# SR2I309 - Sécurité des donneés dans le cloud

*Projet - Compte rendu*
*Calcul sur données chiffrées dans un cloud*

https://github.com/NathSimon/fhe-sorting

Nathanaël Simon
Hamza Zarfaoui
nathanael.simon@telecom-paris.fr
hamza.zarfaoui@telecom-paris.fr

Télécom Paris, Institut Polytechnique de Paris

November 7, 2023

# 1 Présentation

## 1.1 Contexte

La technologie cloud connaît un développement important ces dernières années. En passant de 1.4M à 2.5M d'euro de marché entre 2020 et 2022 [1], le nombre d'entreprise choississant de faire confiance à un Cloud Service Provider (CSP) tel qu'AWS ou Azur explose. Bien que cette technologie permette aux clients des CSP de réduire les coûts liés à leur infrastructure, garanrtir une disponnibilité, une résilence et une scalabilité importante de leurs services, celle-ci introduit également de nombreuses problématiques.

Parmis celles-ci, il existe une partie de risques déjà présents dans des infrastructures classiques, qui doivent donc être adréssés de manière propre à l'architecture cloud, mais également des risques spécifiques, tel que la confiance et la confidentialité des données. En effet, le client souhaitant utiliser les services proposés par le CSP doit lui fournir ses données, lesquelles peuvent être sensibles, confidentielles, ou encore nominatives. Aisni, ce dernier fait confiance au CSP sur son architecture et ses mesures de sécurité pour éviter une fuite de ses données, mais également au CSP de ne pas accéder et utiliser ses données. Afin de répondre à ces problématiques, le chiffrement homomorphe, permettant le calul sur données chiffrées, propose une solution permettant de supprimer le besoin de confiance envers le CSP, tout en se prottégeant d'une fuite de données.

Dans le cadre de l'UE SR2I309, nous avons choisi d'explorer cette solution et de la mettre en place sur un exemple de tri de données, afin d'en comprendre les avantages et inconvénients, afin de mieux se rendre compte des cas d'usages auxquels le chiffrement homomorphe peut répondre.

Il sera question dans ce document de rendre compte des méthodes utilisées et des résultats obtenus, lors de la réalisation de ce projet, en se questionnant donc sur la mise en place à grande échelle de cette méthode de chiffrement.

## 1.2 But du projet

Le but de ce projet est de réaliser un tri sur données chiffrées dans un contexte de cloud. Pour ce faire, nous mettrons en place un client et un serveur. Le serveur possèdera l'algorithme de tri et recevra du client un fichier contenant les données chiffrées. Ce dernier renverra un fichier contenant les données chiffrées triées. Dans un dernier temps, le client pourra les déchiffrer et vérifier et le résultat.

# 2 Calcul sur données chiffrées

## 2.1 Fully Homomorphic Encryption

Afin de pouvoir réaliser un calcul sur données chiffrées, il est nécessaire d'utiliser une technique de chiffrement FHE (Fully Homomorphic Encryption). Un chiffrement est qualifié de FHE si les 3 propriétés suivantes sont respectées :

$$Enc(M_1) + Enc(M_2) = Enc(M_1 + M_2) \quad (1)$$

$$Enc(M_1) * Enc(M_2) = Enc(M_1 * M_2) \quad (2)$$

$$Not(Enc(M_1)) = Enc(Not(M_1)) \quad (3)$$

Garantir ces 3 opérations permet d'utiliser les opérateurs de l'addition, de la multiplication et de la négation sur des données chiffrées, tout en pouvant retrouver le résultat sur les données claires lors du déchiffrement.

Il est ainsi possible, depuis un système de chiffrement FHE, de traduire tous les circuits logiques vers un équivalent FHE permettant de réaliser le calcul souhaité. Les données chiffrées auront cependant besoin d'un taux d'expension très important pour pouvoir garantir ces propriétées. De plus, le chiffrement homomorphe se reposant sur du bruit, chaque opération agrandit le bruit de la donnée chiffrée. Il est donc nécessaire de réaliser des opérations de bootstraping pour reduire ce bruit et poursuivre les calculs. Une probabilité d'erreur apparaît donc, environ de 1/100000 pour les bibliothèques disponnibles.

