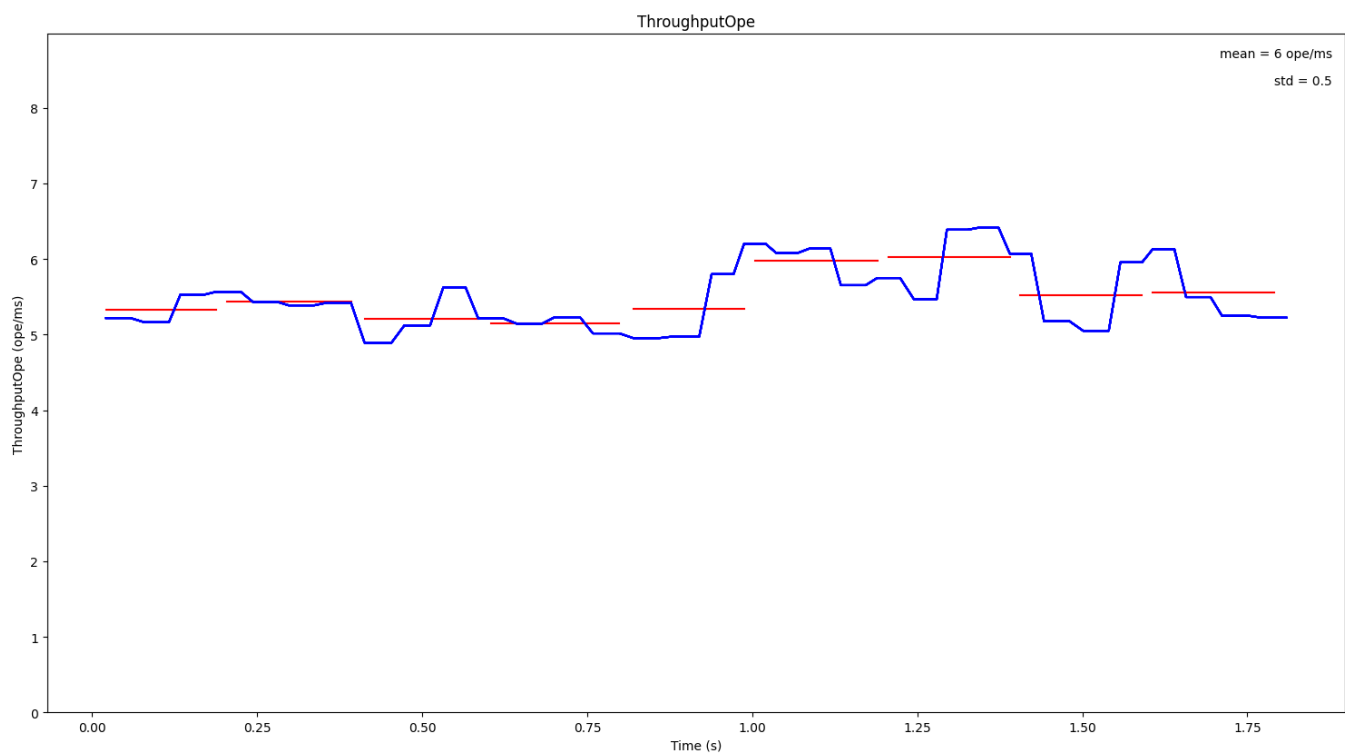


Figure 20: Débit pour l'opération de débit et crédit gRPC seul

Implémentation gRPC avec SGX

