Boosting Consensus Efficiency: Parallelism and Blockchain-Enhanced Remote Attestation in Trusted Execution Environments — Supplementary Material —

Ran Wang, *Graduate Student Member, IEEE,* Cheng Xu, *Member, IEEE,* Sisui Tang, Hangning Zhang, Xiaotong Zhang, *Senior Member, IEEE*

1 Proof of Correctness

Our protocol upholds the safety and liveness attributes inherent to conventional Byzantine Fault Tolerance (BFT) protocols. Within this context, safety ensures that all benign nodes process identical requests in a consistent sequence, whereas liveness guarantees that requests from benign clients are invariably executed. The demonstration of the correctness of TEP-BFT is elaborated upon in this section.

1.1 Safety

Lemma 1. *In the same view v, if a benign node executes an operation op with the identifier $UI.h$, no other benign node will execute this operation with a different identifier $UI.h'$ where $UI.h' \neq UI.h$.*

Proof. $UI.h$ is an identifier assigned by the primary node using the *createUI* function. If a benign node executed an operation op with identifier $UI.h$, it must have accepted $f+1$ valid *Commit* messages for $\langle op, UI.h\rangle$. Let these $f+1$ nodes be denoted as N_s.

By proof of contradiction, assume there is another node S' that executes operation op with identifier $UI.h'$ where $UI.h' > UI.h$. According to TEP-BFT, S' would have accepted $f+1$ valid *Commit* messages for $\langle op, UI.h'\rangle$. Let these $f+1$ nodes be denoted as N_s'. Since $n = 2f+1$ and $|N_s| + |N_s'| = 2f+2 > n$, there must be at least one node S_l (the intersecting node), that sent *Commit* messages for both $\langle op, UI.h\rangle$ and $\langle op, UI.h'\rangle$. Therefore, it can be concluded that S_l is a Byzantine node. S_l could be the current primary node S_p or a replica node Sr_i. We need to consider the following cases:

1) ***When S_l is the primary node*** S_p: S_p generates two UIs for the same operation op, i.e., $UI = \langle UI.h, H(op)\rangle$ and $UI' = \langle UI.h', H(op)\rangle$. At the same time, it is possible that Byzantine replica nodes also accepted UI and UI', sending *Commit* messages with $\langle UI, UI_c\rangle$ and $\langle UI', UI_c'\rangle$ to some benign node. Now, the execution of the benign node S_c can be divided into the following two cases:

- $\underline{S_c \text{ has executed } UI.h.}$ At this time, $V_{seq}[Client] = op.seq$. Then S_c will not accept the *Commit* message with $\langle UI', UI_c'\rangle$, because it contains the same request as $op.seq$. S_c will only accept those *Commit* messages where $op'.seq > V_{seq}[Client]$.
- $\underline{S_c \text{ has not executed } UI.h \text{ yet.}}$ S_c must execute all requests before $UI.h'$ (gaps between identifiers are not allowed). Hence, $UI.h$ must be executed before $UI.h'$.

2) ***When the primary node S_p is benign and S_l is a replica node***: S_p will not generate two different UIs for the same operation op. If S_l sends two different *Commit* messages, i.e., $\langle op, UI.h\rangle$ and $\langle op, UI.h'\rangle$, the other benign nodes, upon validating UI and UI' through *VerifyUI*, will only accept the *Commit* message with the identifier $UI.h$.

To conclude, it is known that a benign node will not execute the same request operation with different identifiers.

This work is supported in part by the National Key Research and Development Program of China under Grant 2021YFB3702403, in part by Shandong Provincial Natural Science Foundation under Grant ZR2022LZH013, in part by the National Natural Science Foundation of China under Grant 62101029, in part by Guangdong Basic and Applied Basic Research Foundation under Grant 2023A1515140071, and in part by the China Scholarship Council Award under Grant 202006465043 and 202306460078. (Corresponding authors: Cheng Xu and Xiaotong Zhang)

- *Ran Wang is with School of Computer and Communication Engineering, Shunde Innovation School, University of Science and Technology Beijing. She is also with School of Computer Science and Engineering, Nanyang Technological University (email: wangran423@foxmail.com).*
- *Cheng Xu is with School of Computer and Communication Engineering, Shunde Innovation School, University of Science and Technology Beijing. He is also with School of Electrical and Electronic Engineering, Nanyang Technological University (email: xucheng@ustb.edu.cn).*
- *Sisui Tang, Hangning Zhang and Xiaotong Zhang are with School of Computer and Communication Engineering, Shunde Innovation School, University of Science and Technology Beijing (email: tangsisui@163.com; hangning0117@163.com; zxt@ies.ustb.edu.cn;).*

Digital Object Identifier 10.1109/IEEE.2024.XXXXXXX

□

Lemma 2. *In view v, if a benign node executes an operation op with identifier $UI.h$, then in any view v' $>$ v, no other benign nodes will execute this operation with a different sequence number $UI.h'$ where $UI.h' \neq UI.h$.*

Proof. Although *v'* can be any value, any view *v"* between *v* and *v'*, can be considered as multiple iterations of *v'* $=$ *v* $+ 1$, regardless of whether any requests were executed in *v"*. Hence, we only need to discuss the case where *v'* $=$ *v* $+ 1$.

Lemma 2 is also proved by contradiction. If a benign node S_c in view *v* executes an operation with identifier $UI.h$, it must receive $f + 1$ valid *Commit* messages for $\langle op, UI.h, v\rangle$, denoted as N_s. Suppose there is another benign node S_c', which in view *v'* executes the operation *op* with identifier $UI.h'$ where $UI.h' > UI.h$. According to the TEP-BFT protocol, S_c' receives $f + 1$ valid *Commit* messages for $\langle op, UI.h', v'\rangle$, denoted as N_s'. Because $n = 2f + 1$ and $|N_s| + |N_s'| = 2f + 2 > n$, there must be at least one intersecting Byzantine node S_l that sent two different *Commit* messages, namely $\langle op, UI.h, v\rangle$ and $\langle op, UI.h', v'\rangle$.

