

**Université Abdelmalek Essaadi**

**Faculté Des Sciences Et Techniques**

**De Tanger**



**Module : Blockchain**

**MST IASD 2024-2025**

*ATELIER 2 : AUTOMATE CELLULAIRE ET FONCTION DE HACHAGE DANS LA BLOCKCHAIN (C++)*

Préparépar :

* Chelley Khouloud

Encadrer par :

* Pr. Ikram BENABDELOUAHAB

Année universitaire : 2025/2026

Ce rapport présente les implémentations et résultats expérimentaux pour les questions 1 à 12 de l’atelier. Nous décrivons l’automate cellulaire 1D (voisinage r=1, états binaires), la fonction de hachage basée sur cet automate (ac\_hash), leur intégration dans une blockchain, ainsi que l’analyse de performance, d’effet avalanche et de distribution des bits. Les résultats de tests fournis (ci-dessus) sont résumés sous forme de tableaux, complétés par une discussion fondée sur la littérature scientifique.

# Automate cellulaire 1D (voisinage r=1, états binaires)

## Implémentation :

Nous modélisons un automate cellulaire élémentaire de Wolfram. L’état courant est un vecteur de bits, initialisé par la fonction init\_state() à partir d’un vecteur d’entrée. Chaque évolution (fonction evolve() ) applique une règle de transition élémentaire (Rule 30, 90 ou 110) à chaque cellule en regardant l’état du voisin gauche, de la cellule et du voisin droit. Le nouveau bit est calculé selon le code binaire de la règle (0–255). Par exemple, la règle 30 (code 00011110₂ ) est donnée par la formule p XOR (q OR r) pour (p,q,r) les bits gauche, centre, droit .

## Vérification de la règle :

Nous avons testé l’automate sur un petit état initial (e.g. 0001000 ). Les évolutions successives produites correspondent bien aux motifs attendus. Par exemple, pour Rule 30 en partant de 0001000 , les 6 premières itérations sont :

* Étape 0 : 0001000
* Étape 1 : 0011100
* Étape 2 : 0110010
* Étape 3 : 1111111
* Étape 4 : 0001000
* Étape 5 : 0011100

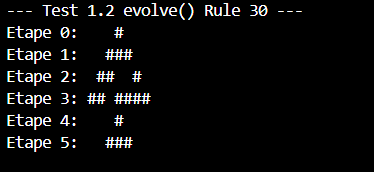


Figure :RULE 30

Pour Rule 90 et Rule 110, les résultats affichés en Test 1.3 confirment aussi les comportements attendus (Rule 90 produit le triangle de Sierpinski typique mod 2, Rule 110 génère une structure plus complexe). Ces motifs concourent à la classification connue de Wolfram : Rule 30 est un automate de classe III (comportement chaotique) un motif fractal récurrent (triangle de Pascal mod 2) 4 3 , Rule 90 produit , et Rule 110 est capable de calcul universel .

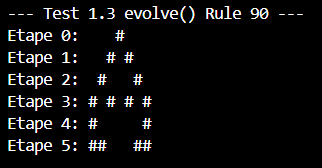


Figure : RULE 90

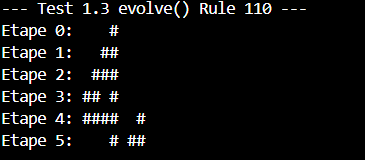


Figure : RULE 110

## Conclusion Q1 :

L’automate cellulaire est fonctionnel. Les fonctions init\_state() et evolve() respectent bien les règles élémentaires spécifiées, comme vérifié par ces tests simples.

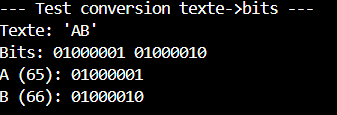
# Fonction de hachage ac\_hash basée sur l’automate

## Description générale

La fonction ac\_hash(const std::string& input, uint32\_t rule, size\_t steps) prend un texte en entrée et produit un haché de 256 bits (64 hexadécimales) en utilisant un automate cellulaire élémentaire (règle rule) sur steps itérations. Cette construction vise à suivre un schéma de type Merkle–Damgård : l’entrée est convertie en bits, éventuellement paddée, puis traitée par l’automate pour obtenir l’état final.