## 2.2 Circuit FHE

Afin de pouvoir éffectuer des opérations sur des données chiffrées homomorphiquement, il est donc nécessaire de traduire un circuit classique vers un circuit FHE. Pour se faire, la bibliothèque Zama [2] propose ainsi un compilateur, Concrete, permettant de compiler du code classique vers un circuit

FHE. Cependant, certaines opérations ne sont pas réalisables sur des données chiffrées nativement. [3]

Par exemple, une comparaison entre deux valeurs chiffrées est une opération très couteuse (nous verrons par la suite comment Zama propose de le réaliser), et entre une valeur claire et une valeur chiffrée est impossible. De plus, le circuit doit être purement déterministe et linéaire sur ses valeurs chiffrées, et donc ne pas posséder d'embranchement à partir de ces dernières. Nous allons prendre un exemple simple pour illustrer, le tri à bulle.

```
def bubble_sort(array):
    n = len(array)
    for i in range(n):
        already_sorted = True
        for j in range(n - i - 1):
            if array[j] > array[j + 1]:
             #Comparaison sur donnees
    chiffrees
                array[j], array[j + 1] =
    array[j + 1], array[j]
                already_sorted = False
        if already_sorted:
            break
    return array
```

Listing 1: Bubble sort python implentation

Nous voyons ici que le tri à bulle réalise une comparaison sur deux éléments du tableau, qui seront chiffrées, pour savoir si il est nécessaire de les inverser. Il est possible d'adapter cette opération en utilsant une variable de comparaison.

```
cmp = array[j] > array[j + 1]
tmp = array[j]
array[j] = array[j + 1] * cmp + array[j] * (1
     - cmp)
array[j + 1] = array[j + 1] * (1 - cmp) + tmp
     * cmp
```

Listing 2: Comparison trick

Ici cmp est une variable chiffré, qui possède le résultat de la comparaison entre nos deux valeurs. Par la suite, nous allons utiliser une opération de multiplication, autorisée sur des données chiffrées, pour échanger les deux valeurs de notre tableau en fonction du résultat de la comparaison. L'implémentation FHE est donc la suivante :

```
@fhe.compiler({"array": "encrypted"})
def function(array):
    n = len(array)
    for i in range(n):
        already_sorted = True
        for j in range(n - i - 1):
            cmp = array[j] > array[j + 1]
            tmp = array[j]
            array[j] = array[j + 1] * cmp +
    array[j] * (1 - cmp)
            array[j + 1] = array[j + 1] * (1
    - cmp) + tmp * cmp
            already_sorted = False
        if already_sorted:
            break
    return array
```

Listing 3: Bubble sort FHE implentation

Nous pouvons remarquer ici que nous perdons grandement en efficacité réelle. Avant l'opération d'échange n'était réalisée que dans le cas où la comparaison était vérifiée, elle est maintenant effectuée à chaque fois.

Cependant, il n'est pas toujours possible de réaliser une adaptation simple des algorithmes classiques. Par exemple, pour l'algoithme quicksort, une opération impliquant une valeur chiffrée et clair en même temps doit être adaptée.

```
for j in range(low, high):
    if arr[j] <= pivot:
        i += 1
        arr[i], arr[j] = arr[j], arr[i]
```

Listing 4: Extrait de quicksort

Ici, i est une variable en claire, un indice permettant d'accéder au tableau chiffrée. Cependant son incrémentation dépends du résultat d'une comparaison sur données chiffrées. Essayons d'utiliser la méthode précédente pour adapter ce code.

```
for j in range(0, len(array)):
    cmp = array[j] <= array[len(array) - 1]
    #i = (i+1) * cmp + i * (1 - cmp)
    tmp = array[i]
    array[i] = array[j] * cmp + array[i] * (1
     - cmp)
    array[j] = array[j] * (1 - cmp) + tmp *
    cmp
```

Listing 5: Adaptation fausse de quicksort

Ici nous tentons donc d'ajuster la valeur de i, mais en dependant de cmp, chiffré, ce qui n'est pas possible. Il est alors necessaire de profondément revoir l'agorithme de base pour pouvoir l'adapter, mais en perdant également grandement en efficacité. Une autre méthode serait d'utiliser une valeur chiffrée pour i, mais l'accès à un indice par une valeur chiffrée est une opération extrêmement couteuse, qui rendrait pratiquement l'implémentation inutilisable pour n'importe quel contexte.