Firstly, we must prove that the new primary node S_p of view *v'* must acknowledge that *op* was accepted or executed in view *v*. This is proved through the new view certificate V_{nv}, which contains $f + 1$ *Viewchange* messages $\langle \text{Viewchange}, S_i, v', cP_{latest}, M, UI_i\rangle$, from $f + 1$ nodes, denoted as N_s'', with at least one benign node S_c among them. The *Viewchange* message from S_c is $\langle \text{Viewchange}, Sc, v', cP_{latest}, M, UI_c\rangle$. Also, Byzantine node S_l would have sent a *Viewchange* message before sending $\langle op, UI.h', v'\rangle$. Now we need to consider the following two cases:

1) ***If** op **was executed after the most recent stable checkpoint***: *op*'s *Commit* message will be included in M of Viewchange. However, Byzantine S_l may not include *op*'s *Commit* message in M. In this case, we further consider the following scenarios:

- If S_p is benign: If $S_c \in N_s''$ executed operation *op* in view *v*, then S_c's *Viewchange* message's M contains the *Commit* message for operation *op*. Hence, through this *Viewchange* message, S_p can determine that *op* has been executed. If S_c did not execute *op* in view *v*, S_l would have to perform one of two actions detectable by S_p to exclude *op* from M: 1) If S_l executed a request op' after *op*, S_l could include op''s *Commit* message in M but not *op*'s, leaving a gap in M detectable by S_p; 2) If S_l sent a *Commit* message with $\langle UI.h, op\rangle$, it might not include any *Commit* messages with $UI.h' > UI.h$ in M, but S_p will detect this because S_l must sign the *Viewchange* message with $UI.h'$. Therefore, for S_l's *Viewchange* message to be inserted into V_{nv} by S_p, S_l must include *op*'s *Commit* message in M.
- If S_p is Byzantine: S_p may attempt to modify the M it inserts into V_{nv}. If it merely removes *op* from M, it leaves a gap that is detectable. If it removes *op* and all subsequent messages, this is also detectable because S_p cannot forge a UI from S_c with a counter value higher than the later messages. If S_p inserts S_l's *Viewchange* message into V_{nv}, however, benign nodes will verify the validity of V_{nv} upon receiving the *newView* message from S_p. Therefore, Byzantine S_p cannot forcibly add S_l's *Viewchange* message to V_{nv}.

2) ***If** op **was executed before the most recent stable checkpoint***: The execution of *op* is implicit in the certificate of the most recent stable checkpoint. Byzantine S_l may attempt to place an older checkpoint in the *Viewchange* message. In this case, we further consider the following scenarios:

- If S_p is benign: If $S_c \in N_s''$ executed operation *op* in view *v*, since S_c is benign, S_c's Viewchange message's cP_{latest} already includes the fact that *op* was executed. Hence, S_p can determine that *op* has been executed. At the same time, S_p will not insert the *Viewchange* message sent by S_l into the new view certificate V_{nv}, because by comparing the hash value $cPHash$ and cP_{latest} of the current node state in the *Viewchange* message of S_c and S_l, it will be found that S_l does not include the execution of *op* in the checkpoint. If S_c did not execute *op* in view *v*, S_p will never insert S_l's Viewchange message into V_{nv}, because S_l must perform one of the detectable actions indicated in the condition 1)-1.
- If S_p is Byzantine: If S_p attempts to replace cP_{latest} with an older checkpoint certificate, it also cannot forge the UI. Even if it uses an older checkpoint sent by S_c, this is also detectable. This is because when a benign node receives a *newView* message, it checks the validity of V_{nv}. Therefore, Byzantine S_p cannot tamper with the contents of benign nodes' *Viewchange* messages. If S_p inserts S_l's *Viewchange* message into V_{nv}, however, benign nodes will verify the validity of V_{nv} upon receiving the *newView* message from S_p. Therefore, Byzantine S_p cannot forcibly add S_l's *Viewchange* message to V_{nv}.

Through the above deductions, it is shown that the new primary node S_p of view *v'* must acknowledge that *op* was accepted or executed before *v'*. The following will continue to prove that benign nodes will not execute *op* in view *v'*, where the sequence number $UI.h'$ in view *v'* is different from $UI.h$. Consider the following two cases:

- If S_p is benign: As proved above, S_p can confirm that *op* has been executed, hence a benign S_p will not generate another UI for the same *op* in view *v'* and send a *Commit* message. Therefore, other benign nodes will also not execute *op* in view *v'* that has already been done in *v*.
- If S_p is Byzantine: S_p can create a new *Prepare* message containing $UI' = \langle UI.h', H(op)\rangle$ and send it to other nodes, where benign replica nodes will validate *op.seq* and find *op.seq* $\leq V_{seq}[Client]$, meaning the request has already been executed, thus benign nodes will not execute it again.

□

1.2 Liveness

Lemma 3. *During a stable view, operations requested by benign clients will be completed.*

Proof. We define a stable view as one where the primary node is benign and there are no timeouts on benign replica nodes. If the *client* is benign, it will send operation *op* with a sequence number seq greater than any previously used to all nodes. Since the primary node S_p is benign in a stable view, it will generate a $UI = \langle UI.h, H(op)\rangle$ and send a *Prepare* message with UI to all other replica nodes. Benign replica nodes, after receiving this message, will call *verifyUI* to validate the UI and send a *Commit* message for $\langle UI.h, op\rangle$. Since there are at most f Byzantine nodes in the system, at least $f+1$ benign nodes (including S_p and other f replica nodes) will generate these *Commit* messages and send them to all others. When a benign node receives $f+1$ *Commit* messages, it will execute *op* and send a *Reply* message to the client Client. Once Client receives $f+1$ matching *Reply* messages, the operation will be considered complete. Because there are $f+1$ benign nodes, the above will inevitably happen, and they will work the same as the $UI.h$-th operation when executing *op*. □

Lemma 4. *If at least $f+1$ benign nodes request a view change, then view v will eventually be changed to a new view v' > v.*

Proof. To request a view change, a benign node S_c sends a $\langle \text{ViewchangeReq}, S_c, v, v'\rangle$ message to all nodes, where *v* is the current view number and $v' = v+1$ is the new. Consider a view change from *v* to $v+1$ requested by a set of $f+1$ benign nodes N_s. By definition, the primary node S_p in view *v* faces two conditions:

1) ***The view is stable***: This means that all nodes in N_s have received *ViewchangeReq* messages from each other. When one node S receives the $f+1$-th *ViewchangeReq* message, it sends a $\langle \text{Viewchange}, S, v', cP_{latest}, M, UI_{vs}\rangle$ message to all others. All *Viewchange* messages sent by nodes in N_s are received by all other nodes. The primary node S'_p of view *v'* is benign, so it sends a $\langle \text{newView}, S'_p, v', V_{nv}, NV_c, UI_n\rangle$ message to all others. Since the view is stable, all nodes receive the *newView* message, and the view changes to *v'*.

