DEPARTMENT OF
COMPUTER SCIENCE
計算機科學系

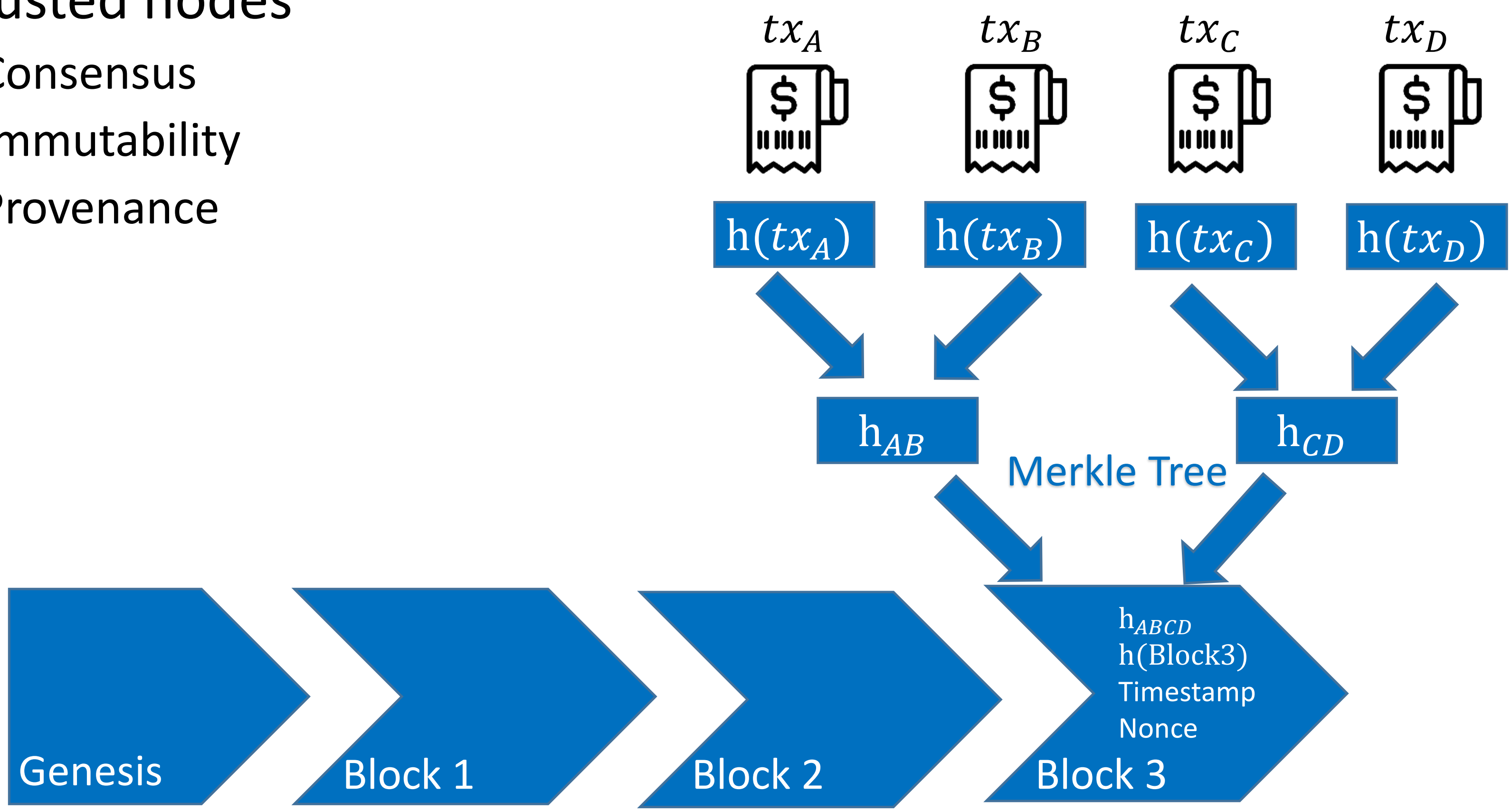
香港浸會大學
HONG KONG BAPTIST UNIVERSITY

Authenticated Keyword Search in Scalable Hybrid-Storage Blockchain

Ce Zhang, Cheng Xu, Haixin Wang, Jianliang Xu, Byron Choi
Hong Kong Baptist University

Blockchain Technology

- Distributed ledger maintained by a network of mutually untrusted nodes
  - Consensus
  - Immutability
  - Provenance



Smart Contract

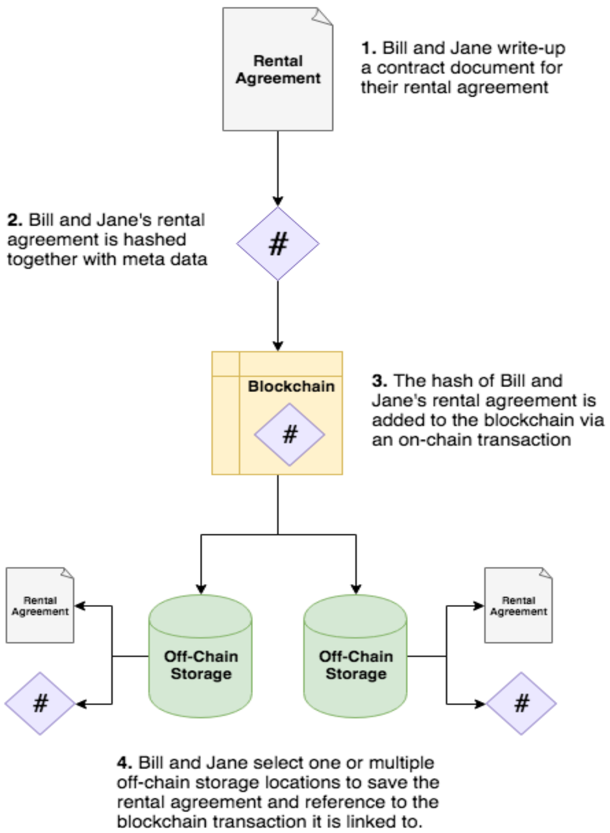
- A user-defined program executed by the blockchain network
  - Interact with the data stored in the blockchain
  - Execution integrity is ensured by the consensus protocol
- Smart contract makes blockchain to be programmable
  - Facilitate automatic logics without participation of third parties