### 2.3 Stratégie de comparaison

Toutes les opérations non natives au FHE sont converties en Table Lookups (TLU) pour pouvoir être réalisées. La complexité exacte de celles ci dépend du hardware, de la probabilité d'erreur, ect ... mais ce sont des opérations beaucoup plus couteuse que celles de bases. Ce sont ces TLUs qui permettent notament de réaliser les comparaisons entre données chiffrées. Il faut donc choisir des algorithmes permettant de minimiser les comparaisons, et limiter le coût propre d'une comparaison. [4]

Afin de réaliser ces opérations de comparaisons couteuses, Zama permet de choisir entre plusieurs stratégies prédéfinies. Chacune de ces stratégies propose un compromis entre flexibilité, complexité et taille de chiffrement de la donnée nécessaire. Pour ce projet, nous en avons retenu et comparée 3, qui expriment pour nous 3 compromis intéressant dans notre cas d'usage. Nous ne rentrerons pas ici dans le détail de l'implentation de chacune d'entre elle, mais nous nous concentrerons sur les avantages et inconvénients.

1. Chunked
   Technique de comparaison par défaut. Propose la plus grande flexibilité en permettant de travailler sur n'importe quel entiers, mais est également la plus coûteuse. Une comparaison effectuera entre 5 et 13 TLUs.
2. One TLU Pormoted
   Permet de garantir une seule TLU par comparaison. Cependant, celle ci augmente la taille néssaire des données chiffrées, et peut ralentir d'autres opérations effectuées sur ces mêmes données.
3. Three TLU Casted
   Garanti entre 1 et 3 TLUs par comparaison, et ne nécessite pas d'augmentation de la taille des données chiffrées, ce qui permet de ne pas compromettre d'autres calculs.

Dans notre cas d'usage, où seulement une seule opération de tri est réalisée, nous ne possédons pas de contraite forte sur la taille des données chiffrées, tant que le volume de stockage nécessaire est conservée. Ceci devrait nous garantir le meilleur temps d'exécution. Cependant, les deux autres présente des points forts intéressant, et chaque usage nécessite un choix éclairé pour optimiser au mieux les calculs.

### 2.4 Réseaux de tri

Les réseaux de tri (sorting netwoks) permettent de réaliser un tri en suivant une suite d'opération prédéfinies, indépendement des données d'entrées. Ceux-ci ne se reposent pas sur les valeurs des données pour réaliser leurs comparaisons, mais sur leur suite prédeterminée de comparaisons. Ils ne fonctionnent donc que sur une entrée de taille prédeterminée, et ne sont généralement pas les plus versatiles ni les plus efficaces dans un contexte sur des données claires.

Cependant, dans un contexte de FHE, tous les algorithmes de tris classiques sont enfaite traduit par le compilateur zama en sorting networks, mais de manière bien moins efficiente qu'un sorting network natif. En particulier, la convertion aboutit a des réseaux ayant une profondeur très importante, ceux-ci ne pouvant parralléliser beaucoup d'opérations. Ces algorithmes sont donc d'excellents candidats pour un tri efficace sur données chiffrées, mais imposent une grande contrainte sur la flexibilité des algorithmes. Dans des cas d'usages où la taille des entrées est variable devront donc s'adapter, soit sur leurs données soit sur les algorithmes utilisés. Dans ce projet, nous utiliserons l'algorithme topk sorting network. [5] [6] [7]

## 3 Mise en place dans un contexte cloud

Nous souhaitons désormais déployer nos circuits dans un contexte cloud. Pour se faire nous allons mettre en relation un client et un serveur, sur un machine virtuelle sur le réseau de l'école. Les circuits sont précompilés, intégré au serveur, qui pourra les charger à la demande du client. Le serveur sera embarqué dans un containeur, qui exposera un port afin de pouvoir être accessible par le client.