2) ***The view is unstable***: The following two cases should be taken into consideration:

- *S_p is Byzantine, but it does not send a newView message or sends an invalid newView message discarded by all benign nodes; or S_p is benign, but communication delays cause all benign nodes' timeouts to expire*: When nodes send *Viewchange* messages, they start a timer that expires after a fixed time unit T_{nc}. In this case, all benign nodes' timers will expire, and they will initiate another view change.
- *S_p is Byzantine, but sends the newView message to at least $f+1$ nodes (denoted as N'_s) among which less than $f+1$ are benign nodes; or S_p is benign, but communication delays lead to the same effect*: In this case, the Byzantine nodes in N'_s can act according to the protocol, making the benign nodes in N'_s believe it is running correctly. The nodes in N'_s can send *Prepare* and *Commit* messages following the normal operation process. For the benign nodes not in N'_s, their timers will expire after a fixed time unit T_{nc}, and these benign nodes will send *ViewchangeReq* messages, but there will not be $f+1$ such messages, so no view change will occur. When Byzantine nodes begin deviating from the normal operation process, the requests will stop being accepted, and the benign nodes in N'_s will send *ViewchangeReq* messages, initiating the view change. In both cases, when another view change begins, the system may fall back into either of the conditions 1) or 2). However, eventually, the view will become stable, and the system will fall into condition 1), and the view will change to the new view *v'*.

□

Theorem 5. *Operations requested by benign clients will eventually be completed.*

Proof. This proof is derived from the previous lemmas. In a stable view, operations requested by benign clients will eventually be completed (Lemma 3). If view *v* is unstable, at the expiration of the timer, there exist two conditions:

1) ***At least $f+1$ benign nodes request a view change***: In this case, the view will change to a new *v'* (Lemma 4).

2) ***Less than $f+1$ benign nodes request a view change***: This scenario is similar to the situation in Lemma 4-2)-2. If there is at least a subset of $f+1$ nodes N'_s that do not request a view change and continue to operate in view *v*, the system will remain in view *v*, and requests from benign clients will be executed. If there is no such N'_s or requests are not executed within a fixed time, all benign nodes will request a view change, leading to condition 1).

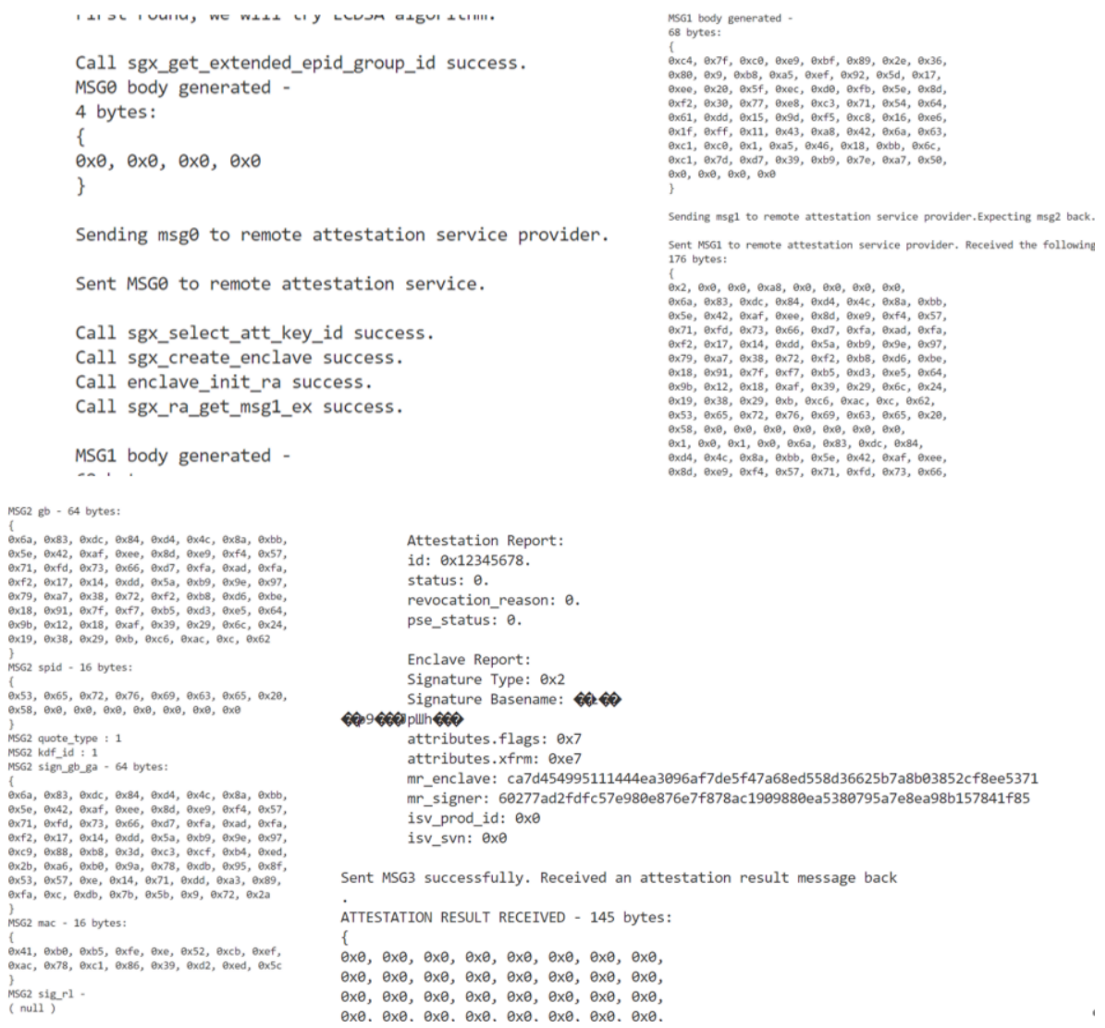
Condition 1) will lead to a view change, but the new view *v'* may not be stable. The system model assumes that processing and communication delays will not grow indefinitely, and in the protocol, the fixed time T_{nc} doubles each time a new view change is needed. Therefore, even if view changes occur consecutively, eventually there will be a view *v"*, and one of the following two scenarios will happen:

- S_p is benign: There are no timeouts expiring on benign replica nodes because T_{nc} is greater than the observed maximum delay. In this case, the view is stable, and operations are executed through Lemma 3.
- S_p is Byzantine: In this case, S_p can deviate from the normal operation process leading to timeouts and new view changes, or follow the normal operation process to avoid view changes. In any case, the view is not stable, so we enter the above condition 1) or 2). Eventually, there will be a view where S_p is benign, because only a minority of nodes are Byzantine, and the view will eventually become stable.