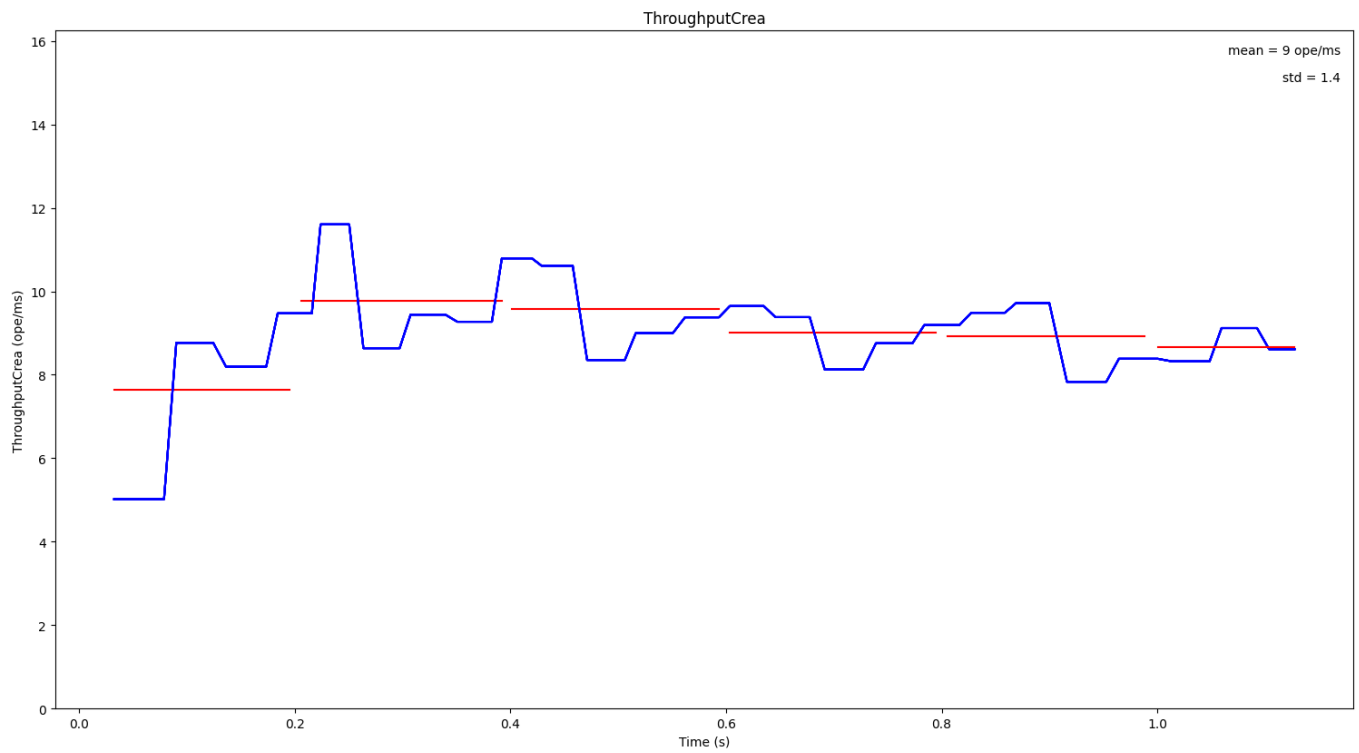


Figure 21: Débit pour la création de compte avec SGX

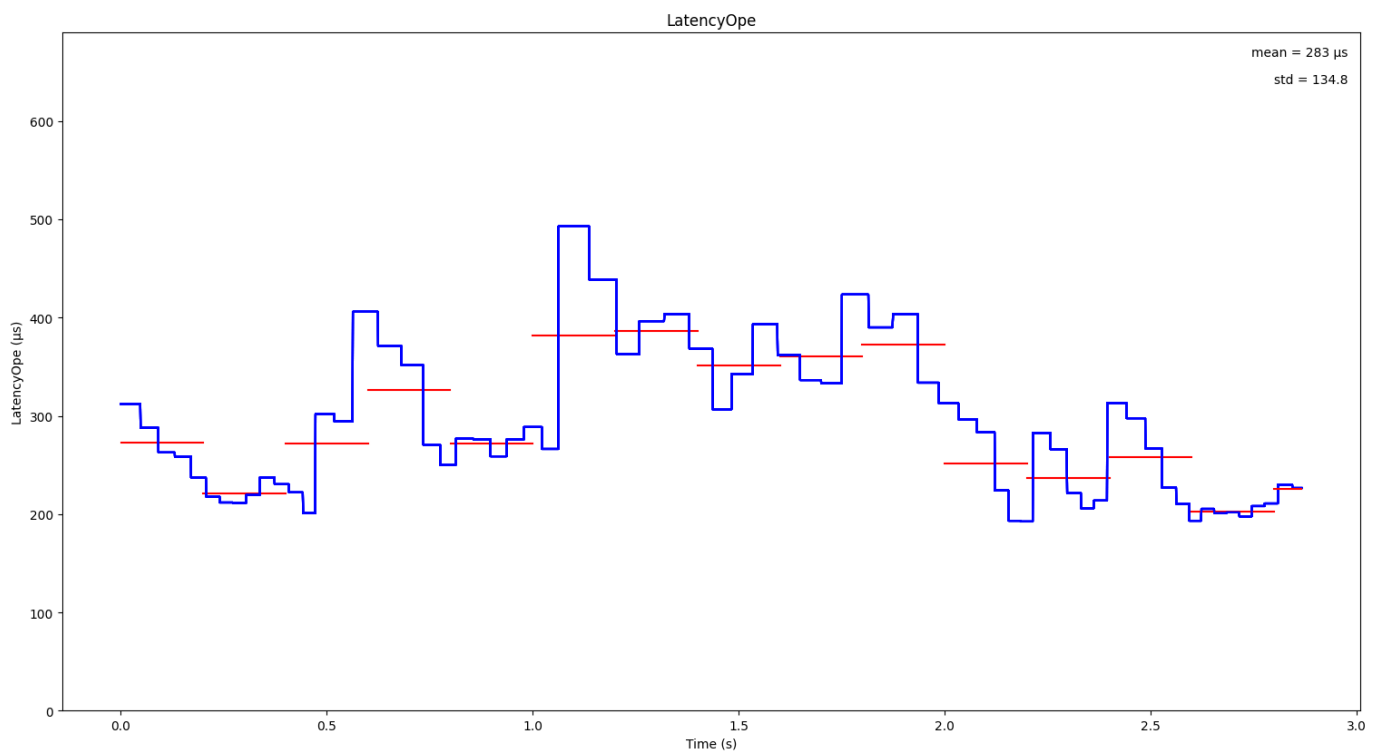