	Traditional Computer	Blockchain VM
Storage	RAM	Blockchain
Computation	CPU	Smart Contract

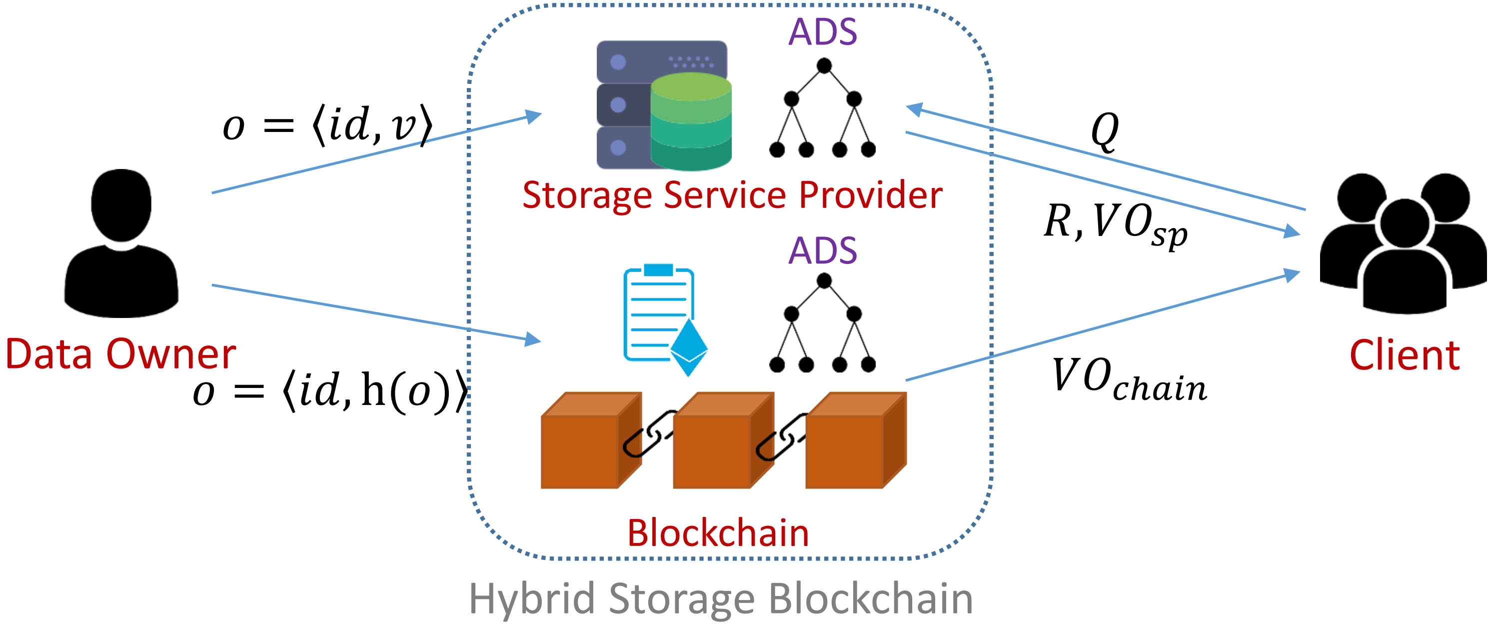
Blockchain Scalability

- Storing *any* information on chain is not scalable
  - Large size: document, image, etc.
  - 500KB per TX x 500 TX per sec => 2 Gb per sec => 8,000 TB annually
- Off-chain storage:
  - Raw data is stored outside of the blockchain
  - A hash of the data is kept on chain to ensure integrity

Example: BACK ALLEY CODER



Hybrid Storage Blockchain



- Integrity-assured queries needed as the SP is not fully trustful
- Key idea: authenticated query processing
  - Use an authenticated data structure (ADS) to support queries
  - Leverage both smart contract and SP to maintain the ADS

Keyword Search Queries

- Keywords are expressed in the disjunctive normal form (DNF)
- $Q = q_1 \vee q_2 \vee \cdots \vee q_n$, where $q_i = w_1 \wedge w_2 \wedge \cdots \wedge w_l$
- Example: $(\text{"COVID}-\text{19"} \wedge \text{"Vaccine"}) \vee (\text{"SARS}-\text{CoV}-\text{2"} \wedge \text{"Vaccine"})$

- Seen as the union of the results from each conjunctive component
- Focus on *conjunctive keyword search*

Challenge

- High update cost: each on-chain update requires a transaction
- Transaction fee for smart contract execution
  - Modeled by gas for storage and computation (Ethereum)
- Challenge: how to design gas-efficient ADS to be maintained by the smart contract while supporting efficient keyword search

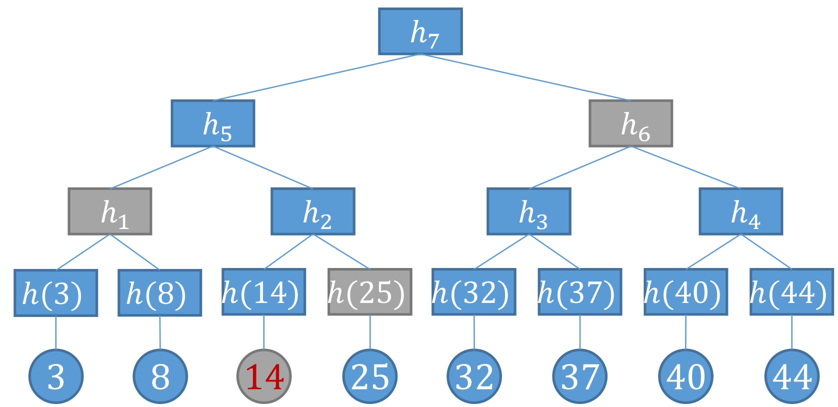
Operation	Gas Used	Explanation
C_{sload}	200	load a word from storage
C_{sstore}	20,000	save a word to storage
$C_{supdate}$	5,000	update a word to storage
C_{mem}	3	access a word in memory
C_{hash}	$30 + 6 \cdot x$	hash a x-word message
C_{tx}	21,000	execute a transaction
C_{txdata}	68	transact a byte of data