□

2 Analysis of Remote Attestation Reports

The remote attestation process based on an authoritative blockchain primarily uses the Sigma protocol to facilitate Diffie-Hellman (DH) key exchange between the client and server. This protocol involves three main steps: protocol initialization, key exchange, and key encryption/decryption. Initially, the service provider and client perform protocol initialization, which includes confirming the protocol version and establishing a secure communication channel. Next, the Sigma protocol is employed for DH key exchange, generating a shared key for encryption and decryption. The service provider encrypts the keys or data to be sent to the client using the shared key, and the client decrypts the encrypted data within the TEE to retrieve the keys or content.



```
First round, we will try ECDSA algorithm.

Call sgx_get_extended_epid_group_id success.
MSG0 body generated -
4 bytes:
{
0x0, 0x0, 0x0, 0x0
}

Sending msg0 to remote attestation service provider.

Sent MSG0 to remote attestation service.

Call sgx_select_att_key_id success.
Call sgx_create_enclave success.
Call enclave_init_ra success.
Call sgx_ra_get_msg1_ex success.

MSG1 body generated -
```

```
MSG1 body generated -
68 bytes:
{
0xc4, 0x7f, 0xc0, 0xe9, 0xbf, 0x89, 0x2e, 0x36,
0x80, 0x9, 0xb8, 0xa5, 0xef, 0x92, 0x5d, 0x17,
0xee, 0x20, 0x5f, 0xec, 0xd0, 0xfb, 0x5e, 0x8d,
0xf2, 0x30, 0x77, 0xe8, 0xc3, 0x71, 0x54, 0x64,
0x61, 0xdd, 0x15, 0x9d, 0xf5, 0xc8, 0x16, 0xe6,
0x1f, 0xff, 0x11, 0x43, 0xa8, 0x42, 0x6a, 0x63,
0xc1, 0xc0, 0x1, 0xa5, 0x46, 0x18, 0xbb, 0x6c,
0xc1, 0x7d, 0xd7, 0x39, 0xb9, 0x7e, 0xa7, 0x50,
0x0, 0x0, 0x0, 0x0
}

Sending msg1 to remote attestation service provider.Expecting msg2 back.

Sent MSG1 to remote attestation service provider. Received the following
176 bytes:
{
0x2, 0x0, 0x0, 0xa8, 0x0, 0x0, 0x0, 0x0,
0x6a, 0x83, 0xdc, 0x84, 0xd4, 0x4c, 0x8a, 0xbb,
0x5e, 0x42, 0xaf, 0xee, 0x8d, 0xe9, 0xf4, 0x57,
0x71, 0xfd, 0x73, 0x66, 0xd7, 0xfa, 0xad, 0xfa,
0xf2, 0x17, 0x14, 0xdd, 0x5a, 0xb9, 0x9e, 0x97,
0x79, 0xa7, 0x38, 0x72, 0xf2, 0xb8, 0xd6, 0xbe,
0x18, 0x91, 0x7f, 0xf7, 0xb5, 0xd3, 0xe5, 0x64,
0x9b, 0x12, 0x18, 0xaf, 0x39, 0x29, 0x6c, 0x24,
0x19, 0x38, 0x29, 0xb, 0xc6, 0xac, 0xc, 0x62,
0x53, 0x65, 0x72, 0x76, 0x69, 0x63, 0x65, 0x20,
0x58, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x1, 0x0, 0x1, 0x0, 0x6a, 0x83, 0xdc, 0x84,
0xd4, 0x4c, 0x8a, 0xbb, 0x5e, 0x42, 0xaf, 0xee,
0x8d, 0xe9, 0xf4, 0x57, 0x71, 0xfd, 0x73, 0x66,
```

```
MSG2 gb - 64 bytes:
{
0x6a, 0x83, 0xdc, 0x84, 0xd4, 0x4c, 0x8a, 0xbb,
0x5e, 0x42, 0xaf, 0xee, 0x8d, 0xe9, 0xf4, 0x57,
0x71, 0xfd, 0x73, 0x66, 0xd7, 0xfa, 0xad, 0xfa,
0xf2, 0x17, 0x14, 0xdd, 0x5a, 0xb9, 0x9e, 0x97,
0x79, 0xa7, 0x38, 0x72, 0xf2, 0xb8, 0xd6, 0xbe,
0x18, 0x91, 0x7f, 0xf7, 0xb5, 0xd3, 0xe5, 0x64,
0x9b, 0x12, 0x18, 0xaf, 0x39, 0x29, 0x6c, 0x24,
0x19, 0x38, 0x29, 0xb, 0xc6, 0xac, 0xc, 0x62
}
MSG2 spid - 16 bytes:
{
0x53, 0x65, 0x72, 0x76, 0x69, 0x63, 0x65, 0x20,
0x58, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0
}
MSG2 quote_type : 1
MSG2 kdf_id : 1
MSG2 sign_gb_ga - 64 bytes:
{
0x6a, 0x83, 0xdc, 0x84, 0xd4, 0x4c, 0x8a, 0xbb,
0x5e, 0x42, 0xaf, 0xee, 0x8d, 0xe9, 0xf4, 0x57,
0x71, 0xfd, 0x73, 0x66, 0xd7, 0xfa, 0xad, 0xfa,
0xf2, 0x17, 0x14, 0xdd, 0x5a, 0xb9, 0x9e, 0x97,
0xc9, 0x88, 0xb8, 0x3d, 0xc3, 0xcf, 0xb4, 0xed,
0x2b, 0xa6, 0xb0, 0x9a, 0x78, 0xdb, 0x95, 0x8f,
0x53, 0x57, 0xe, 0x14, 0x71, 0xdd, 0xa3, 0x89,
0xfa, 0xc, 0xdb, 0x7b, 0x5b, 0x9, 0x72, 0x2a
}
MSG2 mac - 16 bytes:
{
0x41, 0xb0, 0xb5, 0xfe, 0xe, 0x52, 0xcb, 0xef,
0xac, 0x78, 0xc1, 0x86, 0x39, 0xd2, 0xed, 0x5c
}
MSG2 sig_rl -
( null )
```