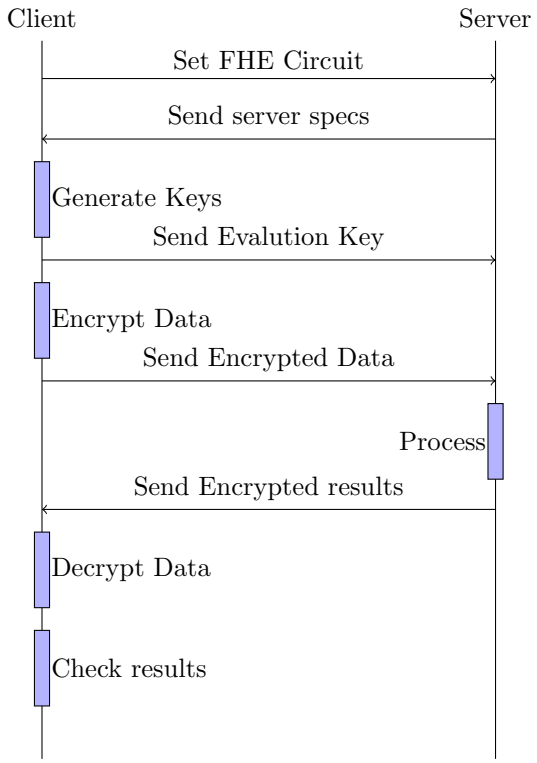


Figure 1: Process entre le client et le server

Dans un premier temps, le client spécifie au serveur le circuit qu'il souhaite utilser, parmis ceux disponnibles. Ce circuit va définir, de par la taille des données qu'il attends ou par sa stratégie de compairaison, quels sont les paramètres nécessaires pour le chiffrement des données, appelées server specifications.

Une fois ces données recus, le client peut donc générer les clés de chiffrement/déchiffrement et d'évaluation, et envoyer celle d'évaluation au serveur, lui permettant de réaliser les opérations de comparaisons. Une fois les données chiffrées, ce dernier les enregistre dans un fichier binaire, et l'upload sur le serveur. Celui ci peut enfin réaliser le tri et enregistrer les résultats dans un fichier. Enfin, le client télécharge ce fichier, le déchiffre et peut vérifier les résultats. Nous avons pris la décision de faire transiter des fichiers entre le serveur et le client afin de pouvoir plus aisément comparer la taille des données chiffrées, mais il est possible de les envoyer comme paramètres de la requête, même si ceux-ci sont très volumineux.

Les clés de chiffrement/déchiffrement n'étant jamais envoyées, nous garantissons ainsi la confidentialité des données côté serveur, et donc la réponse à cette problématique dans le contexte cloud. [8]

# 4 Résultats

## 4.1 Circuits implémentés

Lors de ce projet, nous avons implémenté et comparé les circuits suivants [9] chacun utilisant les 3 stratégies de comparaison précédente:

1. Bubble sort - $\mathcal{O}(n^2)$
2. Insertion sort - $\mathcal{O}(n^2)$
3. Topk sort - $\mathcal{O}(n \log^2 n)$

Nous pouvons comparer leurs taille compilée, le temps de compilation, et la taille des données chiffrées et leur temps d'exécution.

Conditions de test :

- Tri sur 20 valeurs : toutes les valeurs sont comprises entre 0 et $2^4$.
- Les tests d'exécutions ont été menés 2 fois. Une fois sur les VM misent à disposition, une fois sur une machine personnelle (MP) composée d'un CPU : 13th Gen Intel® Core™ i9-13900H × 20 et 32GiB de RAM. Ils sont le résultat d'une éxécution arbitraire et non d'une moyenne, le but étant de donner des ordres de grandeurs.
- Les résultats de compilations ont été obtenus sur MP.
- Le facteur d'exécution est calculée par rapport aux valeurs de MP.

| 20 Values $\in [0, 2^4]$ | Chunked | OTLU | TTLU |
|---|---|---|---|
| Compilation | 303s | 216s | 213s |
| Taille du circuit | | 3610 | |
| Taille du fichier claire/chiffré | 55B/655kB (×11920) | | |
| Execution Chiffrée VM/MP | 193s/75s | 196s/85s | 189s/86s |
| Execution Claire | 0.042s | 0.043s | 0,039 |
| Facteur d'exécution | ×1785 | ×1976 | ×2205 |