Figure 22: Latence pour l'opération de débit et crédit avec SGX

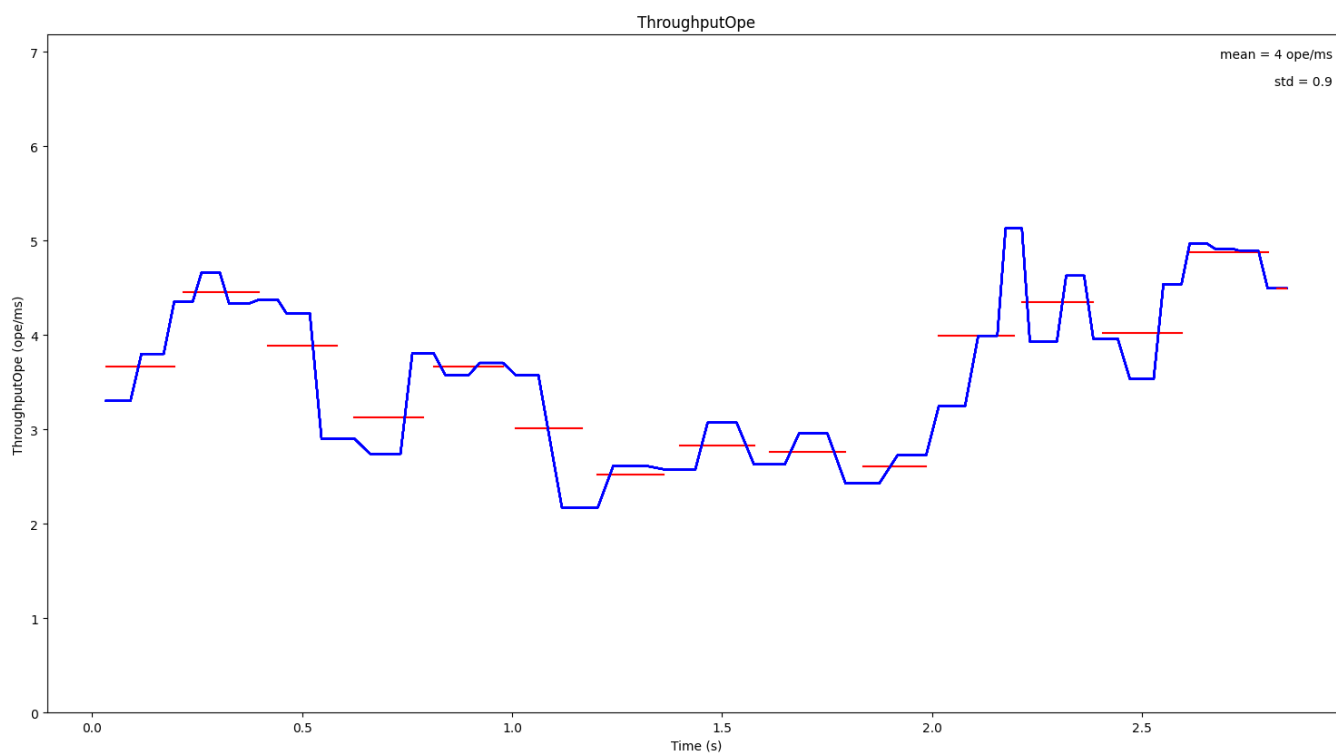


Figure 23: Débit pour l'opération de débit et crédit avec SGX