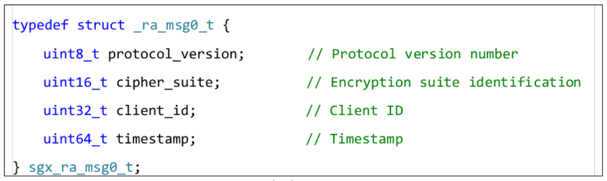
```
          Attestation Report:
          id: 0x12345678.
          status: 0.
          revocation_reason: 0.
          pse_status: 0.

          Enclave Report:
          Signature Type: 0x2
          Signature Basename: ��.��
��9���pШh���
          attributes.flags: 0x7
          attributes.xfrm: 0xe7
          mr_enclave: ca7d454995111444ea3096af7de5f47a68ed558d36625b7a8b03852cf8ee5371
          mr_signer: 60277ad2fdfc57e980e876e7f878ac1909880ea5380795a7e8ea98b157841f85
          isv_prod_id: 0x0
          isv_svn: 0x0

Sent MSG3 successfully. Received an attestation result message back
.
ATTESTATION RESULT RECEIVED - 145 bytes:
{
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0. 0x0. 0x0. 0x0. 0x0. 0x0. 0x0. 0x0.
```

Figure 1: Remote Attestation Report Diagram

As shown in Fig. 3 of the main text, the consensus node acts as the TEE node in the traditional remote attestation scheme, while the attestation node in the authoritative blockchain serves as the service provider. The resulting remote attestation report generated by the system is shown in Fig. 1. The remote attestation report is designed with several message structures, each representing different meanings, such as MSG0, MSG1, MSG2, etc.

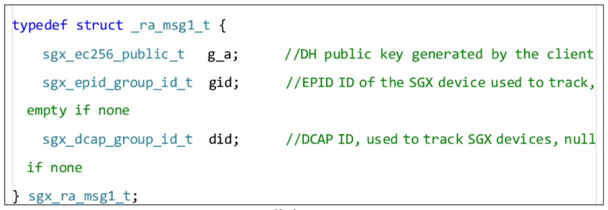
1) **MSG0:** MSG0 is an initial handshake message sent by the consensus node to the attestation node. The structure of MSG0 is shown in Fig. 2-(a). This message establishes the communication foundation, including protocol version, supported encryption algorithms, and other initialization information.
2) **MSG1:** The consensus node constructs the MSG1 object using the DH key exchange protocol to initiate the key negotiation process and sends it to the attestation node. The structure of MSG1 is shown in Fig. 2-(b).
3) **MSG2:** The attestation node constructs its DH parameters, packages its public key and SigRL information, generates a shared key, and responds to the MSG1 request by sending MSG2 to the client. The structure of MSG2 is shown in Fig. 2-(c). This step completes the key negotiation, allowing both parties to share a secure key for subsequent communication.
4) **MSG3:** The consensus node verifies the signature, generates a report, signs it with the shared key, and returns the MSG3 message. The structure of MSG3 is shown in Fig. 2-(d). This step involves signing the report to ensure its integrity and authenticity before returning it to the other party.

5) **Quote Report:** The report, as detailed in Fig. 3, includes information such as the version of the signature report, signature type, EPID group ID, Quote Enclave version, PCE version, basename, report body, signature length, and signature data. The consensus node combines these fields to form a complete signature report for verifying the enclave's security, the application's trustworthiness, and the report's integrity and authenticity.
6) **MSG4:** The attestation node verifies the provided attestation evidence, generates MSG4, and sends it to the client. The service provider submits the reference to the IAS (Intel Attestation Service) to obtain an attestation report, verifies the signature, processes the report, and generates a corresponding message for the client. The format of MSG4 is shown in Fig. 2-(e). Although dependent on the attestation node, it contains information about whether the enclave and PSE (Platform Service Enclave) manifest are trusted.



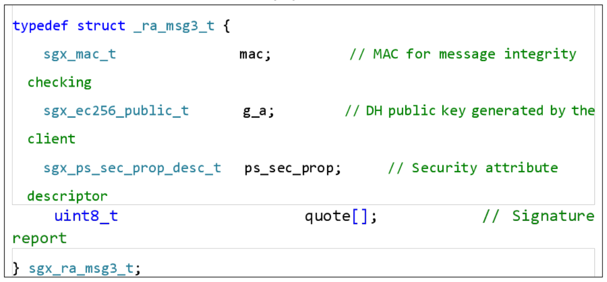
```
typedef struct _ra_msg0_t {
    uint8_t protocol_version;        // Protocol version number
    uint16_t cipher_suite;           // Encryption suite identification
    uint32_t client_id;              // Client ID
    uint64_t timestamp;              // Timestamp
} sgx_ra_msg0_t;
```

(a)



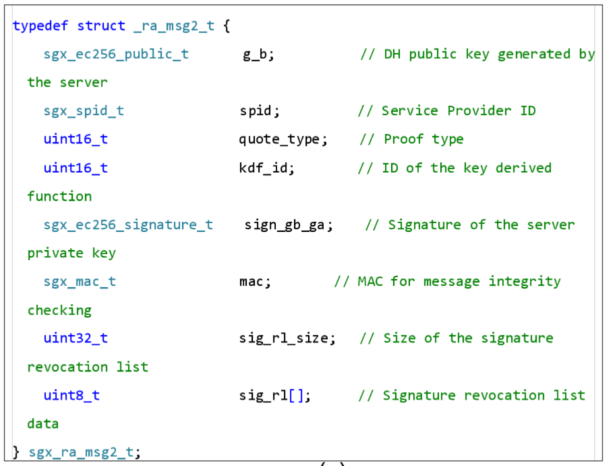
```
typedef struct _ra_msg1_t {
    sgx_ec256_public_t   g_a;      //DH public key generated by the client
    sgx_epid_group_id_t  gid;      //EPID ID of the SGX device used to track,
  empty if none
    sgx_dcap_group_id_t  did;      //DCAP ID, used to track SGX devices, null
  if none
} sgx_ra_msg1_t;
```