Contributions

- Suppressed Merkle^{inv} index
  - Reduce the ADS maintenance cost in terms of gas
- Chameleon^{inv} index
  - Further reduce the ADS maintenance cost to a constant level while still supporting efficient queries
- Optimized Chameleon^{inv*} index
  - Enhance the query and verification performance of the Chameleon^{inv} index

Preliminaries

- ADS: Merkle Hash Tree (MHT)
  - Binary tree
  - Hash function combining the child nodes
  - Verification object (VO): sibling hashes along the search path
  - Verify: reconstructing the root hash

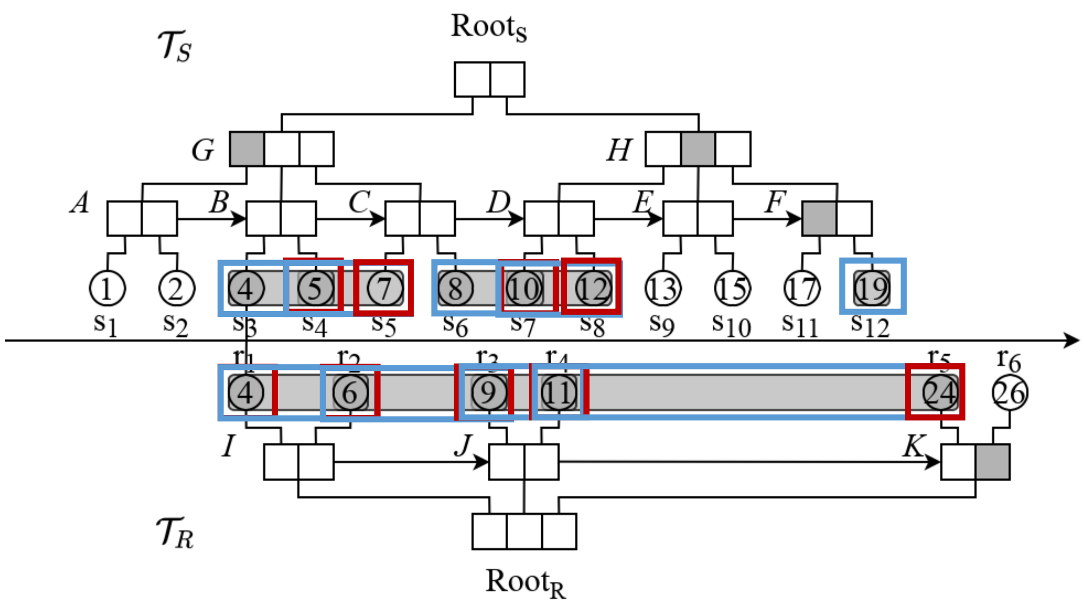


To authenticate object: 14
VO: $\{\mathrm{h}(25), h_1, h_6\}$

- Merkle B-tree (MB-tree)
  - Integrate B-tree with MHT

Preliminaries

- Authenticated join with MB-tree
  - Executed in rounds and each round has a target with matching and boundary objects
  - Proof includes the Merkle path of the targets and boundary objects
  - Verification: reconstruct the Merkle roots and check the boundary objects with the corresponding targets

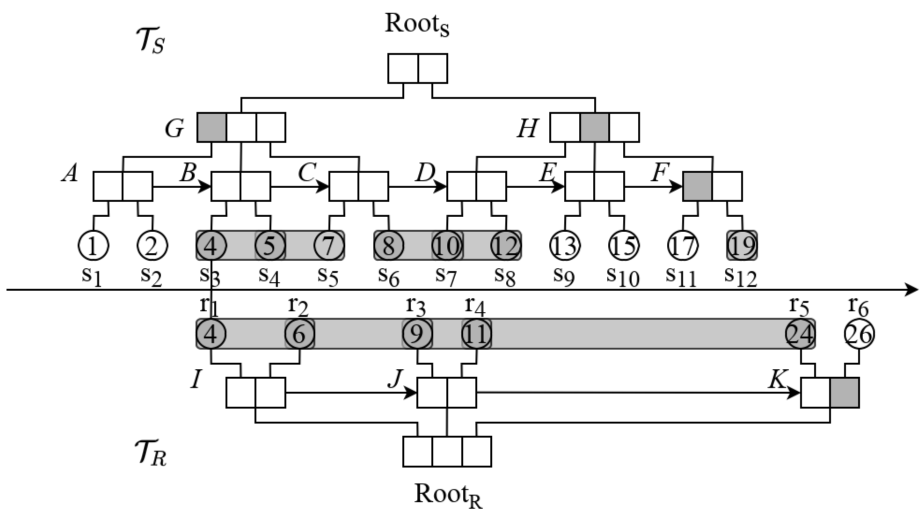


Baseline Solution

- Merkle^{inv} index
  - Build an inverted index
  - Maintain an MB-tree for each keyword's object list
  - MB-tree's search key is object ID

  - Query processing: a conjunctive keyword query is equivalent to joining keywords' object lists

Keyword ID	Keyword w		Object List for w
1	COVID-19	$\mapsto$	1, 2, 4, 5, 7, 8, 10, 12, 13, 15, 17, 19
2	Symptom	$\mapsto$	4, 6, 9, 11, 24, 26
3	SARS-CoV-2	$\mapsto$	1, 3
4	Vaccine	$\mapsto$	4, 5, 8