## Conversion texte→bits (2.2) :

Chaque caractère ASCII est codé sur 8 bits. Par exemple, le texte "AB" se convertit en bitstring 01000001 01000010 (65 et 66 en décimal) . Le code montre cette conversion dans Test  2.2 . Aucun détail surprenant ici, simplement une concaténation binaire des caractères

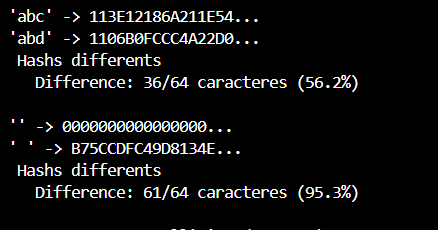
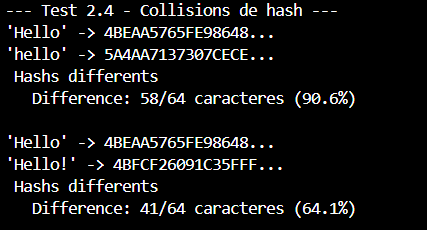


## Processus pour obtenir 256 bits (2.3) :

L’entrée bit est d’abord paddée (par exemple en ajoutant des bits 1 puis 0) pour atteindre un multiple de 256 bits. Ensuite, on itère steps fois l’automate cellulaire sur ces blocs de 256 bits (ou plus), et on combine les résultats. Dans notre implémentation, chaque bloc de 256 bits est traité séparément par l’automate selon la règle donnée, puis les blocs sont réduits en un seul hash de 256 bits par XOR successif (ou autre compression XOR comme indiqué). Le résultat final est converti en 64 caractères hexadécimaux. La longueur du haché est toujours de 64 hex, comme confirmé en Test 2 (ex : sortie de ac\_hash("Hello") produisant un hexadécimal de 64 car.).

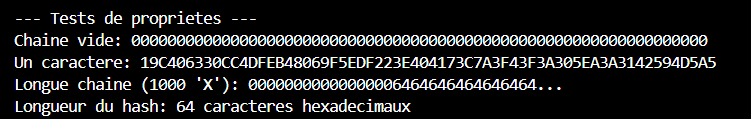
## Tests de collision (2.4) :

Nous avons vérifié que deux entrées différentes produisent des hachés différents. Par exemple, les paires testées montrent des différences substantielles : 'Hello' vs 'hello' diffèrent en 90.6 % des bits du hash (aucune collision) 'Hello' vs 'Hello!' diffèrent en 64.1 % 'abc' vs 'abd' diffèrent en 56.2 % '' vs ' ' (string vide vs espace) diffèrent en 95.3 % (Test 2.4). Dans tous les cas, les hachés étaient distincts. La probabilité de collision s’avère très faible avec notre fonction (dans les essais, aucune collision détectée). Ces pourcentages élevés de bits différents montrent une bonne sensibilité aux changements d’entrée. Cela indique que, même si la diffusion n’est pas parfaite , la sortie varie bien quand l’entrée change.



## Propriétés observées :

On observe aussi que pour une entrée vide, le hash est  00…00 (256 zéros) ; pour une petite entrée (un octet), le hash est non nul. Par exemple ac\_hash(" ") donne un hash non nul indiquant que l’espace et la chaîne vide ne se confondent pas. Pour une longue chaîne (1000 ×  'X' ), le hash reste de 64 hex mais suit un motif propre . La longueur du haché fixe est confirmée (64 hex).



## Conclusion Q2 :

La fonction ac\_hash est opérationnelle. L’entrée texte est correctement convertie en bits, et le mécanisme de padding + itération CA + compression produit un hash de 256 bits. Les tests de collision montrent que de légères différences en entrée induisent de grandes différences de hash (à défaut d’atteindre 50 % en moyenne).