Table 1: Résultats pour le tri à bulle sur 20 valeurs

| 20 Values $\in [0, 2^4]$ | Chunked | OTLU | TTLU |
|---|---|---|---|
| Compilation | 240s | 261s | 273s |
| Taille du circuit | | 3249 | |
| Taille du fichier claire/chiffré | 55B/655kB (×11920) | | |
| Execution Chiffrée VM/MP | 200s/85s | 197s/90s | 198s/87s |
| Execution Claire | 0.037s | 0.037 | 0.037s |
| Facteur d'exécution | ×2297 | ×2432 | ×2350 |

Table 2: Résultats pour le tri par insertion sur 20 valeurs

| 20 Values $\in [0, 2^4]$ | Chunked | OTLU | TTLU |
|---|---|---|---|
| Compilation | 76s | 68s | 69s |
| Taille du circuit | | 702 | |
| Taille du fichier claire/chiffré | 55B/328kB (×5961) | | |
| Execution Chiffrée VM/MP | 21.9s/2.8s | 20.3s/2.7s | 21.4s/2.76s |
| Execution Claire | 0.014s | 0.014 | 0.016s |
| Facteur d'exécution | ×200 | ×192 | ×172 |

Table 3: Résultats pour le réseau de tri topK 20 valeurs

## 4.2 Analyse

La première observation que nous pouvons réaliser sur ces résultats sont les facteurs d'exécution et de chiffrement.

- Facteur de chiffrement - 5000 à 1000
- Facteur d'execution - 200 à 2000

Le facteur de chiffrement est particulièrement important du au nombre de comparaisons réalisées dans les algorithmes de tris. On voit que dans le cas de topk, où le nombre de comparaisons est plus faible, le facteur d'expansion l'est également.

Le facteur d'éxécution est également grandement réduit dans le cadre de topk. Sa compléxité inférieure et la capacité naturelle du réseaux de tri à paralléliser ses opérations permet une éxécution 10x plus rapide que pour les deux autres algorithmes. On remarque également que ces deux ont des performances très similaires, dû à leur compléxité classique proche et la conversion semblable en circuit FHE.

Cependant, à notre surprise, les différentes stratégies de comparaisons n'ont pas eu d'impact sur la taille des circuits ni sur leur temps d'execution. Ceci peut être du à la taille du tableau ou à la plage de valeur sur lesquelles nous avons travaillé, qui pourraient être trop faibles pour que ces stratégies puissent montrer des différences à l'exécution. Il faudrait ici réaliser plus de tests sur des machines plus puissantes pour pouvoir conclure.

## 4.3 Scalabilité

Au cours de nos tests, nous avons souhaité traiter des tableaux plus importants sur des plages de valeurs plus élevées. On se heurt à 3 problèmes de scalabilité importants, en fonction du paramètre que l'on souhaite augmenter.

- Algorithme choisi : temps d'exection
- Taille du tableau à trier : Temps de compilation/execution
- Plage de donnée : temps de chiffrement/déchiffrement et expension de la donnée

L'augmentation de la taille du tableau à trier entraîne une baisse très importante de performance lors de la compilation du circuit. Par exemple, la compilation d'un circuit bubble sort sur 100 valeurs produit un graph de 94250 entrées, et n'a pas pu être compilé en plus de 12h. Bien qu'il ne suffise que d'une compilation, ceci est un frein important à la scalabiltié des algorithmes FHE, sans prendre ne compte le temps d'execution par la suite. Le deuxième paramètre que nous avons souhaité augmenter est la plage de valeur possible des données d'entrées.

Par exemple, pour le circuit topk sur 20 valeurs $\in [0, 2^8]$, le temps de compilation reste identique, car celui ci ne depends de la taille du tableau d'entrée et non de sa plage de valeur, par contre le temps nécessaire au chiffrement et déchiffrement des données est bien plus important. Plus de 2 heures n'ont pas suffit à chiffrer le tableau donné en entrée.

Ceci pourrait représenter une limitation bien plus importante au déploiment des circuits FHE que le temps de compilation ou d'execution. Dans une utilisation réelle, les données seront sur une plage de donnée bien plus importante que $[0, 2^8]$, et l'opération de chiffrement/déchiffrement est une opération répétée à chaque lancement d'un calcul, contrairement à la compilation du circuit. Il est donc

nécessaire de mutualiser les calculs au maximums pour limiter ces opérations coûteuses.