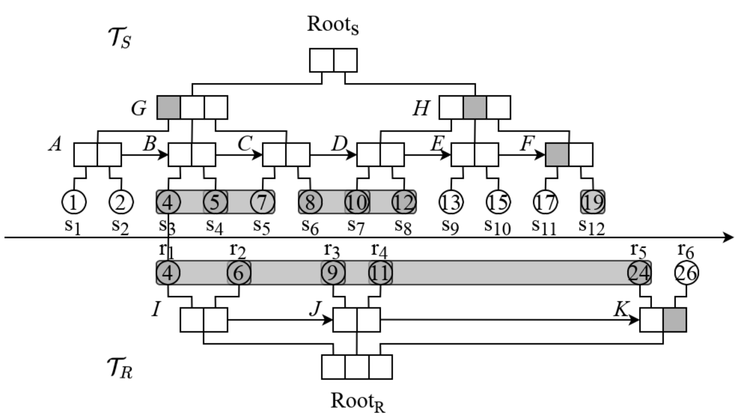


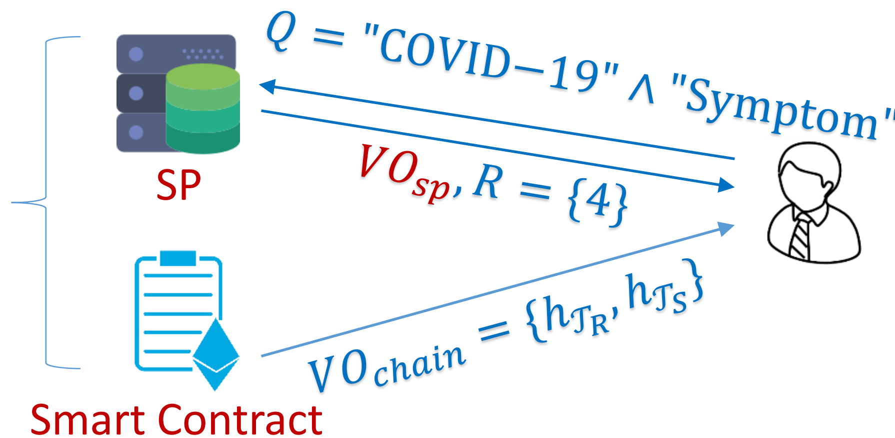
香港浸會大學 HONG KONG BAPTIST UNIVERSITY

Baseline Solution

- Merkle^{inv} index
  - Maintained by both the smart contract and the SP

Keyword ID	**Keyword** w		**Object List for** w
1	COVID-19	$\mapsto$	1, 2, 4, 5, 7, 8, 10, 12, 13, 15, 17, 19
2	Symptom	$\mapsto$	4, 6, 9, 11, 24, 26
3	SARS-CoV-2	$\mapsto$	1, 3
4	Vaccine	$\mapsto$	4, 5, 8





VO_{sp}: Merkle proofs of targets, matching & boundary objects

香港浸會大學 HONG KONG BAPTIST UNIVERSITY

Baseline Solution

- Merkle^{inv} index maintenance
  - When o_i is added to the Merkle^{inv} index, $\langle o_i.id, \text{h}(o_i)\rangle$ is inserted to the MB-tree of each its keyword

  - Suffer from high maintenance cost
  - Data update requires hash updates on the entire tree path
  - Cost of adding an object to a single keywords' MB-tree is logarithmic w.r.t. expensive storage operations

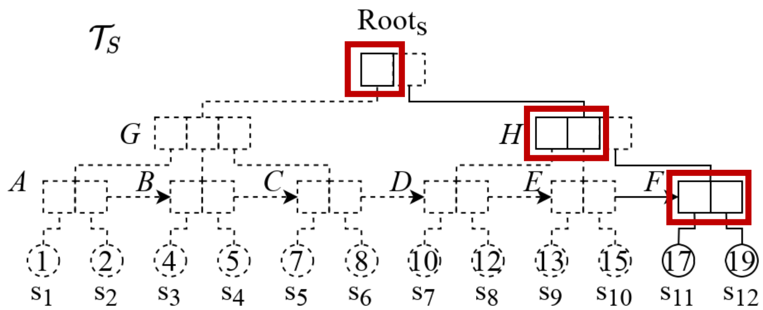
$$C_{\text{MI}}^{\text{insert}} = \log_F N \left(2C_{sstore} + 2C_{supdate} + (2F+1)C_{sload} + C_{hash}\right) + C_{sstore}$$

Suppressed Merkle^{inv} index

- Observation: only on-chain root hashes (VO_{chain}) are needed during the authenticated keyword search
- General idea:
  - Fully suppress the on-chain MB-trees
  - The SP maintains the complete structures to support efficient queries
  - Key issue: how can the smart contract maintain the root hashes without knowing the complete structure?
    - Ask the off-chain SP to construct an *update proof*, during a new object's insertion
    - With *update proof*, MB-trees' root hashes can be updated

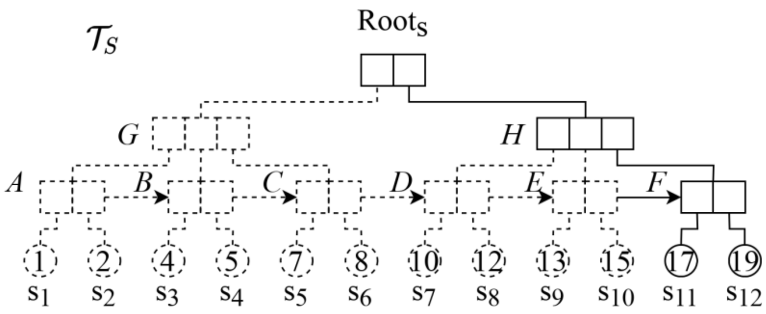
Suppressed Merkle^{inv} index

- Generation of *update proof* for a MB-tree by the SP
  - Assuming object ids are monotonically increasing
  - Include the tree path of the right-most leaf node
- Example for $\mathcal{T}_S$
  - A new object s_{13} with $id = 23$ is added to $\mathcal{T}_S$
  - Update proof: (i) $\langle h_G \rangle$; (ii) $\langle h_D, h_E \rangle$; (iii) $\langle h_{s_{11}}, h_{s_{12}} \rangle$; (iv) $\langle h_{s_{13}} \rangle$