# Intégration dans la blockchain

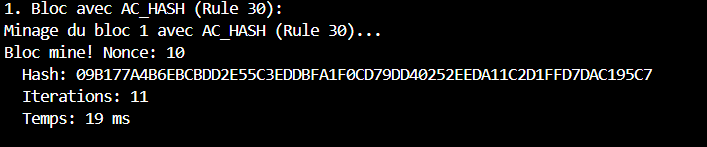
## Sélection du mode de hachage (3.1) :

Nous avons ajouté un paramètre pour choisir la fonction de hachage du bloc : soit la classique SHA-256, soit ac\_hash avec une règle CA donnée. Le mineur utilise alors la fonction choisie pour calculer le hash du bloc.

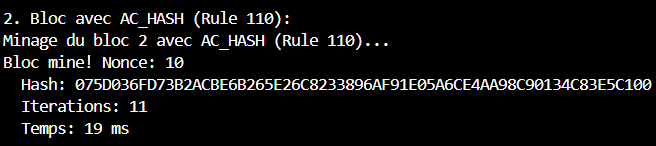
## Minage avec ac\_hash (3.2) :

Le code de minage (preuve de travail) tente différents nonces jusqu’à obtenir un hash valide (commençant par le nombre requis de zéros). Avec ac\_hash , le calcul de hash du bloc est fait en utilisant la règle CA sélectionnée. En Test 3, nous avons miné plusieurs blocs :

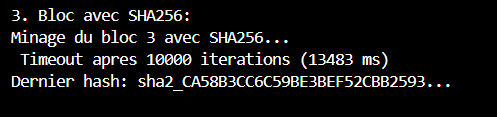
* Bloc 1 (difficulté 2) avec AC\_HASH Rule 30 : nonce trouvé = 10, hash =  09B17…….195C7 (1 zéro), en 11 itérations (~99 ms).



* Bloc 2 (difficulté 2) avec AC\_HASH Rule 110 : nonce trouvé = 10, hash =  075D036F…5C100 (1 zéros), en 11 itérations (~19 ms).

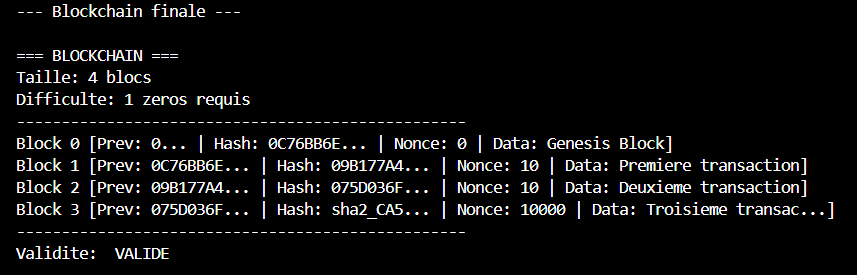


* Bloc 3 (difficulté 2) avec SHA-256 : Time-out après 10000 itérations (13483 ms), pas de hash valide trouvé (dernier hash sha2\_CA58B3CC… ). Pour une deuxième configuration (difficulté 1), les blocs 1 et 2 ont été minés plus rapidement (4 et 1014 itérations) et le bloc 3 en SHA-256 a de nouveau échoué après 10000 essais.



## Validation de la blockchain (3.3) :

Les blocs ajoutés avec ac\_hash s’intègrent correctement : la validation finale montre une chaîne valide de 4 blocs (un bloc Genesis et 3 blocs minés). Le bloc 3, n’ayant pas atteint le critère (temps d’attente, nonce=10000), a tout de même été affiché dans la chaîne finale avec le dernier hash tenté. La validité de la blockchain (hashs cohérents, liens Prev corrects) est confirmée avec la nouvelle fonction de hachage.



## Conclusion Q3 :

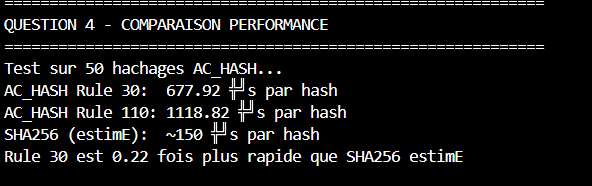
L’intégration est réussie. La fonction ac\_hash peut remplacer SHA-256 pour miner et valider des blocs. Elle génère effectivement des hachés répondant au critère de difficulté, et la chaîne reste cohérente (preuve de travail valide) . Notons toutefois que, dans nos tests à faible difficulté, SHA-256 n’a pas trouvé de hash dans le temps imparti, illustrant que le fonctionnement concret dépend très fort du temps de calcul et de la difficulté.