(b)



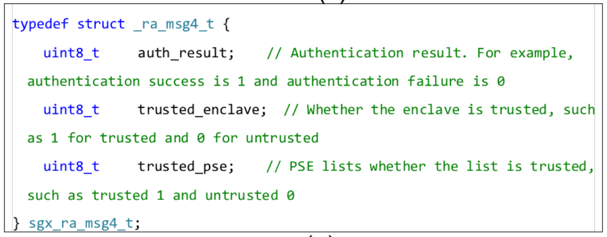
```
typedef struct _ra_msg3_t {
    sgx_mac_t               mac;         // MAC for message integrity
  checking
    sgx_ec256_public_t      g_a;         // DH public key generated by the
  client
    sgx_ps_sec_prop_desc_t  ps_sec_prop;     // Security attribute
  descriptor
      uint8_t                     quote[];            // Signature
report
} sgx_ra_msg3_t;
```

(d)



```
typedef struct _ra_msg2_t {
    sgx_ec256_public_t      g_b;          // DH public key generated by
  the server
    sgx_spid_t              spid;         // Service Provider ID
    uint16_t                quote_type;   // Proof type
    uint16_t                kdf_id;       // ID of the key derived
  function
    sgx_ec256_signature_t   sign_gb_ga;   // Signature of the server
  private key
    sgx_mac_t               mac;       // MAC for message integrity
  checking
    uint32_t                sig_rl_size;  // Size of the signature
  revocation list
    uint8_t                 sig_rl[];     // Signature revocation list
  data
} sgx_ra_msg2_t;
```

(c)



```
typedef struct _ra_msg4_t {
    uint8_t    auth_result;    // Authentication result. For example,
  authentication success is 1 and authentication failure is 0
    uint8_t    trusted_enclave;  // Whether the enclave is trusted, such
  as 1 for trusted and 0 for untrusted
    uint8_t    trusted_pse;    // PSE lists whether the list is trusted,
  such as trusted 1 and untrusted 0
} sgx_ra_msg4_t;
```

(e)

Figure 2: The MSG structure. (a)-(e) represents MSG0 to MSG4,respectively.

7) **Token:** The authoritative blockchain generates a time-sensitive token based on the trusted enclave information received, creating a token for each consensus phase based on the node ID. This token is used for quick authentication during the consensus phase and is included in every message sent. The token structure is shown in Fig. 4. Using the consensus phase and token status, it determines whether a new token is generated and uses the node cluster ID for quick authentication.

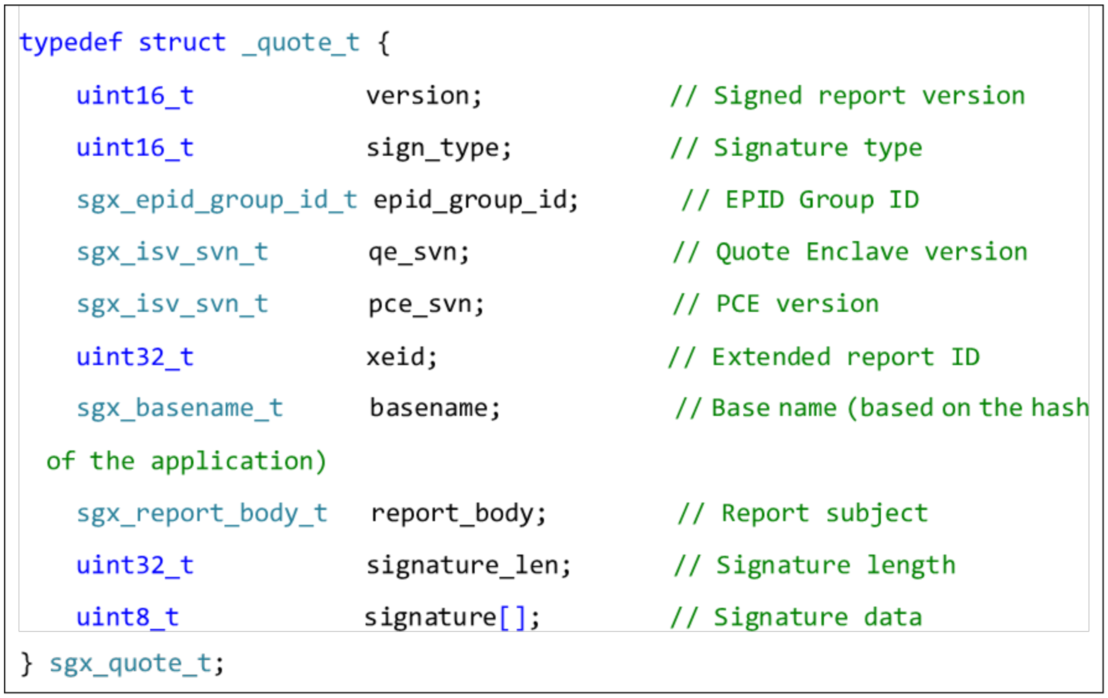
By combining the above processes and message structures, we constructs the basic flow of the remote attestation protocol. Through key exchange, signature verification, and report processing steps, it ensures the security and trustworthiness of communications, allowing consensus nodes to join the blockchain network and conduct periodic communications with quick remote attestation. This blockchain-based remote attestation scheme allows consensus nodes to query the authoritative blockchain for attestation and token information to establish secure channels. It enhances the security of the enclave, the trustworthiness of the application, and the integrity and authenticity of reports. By implementing decentralized remote attestation with an authoritative blockchain, the process is traceable, tamper-proof, and verifiable, providing a foundation of trust for the TEP-BFT consensus algorithm mentioned earlier.