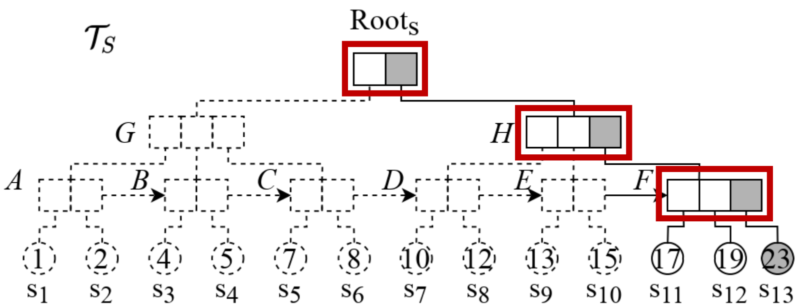
Suppressed Merkle^{inv} index

- Verification of *update proof* by smart contract
  - Reconstruct the root hash and compare with the one stored on-chain
- Example for $\mathcal{T}_S$
  - $\mathrm{h}\left(h_G|\mathrm{h}\left(h_D|h_E|\mathrm{h}\left(h_{s_{11}}|h_{s_{12}}\right)\right)\right)$ and compare it with the one stored on-chain
  - Check $h_{s_{13}}$ w.r.t. the one sent by DO



Suppressed Merkle^{inv} index

- Update the root hash using *update proof* by smart contract
  - in a bottom-top manner
- Example for $\mathcal{T}_S$
  - Object s_{13} with $id = 23$ is added to $\mathcal{T}_S$
  - Leaf F's node hash: $h'_F = \mathrm{h}(h_{s_{11}} | h_{s_{12}} | h_{s_{13}})$
  - Node H's: $h'_H = \mathrm{h}(h_D | h_E | h'_F)$
  - Root hash: $\mathrm{h}(h_G | h'_H)$



Suppressed $\mathbf{Merkle}^{inv}$ index

- Cost Analysis
  - Consider updating the MB-tree for a single keyword
  - $C_{\text{SMI}}^{\text{insert}} = \log_F N \left(F \cdot |h| \cdot C_{txdata} + 3C_{hash} + (2F+1)C_{mem}\right) + 2C_{sload} + C_{supdate}$
  - The coefficient of logarithmic term only contains cheap operations: C_{txdata}, C_{hash}, C_{mem}
  - The costly operations C_{sload}, $C_{supdate}$ are with a constant coefficient
  - $C_{\text{SMI}}^{\text{insert}} < C_{\text{MI}}^{\text{insert}}$

We have reduced the maintenance cost. Can we do even better?

Preliminaries

- Vector Commitment (VC)
  - VC maps a vector of messages to a fixed-sized commitment, which can be used to prove that m_i is the i^{th} committed message

$\vec{m} =$

m_1	m_2	m_3	…	m_i	…	m_{q-1}	m_q

- $\mathrm{Gen}(1^\lambda, q) \rightarrow \mathrm{pp}$
- $\mathrm{Com}_{\mathrm{pp}}(\langle m_1, \ldots, m_q \rangle, r) \rightarrow \{c, \mathrm{aux}\}$
- $\mathrm{Open}_{\mathrm{pp}}(i, m, \mathrm{aux}) \rightarrow \pi$
- $\mathrm{Ver}_{\mathrm{pp}}(c, i, m, \pi) \rightarrow 0/1$

HONG KONG BAPTIST UNIVERSITY 香港浸會大學

Preliminaries

- Chameleon Vector Commitment (CVC)
  - A CVC is a trapdoor vector commitment scheme. A user who owns a private trapdoor can update a message m_i in a vector without changing the vector's commitment.

$\vec{m} =$

m_1	m_2	m_3	…	$m_i{}'$	…	m_{q-1}	m_q

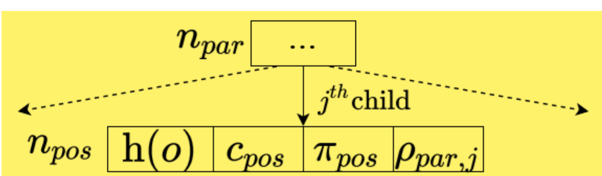
- $\mathrm{Gen}(1^\lambda, q) \to \{\mathrm{pp}, \mathrm{td}\}$
- $\mathrm{Com}_{\mathrm{pp}}(\langle m_1, \dots, m_q \rangle, r) \to \{c, \mathrm{aux}\}$
- $\mathrm{Open}_{\mathrm{pp}}(i, m, \mathrm{aux}) \to \pi$
- $\mathrm{Ver}_{\mathrm{pp}}(c, i, m, \pi) \to 0/1$
- $\mathrm{CCol}_{\mathrm{pp}}(c, i, m, m', \mathrm{td}, \mathrm{aux}) \to \mathrm{aux}'$
  - $\mathrm{Open}_{\mathrm{pp}}(i, m', \mathrm{aux}') \to \pi'$
  - $\mathrm{Ver}_{\mathrm{pp}}(c, i, m', \pi') \to 0/1$