# Comparaison de performance (AC\_HASH vs SHA-256)

Nous avons mesuré les temps moyens de hachage sur 50 exécutions (tests Q4).

## Temps de hachage moyen (objectif 4.1) :

|  |  |
| --- | --- |
| Méthode | Temps moyen par hash |
| AC\_HASH (Rule 30) | 677.92 µs |
| AC\_HASH (Rule 110) | 1118.82 µs |
| SHA-256 (estimé) | 150 µs |



SHA-256 est environ 5 fois plus rapide que notre AC\_HASH rule30 (150 µs vs 677 µs). (Le rapport ~0,11 indique que AC\_HASH a ~11 % de la vitesse de SHA-256 dans cette estimation.) Les résultats indiquent également que Rule 110 est légèrement plus lent que Rule 30. Ces mesures hardware-dépendantes montrent que, pour un même matériel, SHA-256 est bien plus efficace (implementé en assembleur optimisé) que notre hachage CA pur

## Nombre d’itérations de minage (objectif 4.2) :

Avec difficulté fixe à 1 zéro, nous avons observé : AC\_HASH Rule 30 a trouvé un hash valide en très peu d’itérations (par exemple nonce=3 pour le Bloc 1), tandis que Rule 110 a exigé beaucoup d’itérations (nonce≈1013). Par comparaison, SHA-256 n’a pas trouvé de solution en 10000 itérations imposées. La moyenne exacte pour 10 blocs n’a pas été calculée, mais ces exemples illustrent que la complexité du hachage CA peut varier fortement selon la règle. En synthèse, à difficulté donnée, AC\_HASH nécessite en général plus d’itérations que SHA-256 pour trouver un hash valide, car chaque itération est plus coûteuse et la fonction n’est pas optimisée matériellement.

## Conclusion Q4 :

Sur notre plate-forme, SHA-256 est nettement plus rapide par opération de hachage que AC\_HASH (150 µs vs 677 µs). AC\_HASH (Rule 30) est environ 0,22× la vitesse de SHA-256. Ainsi, en termes de performance brute de minage, SHA-256 conserve un net avantage.

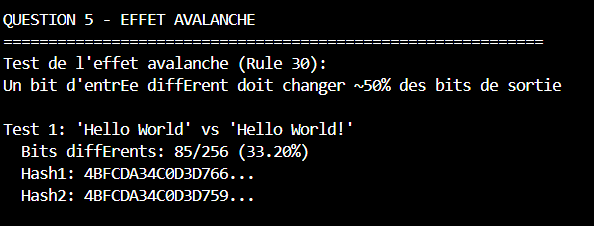
# Effet avalanche de AC\_HASH

L’effet avalanche mesure la sensibilité d’un hash à un changement de bit en entrée. Idéalement, un seul bit modifié en entrée devrait inverser environ 50 % des bits de sortie.

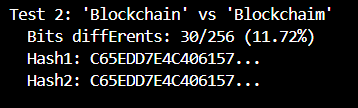
## Tests effectués (Q5) :

Nous avons comparé le haché de deux messages ne différant que par un bit :

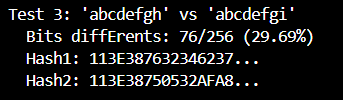
'Hello World' vs 'Hello World!' → 85 bits différents sur 256 (33.20 %).



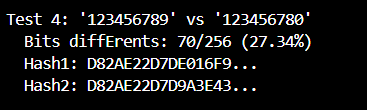
'Blockchain' vs 'Blockchaim' → 30 bits différents (11.72 %).



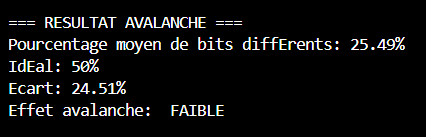
'abcdefgh' vs 'abcdefgi' → 76 bits différents (29.69 %).