3 Performance Evaluation

3.1 How QPS impacts on CPU and memory utilization

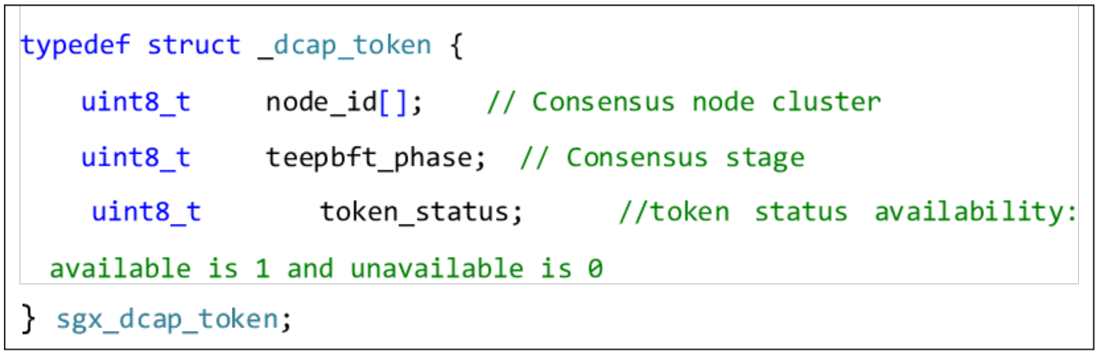
This case study evaluates the impact of varying Query Per Second (QPS) rates on CPU load. A comparative analysis of CPU core counts was conducted using industrial servers equipped with 4/8/16/32-core CPU configurations under a load of 10,000 QPS. The results are as follows:

The impacts on CPU workload under different QPS: Fig. 5-(a) illustrates that, before protocol optimization, the average load on the primary node consistently exceeds 4 under all QPS levels within one minute, indicating prolonged resource contention and resulting in extended response times. At lower QPS levels, replica nodes maintain a load below 4, but surpass this threshold as QPS increases. The CPU load utilization exceeds 100%, operating beyond capacity and risking system failure at any moment. Following protocol optimization, the load across all QPS levels diminishes, particularly at higher QPS where the optimization significantly alleviates system pressure. Despite nearing 100% load rate, there is a marked improvement in performance compared to pre-optimization conditions.



```
typedef struct _quote_t {
    uint16_t           version;            // Signed report version
    uint16_t           sign_type;          // Signature type
    sgx_epid_group_id_t epid_group_id;      // EPID Group ID
    sgx_isv_svn_t      qe_svn;             // Quote Enclave version
    sgx_isv_svn_t      pce_svn;            // PCE version
    uint32_t           xeid;               // Extended report ID
    sgx_basename_t     basename;           // Base name (based on the hash
  of the application)
    sgx_report_body_t  report_body;        // Report subject
    uint32_t           signature_len;      // Signature length
    uint8_t            signature[];        // Signature data
} sgx_quote_t;
```

Figure 3: The Quote report.



```
typedef struct _dcap_token {
    uint8_t     node_id[];     // Consensus node cluster
    uint8_t     teepbft_phase;  // Consensus stage
     uint8_t       token_status;      //token  status  availability:
 available is 1 and unavailable is 0
} sgx_dcap_token;
```

Figure 4: The Token structure.

These observations underscore the positive effect of protocol optimization on system performance, particularly under high QPS conditions. This enhancement reduces the system's load rate without additional hardware resources, thereby augmenting the system's capability to manage high volumes of concurrent requests. The achievements in improving system throughput and ensuring stability and responsiveness underscore the success of our optimization strategy in enhancing resource utilization efficiency and system robustness.

The impacts on CPU workload under different CPU core counts: As depicted in Fig. 5-(b), data indicate that with an increasing number of CPU cores, average CPU utilization rates for the unoptimized PBFT protocol are consistently high across smaller core counts but decrease with larger configurations: 100%, 100%, 49.7%, and 33.7% respectively. In contrast, the TEP-BFT protocol demonstrates lower average CPU utilizations of 99.6%, 93.1%, 44%, and 28.9% respectively. These figures reveal that an increase in CPU core count effectively reduces CPU utilization and enhances protocol performance, with TEP-BFT consistently showing lower CPU utilization across various core configurations.

Based on these experimental results, it is evident that in an SGX environment, the consensus efficiency of nodes is profoundly influenced by the CPU performance. Experiments on 4-core CPU demonstrate significant system strain under all measured metrics—TPS, system latency, and system load. With enhancements in CPU performance moving from 4-core to 8-core and then to 16-core CPU, as illustrated in Fig. 5-(a) and (b), the system's TPS continuously improves while the system load rate declines, indicating an enhanced capacity to manage trusted and untrusted area messages under high pressure. Notably, under similar conditions, TEP-BFT exhibits significant performance improvements compared to traditional PBFT, achieving efficient consensus even on less capable hardware.

The impacts on Memory under different CPU core counts: The experimental setup involved configuring Gramine to allocate 8GB of memory to each trusted container, analyzing system memory usage and its potential impact on performance under different CPU core counts. The results are as follows:

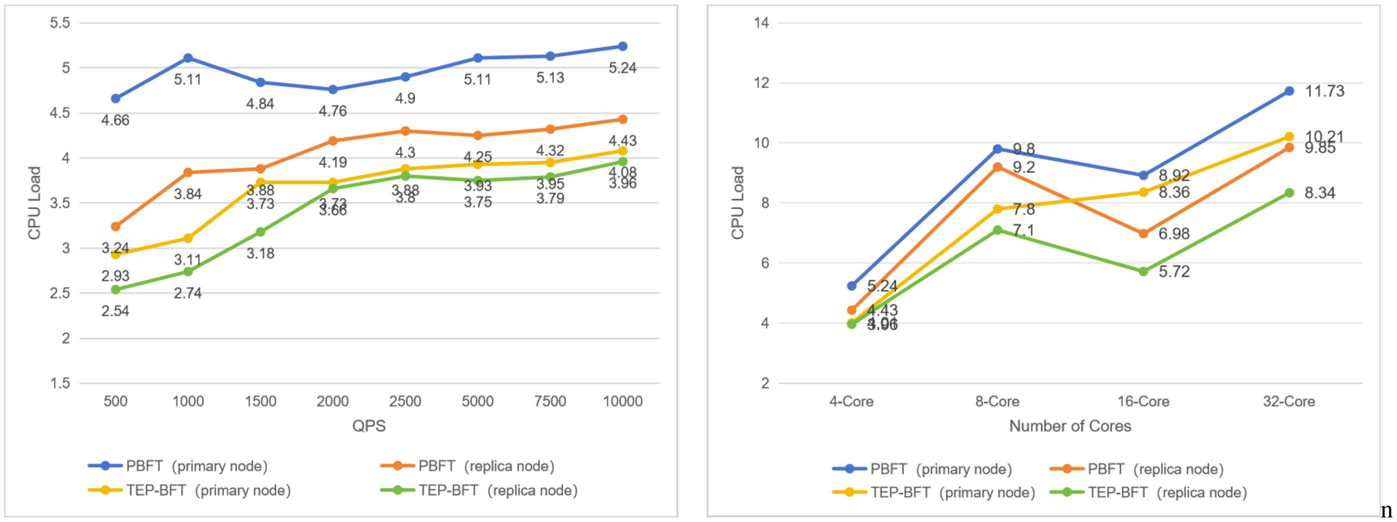


Figure 5: The CPU workload performance comparison of the consensus protocol w/wo optimization. (a) Performance on a 4-core CPUs under various QPS. (b) Performance on various CPU core counts under 10,000 QPS.

Memory Usage: Fig. 6-(a) shows that as QPS increases, operating system memory usage rises while user memory remains stable, a phenomenon observed in 8-core and 16-core CPU configurations (Fig. 6-(b) and (c)). This indicates that higher QPS affects the utilization of trusted memory under the same Gramine settings. However, compared to CPUs, the direct impact of trusted memory on performance appears more limited.

Enclave Memory Overhead: In the SGX protected execution environment (Enclave), memory overhead is relatively small, primarily encompassing code, data, and heap. While direct observations of trusted memory changes are not feasible, analysis of user space memory and operating system memory variations can infer the impact of trusted memory. SGX has minimal effect on user space memory, but kernel space memory increases with QPS.

In conclusion, although Enclave memory usage is relatively small in an SGX environment, an increase in QPS leads to higher operating system memory usage. This indirectly indicates that the utilization rate of trusted memory is influenced by QPS, although its direct impact on performance may not be substantial. Additionally, the rise in Enclave memory overhead is mainly due to the complexity of code, the volume of data structures and buffers required, and the storage of security-related information. Thus, optimizing code and data management within the Enclave, along with appropriate Gramine configuration settings, is essential for controlling memory usage and enhancing system performance.

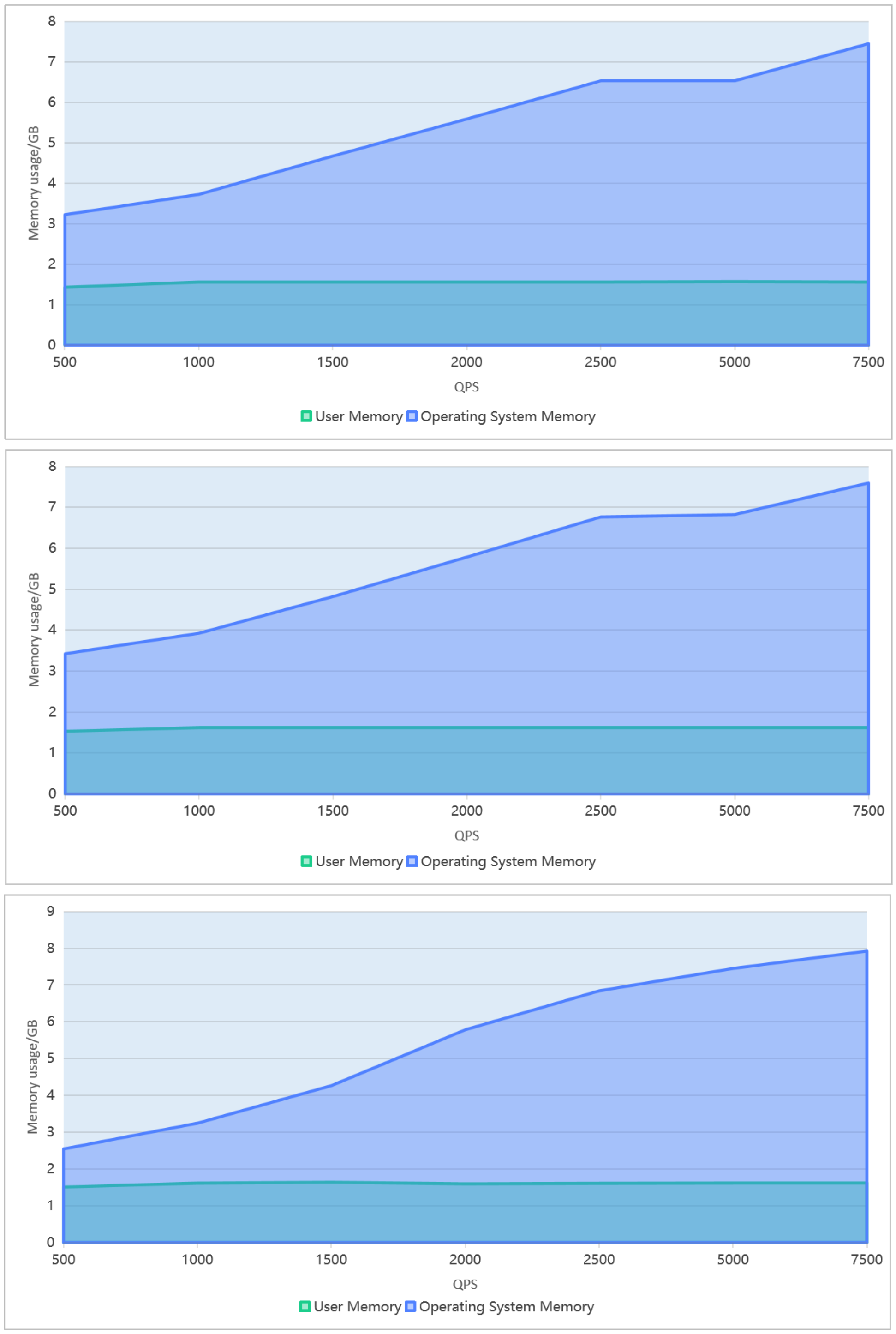


Figure 6: The comparison of operating system memory usage under (a) 4-Core CPU, (b) 8-Core CPU, and (c) 16-Core CPU.