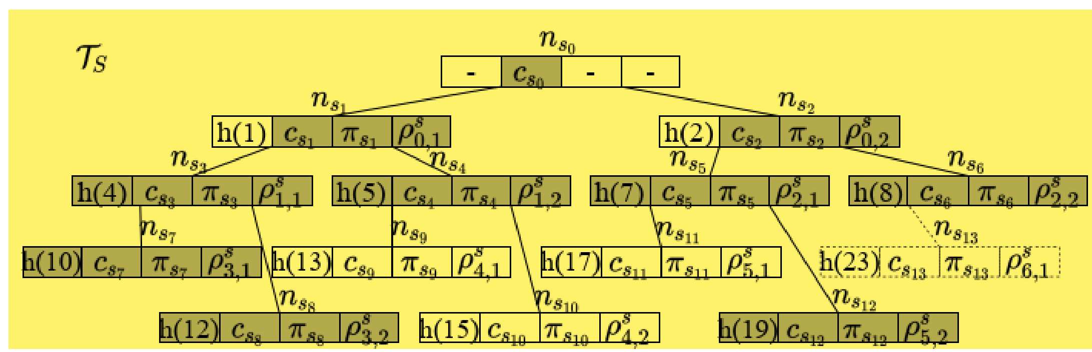
Chameleon^{inv} index

- Objective
  - Design an ADS that has constant maintenance cost while supports efficient authenticated keyword search
- Inspiration
  - CVC: one can update a vector without changing its digest using a secret trapdoor
  - Build a Chameleon tree with fixed root commitment

Chameleon^{inv} index

- Chameleon Tree
  - Each node (except the root) corresponds to a data object
  - Each node's commitment is determined by its position pos and keyword w
  - We use the root commitment c_0 and current object number cnt to authenticate the tree



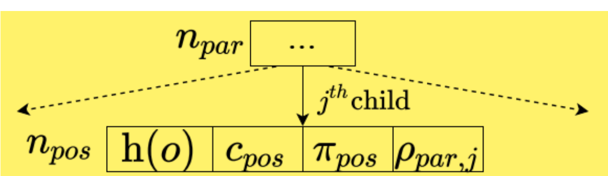


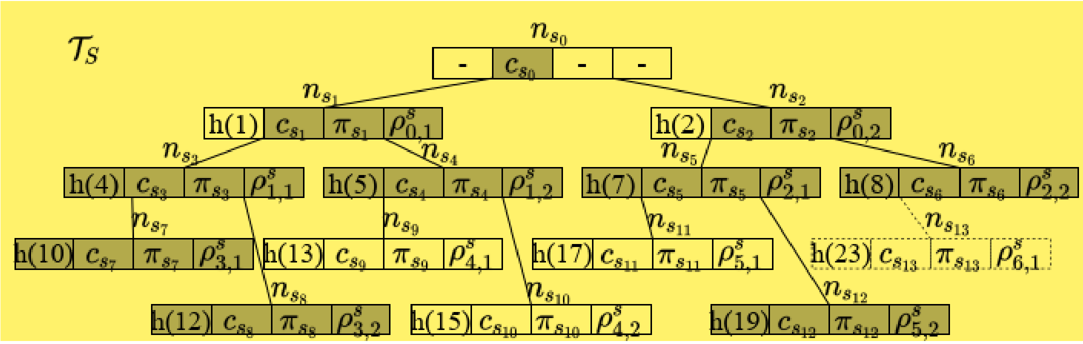
Chameleon^{inv} index

- Chameleon Tree
  - Each non-root node is a 4-tuple $\langle h(\text{o}), c_{pos}, \pi_{pos}, \rho_{par,j} \rangle$
  - $h(o)$: the hash of object o
  - c_{pos} is the node commitment derived from pos and keyword w:

$$c_{pos} = \text{Com}_{\text{pp}}(\langle 0, \dots, 0 \rangle, PRF(sk, pos||w))$$

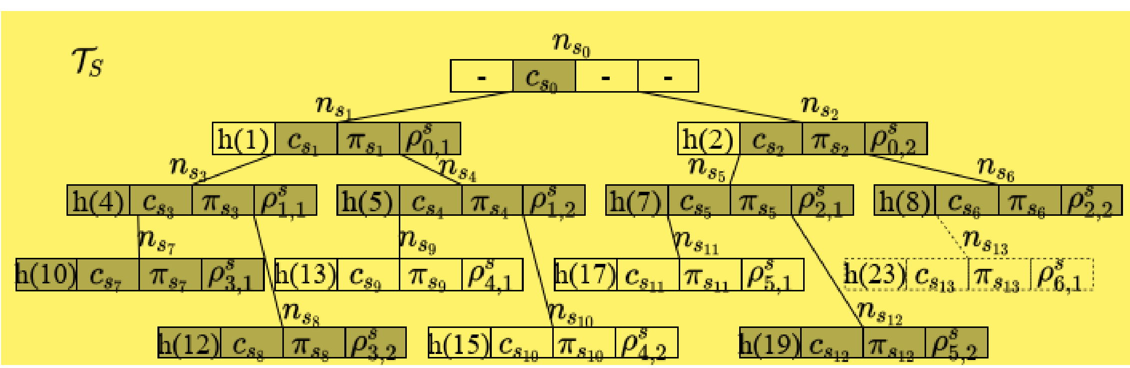
  - π_{pos} proves that $h(o)$ is the 1st element stored in c_{pos} (find collision of c_{pos})
  - $\rho_{par,j}$ proves that the node is linked to the j^{th} child of the parent node at position par (find collision of c_{par})





Chameleon^{inv} index

- Chameleon Tree Maintenance
  - Create a new node for o -> compute c_{pos}, π_{pos}
  - Link the new node to its parent node -> compute $\rho_{par,j}$
  - Store $\langle c_{pos}, \pi_{pos}, \rho_{par,j} \rangle$ as the insertion proof of o
- Chameleon^{inv} index
  - Each keyword corresponds to a Chameleon tree.
  - Constant maintenance cost: $C_{\text{Chameleon}}^{\text{insert}} = C_{supdate}$