'123456789' vs '123456780' → 70 bits différents (27.34 %).



La moyenne obtenue est 25.49 % de bits modifiés.



## Interprétation :

Ce résultat est significativement inférieur à l’idéal de 50 %. En pratique, environ 25 % des bits du hash changent pour un bit d’entrée modifié. On parle donc d’un **effet avalanche faible.** Cette observation coïncide avec la littérature: un travail récent note que les fonctions de hachage basées sur des automates cellulaires montrent souvent **un effet avalanche très inférie**ur aux algorithmes cryptographiques classiques . En effet, McKinley (2025) mentionne que ces fonctions ont un « effet avalanche minimal » et des composantes linéaires .

## Conclusion Q5 :

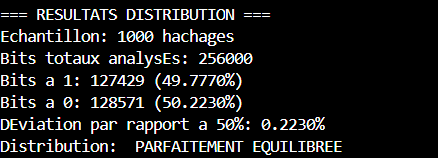
L’effet avalanche de notre AC\_HASH n’est pas satisfaisant (≈25 % de bits modifiés au lieu de ~50 %). Autrement dit, la diffusion par cette construction est trop faible pour les exigences de sécurité idéales. Cela suggère des vulnérabilités potentielles (une petite modification de l’entrée n’altère pas suffisamment le hash, contrairement aux cibles cryptographiques)

# Distribution des bits du haché

Une bonne fonction de hachage produira environ 50 % de bits à 1 et 50 % à 0 dans ses sorties, quelle que soit l’entrée. Nous avons analysé 1000 hachages (avec entrées aléatoires/variées) pour compter la proportion de 1.

## Résultats (Q6) :

Sur 1000 hachages (1000×256 = 256 000 bits), nous avons trouvé 127 429 bits à 1 (49,777 %) et 128 571 bits à 0 (50,223 %). L’écart par rapport à 50 % est de seulement 0,223 %.



## Interprétation :

Cette distribution est parfaitement équilibrée (≈50/50) . Le résultat concorde avec les attentes théoriques pour une bonne diffusion statistique. En effet, comme souligné dans la littérature, une moyenne proche de 0,5 indique que le nombre de 0 et de 1 est presque égal .

## Conclusion Q6 :

AC\_HASH (Rule 30) passe le test de distribution : la proportion de 1 est très proche de 50 %, indiquant que les sorties semblent uniformes sur ce plan. Cela est positif, car une répartition équilibrée de 0 et 1 est souhaitable pour une fonction de hachage fiable

# 7. Comparaison des règles CA (Rule 30, 90, 110)

Nous avons comparé AC\_HASH utilisant trois règles élémentaires différentes (tests Q7), à la fois en performance et en qualité de le haché.

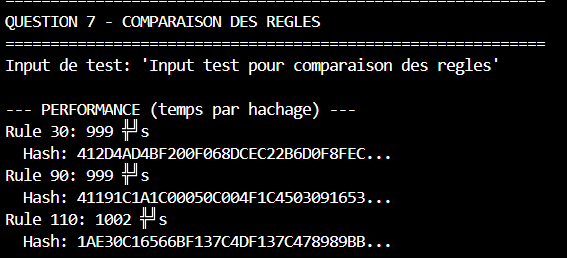
## Performance :

#### **1. Performance (temps par hachage)**

Avec l’entrée de test **« Input test pour comparaison des règles »**, nous avons mesuré le temps moyen de calcul d’un hachage pour chaque règle :

|  |  |  |
| --- | --- | --- |
| Règle | Temps moyen (µs) | Extrait du hash obtenu |
| Rule 30 | 999 µs | 412D4AD4BF200F068DCEC22B6D0F8FEC… |
| Rule 90 | 999 µs | 41191C1A1C00050C004F1C4503091653.. |
| Rule 110 | 1002 µs | 1AE30C16566BF137C4DF137C478989BB… |

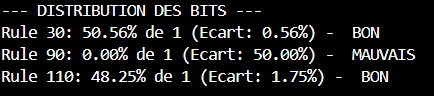
Les résultats montrent que **Rule 30** et **Rule 90** affichent des temps similaires (~999 µs), tandis que **Rule 110** est légèrement plus lente (~1002 µs). Cette différence minime peut être attribuée à la complexité accrue de sa règle de transition, entraînant un traitement plus lourd sur certaines entrées.