Cependant, il est nécessaire de prendre en compte que de manière générale, les opérations sont lentes et demandeuses en performance des machines, aussi bien sur la RAM que sur le CPU. Des serveurs dédié à ces calculs doivent être mis en place, et ceci, surtout dans une architecture cloud ou la tarification est à l'usage, peut représenter rapidement un coût très important en comparaison avec les algorithmes classiques.

# 5 Conclusion

Lors de ce projet nous avons donc pu mettre en place, dans une architecture cloud, des techniques de calculs sur données chiffrées afin d'aboutir à un tri de données. Cette technologie possède des points forts indénibales, garantissant une confidentialité très importante sur les données, et peut permettre de travailler avec des ressources externes sur des données sensibles ou personnelles. La mise en place de cette technologie est grandement facilitée par l'utilisation de bibliothèque dédiées et particulièrement bien documentée dans le cas de Zama, ce qui n'est pas toujours le cas dans les bibliothèques cryptographiques.

Cependant cette technologie connaît encore de nombreux défis avant de pouvoir espérer être déployée à grande échelle. En particulier, les problématiques de scalabilité des algorithmes sont importantes et sont dans certains cas les limites fondamentales de la technologie. Ces limitation entraînent des calculs long et coûteux et sont, dans ces échelles de grandeurs, trop imporante pour espérer se démocratiser. Cependant, la recherche scientifique est particulièrement active dans le domaine [10] [11], avec des propositions régulières d'algorithmes spécialisés et optimisés pour ce cas d'usage [12]. Cependant les calculs apporteront des contraintes dans l'utilisation des algorithmes, tel que le besoin d'avoir des entrées de taille fixe, qui devront être adréssé au cas par cas.

Cette technique de chiffrement est donc primordiale pour les cas d'usages où la confidentialité est la contrainte la plus importante. Cela pourrait permettre la collaboration et le calculs sur des données qui n'étaient auparavant pas partagées par des acteurs tel que la santé, la défense ou encore des états, dans l'entraînement de modèles d'intelligence artificielle par exemple.

Cependant, dans des cas d'usages ou la capacité de calcul, la rapidité de calcul ou bien la taille des données d'entrées sont des impératifs, le chiffrement FHE ne semble pas être une solution plus intéressante que les standard du NIST déjà en place, et il sera nécessaire de faire des compromis sur la confidentialité des données.

# 6 Annexes

En plus de ce document, les annexes suivantes sont fournies :

- Le code source des circuit FHE, et leur compilation
- Le code du serveur et les ressources pour le contenairisé
- Le code client et sa GUI

Les ressources sont trouvables au lien suivant : `https://github.com/NathSimon/fhe-sorting`

# References


[1] "Distributique, cloud." `https://www.distributique.com/actualites/lire-en-10-ans-les-depenses-dans-le-cloud-ont-explose.html`.

[2] "Zama, fully homomorphic encryption." `https://www.zama.ai/`.

[3] "Zama, concrete compiler." `https://docs.zama.ai/concrete`.

[4] "Zama, comparisons strategies." `https://docs.zama.ai/concrete/tutorials/comparisons`.

[5] "Introduction to algorithms, thomas h. cormen, charles e. leiserson, and ronald l. rivest." `http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap28.htm`.

[6] "Sorting Netwoks, thomas sauerwald, cambridge university." `https://www.cl.cam.ac.uk/teaching/1415/AdvAlgo/lec1_ann.pdf`.

[7] "List of sorting networks, bert dobbelaere." https://bertdobbelaere.github.io/sorting_networks.htmlc.

[8] "Zama, deploy application." https://docs.zama.ai/concrete/how-to/deploy.

[9] "Sorting Algorithms, wikipedia." https://en.wikipedia.org/wiki/Sorting_algorithm.

[10] H. Narumanchi, D. Goyal, N. Emmadi, and P. Gauravaram, "Performance analysis of sorting of fhe data: Integer-wise comparison vs bit-wise comparison," in *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pp. 902–908, 2017.

[11] A. Chatterjee and I. Sengupta, "Searching and sorting of fully homomorphic encrypted data on cloud." Cryptology ePrint Archive, Paper 2015/981, 2015. https://eprint.iacr.org/2015/981.

[12] S. Hong, S. Kim, J. Choi, Y. Lee, and J. H. Cheon, "Efficient sorting of homomorphic encrypted data with k-way sorting network," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 4389–4404, 2021.