SP

$\langle c_{pos}, \pi_{pos}, \rho_{par,j} \rangle$

Smart Contract

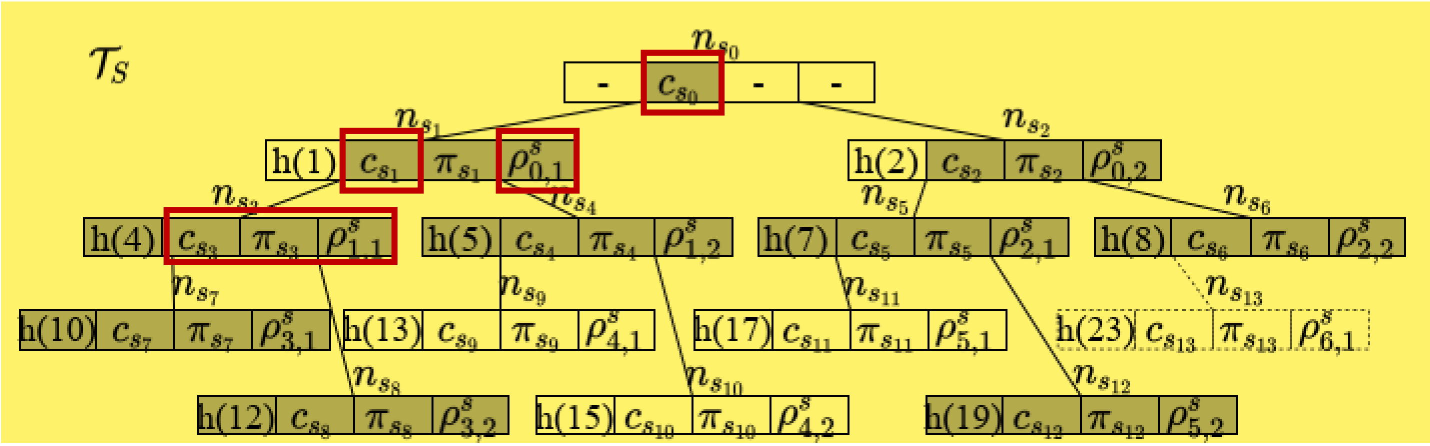
$\langle c_0, cnt \rangle$

Chameleon^{inv} index

- Keyword search query processing
  - A keyword search is transformed to join the query keywords' Chameleon trees for each conjunctive component
  - Build a hash map for $\langle id, pos \rangle$ since the Chameleon tree is indexed by the position
  - Add the membership proofs of (i) target; (ii) matching & boundary objects of each round to VO_{sp}

Chameleon^{inv} index

- Authenticated membership test with Chameleon Tree
  - Given object's position pos, the SP generates a membership proof
  - Include the insertion proofs of the object at pos and all its ancestor nodes except the root
  - Example: s_3's membership proof $\{c_{s_3}, \pi_{s_3}, \rho^s_{1,1}, c_{s_1}, \rho^s_{0,1}\}$
  - Verification: use π_{s_3} to prove s_3 is stored in n_{s_3}; use $\rho^s_{1,1}$ to prove n_{s_3} is the first child of n_{s_1}; use $\rho^s_{0,1}$ and root commitment c_{s_0} to prove n_{s_1} is the first child of the root.



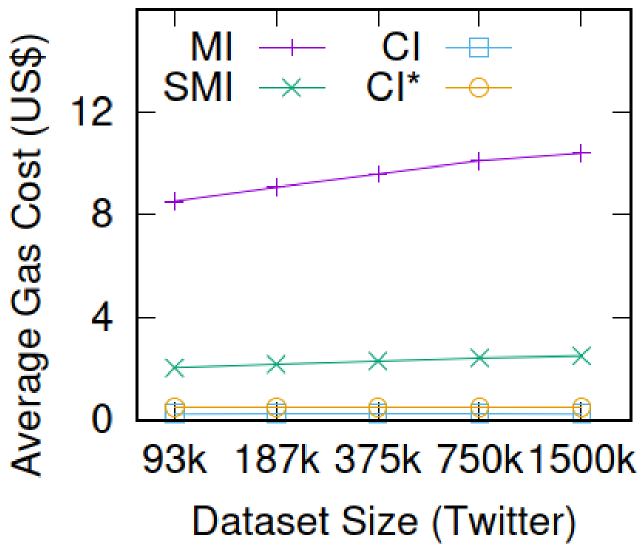
Chameleon^{inv*} index

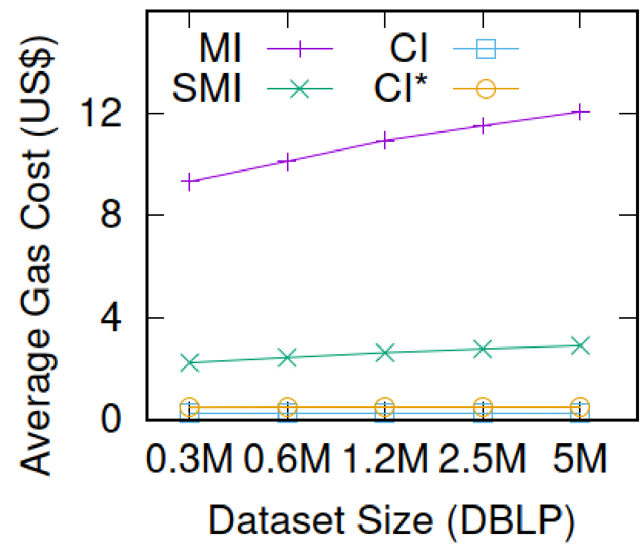
- Mitigate the client's verification cost
  - Create a Bloom filter for every b objects in each Chameleon tree
  - A Bloom filter can efficiently prove an object's non-existence
  - Smart contract maintains the Bloom filters for integrity assurance
- Authenticated Keyword Search
  - Similar to Chameleon^{inv} index
  - Use Bloom filters in the second index to test whether a matching object exists
    - If existing, proceed as Chameleon^{inv} index
    - Otherwise, the consecutive object is set as the target to continue the join process