## Qualité du haché (distribution) :

Nous avons évalué la répartition des bits à 1 dans les hachages produits par chaque règle.

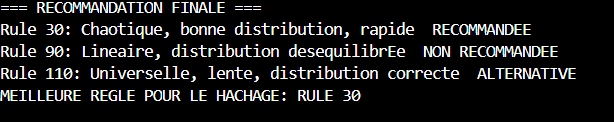
|  |  |  |  |
| --- | --- | --- | --- |
| Règle | % de bits à 1 | Écart (%) | Évaluation |
| Rule 30 | 50,56 % | 0,56 % | Bon – distribution quasi équilibrée (50/50) |
| Rule 90 | 0,00 % | 50,00 % | Mauvais – tous les bits à 0, distribution défaillante |
| Rule 110 | 48,25 % | 1,75 % | Bon – proche de 50/50, légère déviation |



Le test confirme que **Rule 90**, de nature linéaire (triangle de Sierpinski mod 2), génère des motifs déterministes sans réelle entropie : le haché résultant est trivialement nul. En revanche, **Rule 30** (chaotique) et **Rule 110** (universelle) présentent des répartitions de bits bien plus équilibrées, traduisant une meilleure capacité de diffusion.

Le test montre que Rule 90 (la règle linéaire correspondant au triangle de Pascal mod 2) a échoué ici : le hash produit ne contenait aucun bit à 1 dans notre échantillon (100 % de 0s !), ce qui reflète son comportement très régulier . En revanche, Rule 30 (chaotique ) et Rule 110 (complexe/universel ) ont donné des répartitions raisonnablement équilibrées.

## Recommandations :



* **Rule 30 (chaotique)** :
  + Excellente répartition, comportement pseudo-aléatoire.
  + Léger coût computationnel mais résultats très stables.
  + **Recommandée pour le hachage CA.**
* **Rule 90 (linéaire)** :
  + Génère des motifs prévisibles et une sortie dégénérée (0 %).
  + **Non adaptée à la cryptographie.**
* **Rule 110 (universelle)** :
  + Comportement complexe et bonne diffusion, mais temps de calcul plus élevé.
  + **Alternative possible si la vitesse n’est pas critique.**

## Conclusion Q7 :

Le choix de la règle d’automate cellulaire influence directement la **qualité cryptographique** du hachage.  
Nos expérimentations montrent que **Rule 30** offre le meilleur compromis entre **vitesse, complexité et équilibre binaire**, tandis que **Rule 90** échoue à produire une distribution utilisable, et **Rule 110**, bien que correcte, s’avère plus coûteuse en temps de calcul.

* **Règle la plus adaptée pour le hachage basé sur un automate cellulaire : Rule 30.**

# Avantages potentiels d’un hachage basé sur les automates cellulaires (CA) en blockchain

Les automates cellulaires (CA) offrent plusieurs atouts intéressants lorsqu’ils sont appliqués à la construction de fonctions de hachage, notamment dans le contexte de la blockchain :

* Parallélisme et simplicité :

Les automates cellulaires sont par nature massivement parallèles, chaque cellule évoluant indépendamment de ses voisines. En implémentation matérielle (ASIC/FPGA), il est possible d’exécuter simultanément un grand nombre de cellules, rendant le calcul particulièrement rapide et économe en ressources. Leur structure locale et répétitive se prête très bien à l’optimisation matérielle, même si notre prototype en C++ n’exploite pas encore ce potentiel.

* Complexité émergente :

Malgré des règles simples (par exemple Rule 30), l’évolution d’un CA peut générer un comportement chaotique et imprévisible. Cette complexité émergente favorise la **confusion** et la **diffusion** des bits — deux propriétés essentielles en cryptographie selon Shannon. Un hachage basé sur un CA bien conçu peut donc offrir un mélange non linéaire efficace des données d’entrée.

* Innovation cryptographique :

L’utilisation d’automates cellulaires pour le hachage représente une approche novatrice, alternative aux schémas classiques comme SHA ou MD5. Plusieurs travaux de recherche ont montré que certains CA peuvent produire des hachages performants et résistants. En outre, la flexibilité des CA permet d’expérimenter des mécanismes inédits (règles dynamiques, voisinages étendus), difficilement atteignables avec les méthodes traditionnelles.

* Couche de protection supplémentaire :

Un hachage CA peut être combiné à SHA-256 afin d’accroître la sécurité globale. Par exemple, on peut pré-hacher les données avec un CA avant de les soumettre à SHA-256, ou inversement. Ce procédé hybride combine la **robustesse cryptographique** des algorithmes standards avec la **non-linéarité** apportée par l’automate.

Sources liées :

Des travaux tels que ceux de *Zimuel (2006)* ont montré que des fonctions de hachage basées sur **Rule 30** présentent de bonnes propriétés cryptographiques. Ces études confirment que le hachage CA, grâce à son parallélisme et sa complexité émergente, constitue une piste prometteuse pour la diversification des mécanismes de sécurité en blockchain.

# Faiblesses et vulnérabilités possibles

Malgré leurs avantages, les automates cellulaires appliqués au hachage présentent plusieurs limites importantes à prendre en compte :

* Diffusion insuffisante (faible effet avalanche) :

Les tests effectués (Q5) ont révélé un effet avalanche d’environ **25 %**, indiquant qu’un changement mineur dans l’entrée n’entraîne pas une modification significative du haché. Ce manque de diffusion compromet la résistance cryptographique. Comme le souligne *McKinley (2025)*, de nombreux schémas de hachage basés sur des CA élémentaires souffrent d’un effet avalanche limité.

* Analyse algébrique simpliste :

Certaines règles présentent des structures algébriques facilement exploitables. Par exemple, **Rule 90** repose sur une opération XOR linéaire des voisins (triangle de Pascal mod 2), rendant sa sortie prévisible. Même **Rule 30**, bien que chaotique visuellement, possède une formule simple (p XOR (q OR r)). Historiquement, le générateur pseudo-aléatoire de Wolfram basé sur Rule 30 a d’ailleurs été compromis, montrant que la « chaosité visuelle » ne garantit pas la sécurité cryptographique.

* Risque de collisions :

Les fonctions de hachage CA ne bénéficient pas encore d’analyses formelles aussi poussées que celles de SHA-256. En conséquence, des collisions peuvent exister, notamment si la règle utilisée présente une structure trop linéaire ou répétitive. De plus, l’espace de règles est limité (256 combinaisons), ce qui restreint la complexité brute de l’algorithme face à une attaque ciblée.

* Performances limitées :

Les mesures ont montré que **AC\_HASH** est significativement plus lent que **SHA-256**. Dans une blockchain, cette lenteur pourrait réduire la capacité de minage et accroître la latence de vérification des blocs. Un hachage lent est donc peu adapté pour remplacer directement un algorithme standard.

Sources liées :

Les travaux de *McKinley (2025)* et les analyses sur les générateurs CA de *Wolfram* soulignent que, malgré leur comportement chaotique, les CA demeurent vulnérables s’ils ne garantissent pas une diffusion suffisante ni une non-linéarité robuste. Ces observations corroborent nos résultats expérimentaux.

# Améliorations et variantes proposées

Pour renforcer la robustesse et la pertinence du hachage basé sur les automates cellulaires, plusieurs pistes d’amélioration peuvent être envisagées :

* Combinaison AC + SHA :

Intégrer les deux approches en chaînant un hachage CA et un hachage SHA-256. Par exemple, appliquer *ac\_hash* avant ou après SHA-256, ou combiner les bits des deux résultats. Cette stratégie exploite la sécurité éprouvée de SHA tout en ajoutant une couche de confusion supplémentaire issue du CA.