Performance Evaluation

- Datasets
  - DBLP: 5M paper entries including titles, authors, and affiliations
  - Twitter: 1.5M tweets
  - 32-bit incremental identifier
- Parameters of the index
  - Fan-out of the MB-tree set to 4 according to the word size 32 bytes
    - $(f-1)l_d + f l_p < 32$byte
  - Fan-out of Chameleon tree is set to 4
  - Fixed Bloom filter size: 256 bytes
  - # objects inserted to a Bloom filter $b = 30$
- Denote the four indexes as MI, SMI, CI, and CI^*

Gas Consumption vs Dataset Size

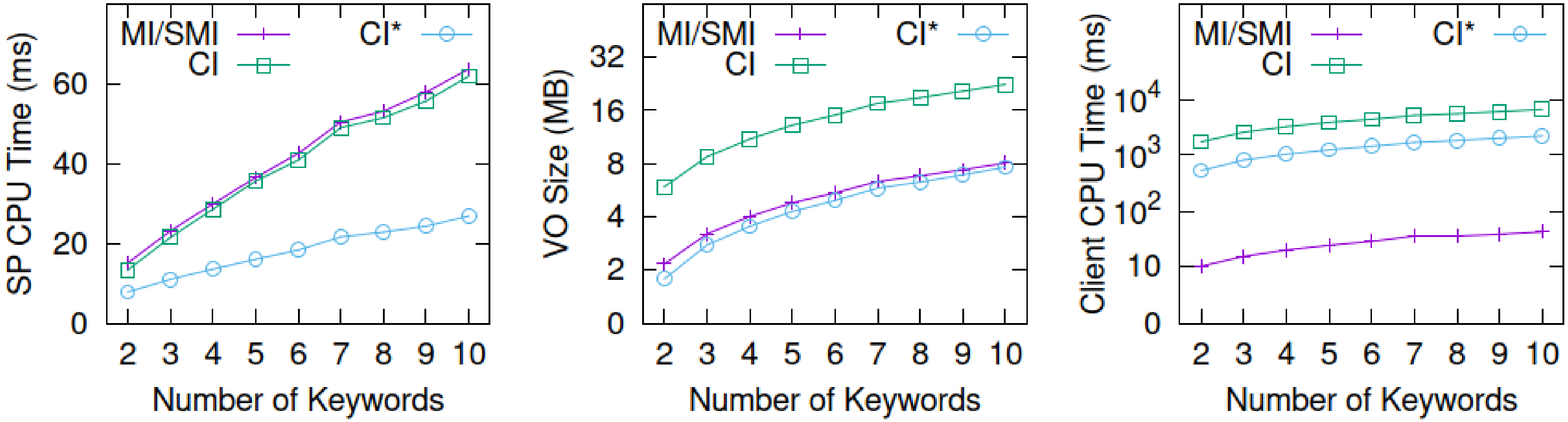




- SMI reduces the average gas consumption from US$11.21 to US$2.69 (saving 76%)
- CI takes US$0.24 and CI* takes US$0.50 for each insertion

The gas consumption is reported in US$ with an average gas price of 15 Gwei and Ether price of US$229 as of June 15, 2020.

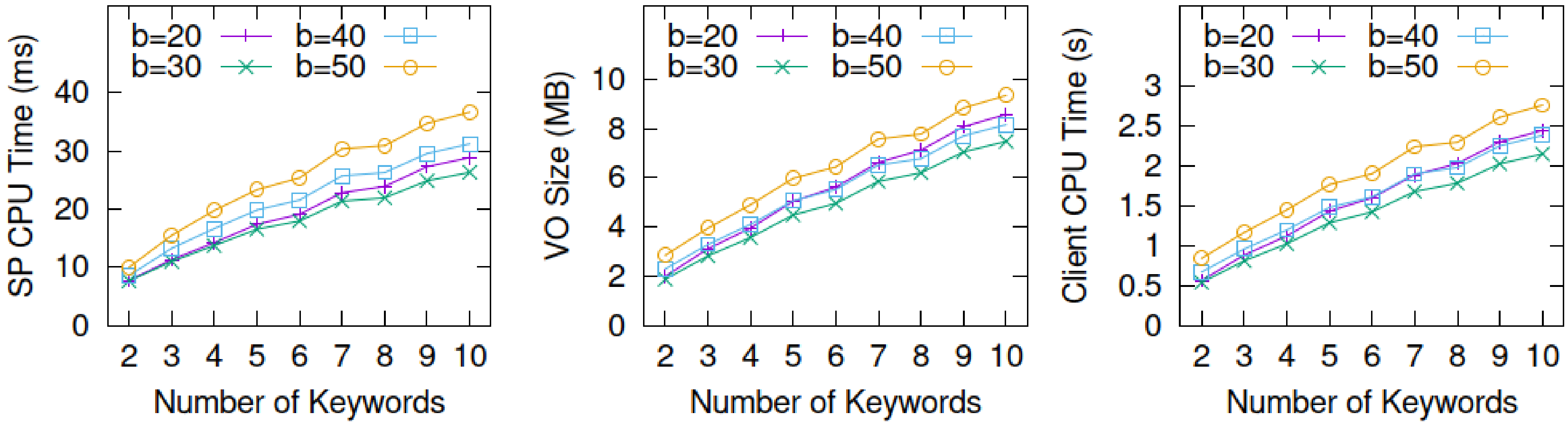
Authenticated Query Performance



Twitter Dataset

- CI^* is more efficient owing to its use of Bloom filters
- Verification of CI and CI^* is relatively slow owing to the costly CVC operations

Authenticated Query Performance



Twitter Dataset

- Default setting $b = 30$ yields the best results
- If b is too small, the effectiveness of using Bloom filter to filter the unmatched objects is not obvious
- If b is too large, a high false positive rate makes it less effective

Thanks!
Q&A