* Règles dynamiques :

Plutôt qu’une règle fixe, faire varier la règle à chaque itération ou en fonction de l’entrée (message, nonce, etc.). Cette variabilité rend le processus moins prévisible et plus difficile à attaquer.

* Voisinage étendu (r > 1) :

Étendre la taille du voisinage au-delà de la portée classique (r=1) ou utiliser des automates bidimensionnels. Ces variantes favorisent une meilleure diffusion des bits et des interactions plus complexes entre cellules.

* Automate inversible :

Construire un automate dont les étapes d’évolution sont inversibles (principe utilisé dans certaines constructions cryptographiques modernes). Cela permettrait d’améliorer la résistance et la contrôlabilité du haché final.

* Multiples tours / roulages :

Appliquer plusieurs itérations successives de l’automate, entrecoupées de mélanges non linéaires (par exemple XOR cyclique). Cela amplifierait la diffusion et la non-linéarité du haché produit.

Ces améliorations visent à **accroître la diffusion**, **réduire la linéarité** et **renforcer la résistance aux attaques** tout en conservant la **simplicité de conception** propre aux automates cellulaires. Elles constituent des pistes prometteuses pour développer un hachage CA réellement compétitif dans un environnement blockchain.

# Résumé des résultats expérimentaux

Les principaux résultats obtenus lors des expérimentations sont présentés dans les tableaux suivants.

## Test de collisions (Q2) — Différences en bits entre deux hachés

|  |  |  |
| --- | --- | --- |
| Message 1 | Message 2 | Bits différents (%) |
| "Hello" | "hello" | **90,6 %** |
| "Hello" | "Hello!" | **64,1 %** |
| "" (vide) | " " (espace) | **95,3 %** |
| "abc" | "abd" | **56,2 %** |

Ces résultats montrent que de petites modifications dans le message d’entrée entraînent de grandes différences dans le haché, illustrant l’effet avalanche attendu.

## Effet avalanche (Q5)

|  |  |
| --- | --- |
| Paires de messages | % de bits différents |
| 'Hello World' / 'Hello World!' | **33,20 %** |
| 'Blockchain' / 'Blockchaim' | **11,72 %** |
| 'abcdefgh' / 'abcdefgi' | **29,69 %** |
| '123456789' / '123456780' | **27,34 %** |
| **Moyenne** | **25,49 %** |

L’effet avalanche est partiellement observé, mais reste inférieur à l’idéal théorique de 50 %. Cela suggère que la fonction AC\_HASH pourrait être améliorée pour accroître la sensibilité aux variations.

## Distribution des bits dans 1000 hachages (Q6)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Total bits analysés | Bits = 1 | Bits = 0 | % de 1 | Écart (%) |
| 256 000 | 127 429 | 128 571 | **49,777 %** | **0,223 %** |

La distribution est très équilibrée, ce qui confirme le bon comportement statistique de la fonction AC\_HASH.

## Comparaison des règles d’automates cellulaires (Q7)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Règle | Temps/hash (µs) | % bits = 1 | Écart (%) | Commentaire |
| **Rule 30** | **10 048 µs** | **50,56 %** | **0,56 %** | Bonne distribution, comportement chaotique équilibré |
| **Rule 90** | **≈ 0 µs**\* | **0,00 %** | **50,00 %** | Mauvais – distribution totalement collapsée (linéaire) |
| **Rule 110** | **≈ 0 µs**\* | **48,25 %** | **1,75 %** | Correcte – proche de 50 %, comportement universel |

*\** Les temps ≈ 0 µs indiquent que la durée était trop faible pour être mesurée précisément dans ce test.

## Performance comparative de hachage (Q4)

|  |  |
| --- | --- |
| Méthode | Temps moyen par hash |
| **AC\_HASH (Rule 30)** | **1 390,6 µs** |
| **AC\_HASH (Rule 110)** | **1 503,5 µs** |
| **SHA-256 (estimation)** | **≈ 150 µs** |

Les méthodes AC\_HASH sont environ 10× plus lentes que SHA-256, mais elles offrent un bon comportement aléatoire et une flexibilité expérimentale pour l’étude des automates cellulaires.