# Gemer: A Decentralized Parallel Application Programming Framework

**WORKING DRAFT**

**gemer.io**

Daniel SH Hong
me@unifiedh.com

**Abstract.** In this paper, we propose Gemer, a parallel application programming framework for decentralized applications. While not a blockchain or a DLT by itself, Gemer generates a "local state" unique to each individual function that can be verified by comparing the root state hash of a callstack graph of an application represented as a Patricia tree. Byzantine nodes can be detected by performing a tree search/state hash comparison of the resulting callstack graph within the verification stage. Nodes deemed as Byzantine may be penalized by revoking the device unique key stored on an application-specific keychain. Trustlessness can be established by periodically submitting verified state updates, including updates to the application keychain, to multiple external blockchains specified as a "world state provider". With Gemer, application developers may be able to fully isolate purely functional codebases from limited state processing capabilities of a blockchain, potentially achieving better scalability compared to using blockchain/DLT-based scalability solutions. Commercial application developers may take advantage of this "Protocol as an Application" concept by achieving trustlessness enabled by a blockchain, while still retaining full control over their own application codebase, thus allowing commercial apps and services to operate in a P2P fashion without the need of a trusted third party.

## 1. Introduction

Ethereum, as proposed by Vitalik Buterin (2014) [1], introduced a way to generalize transactions on a distributed ledger as a state transition. This has enabled Turing-complete code to execute on blockchains and distributed ledgers in the form of a Smart Contract, as in contrast with "first-generation" blockchain software based on the design of Bitcoin (2008) [2].

Because Smart Contracts are code that execute on a distributed ledger, which are not controlled by a single, centralized party, execution of a Smart Contract is (i) unstoppable, (ii) fault-tolerant and (iii) publicly verifiable in a trustless fashion. Even though some of these features are required for creating a decentralized application, of which code execution should not be controlled by a single party, not all of them are essential for commercial applications exposed to end users.

While certain use cases, including but not limited to fundraising, automatic contracts and DAOs (Decentralized Autonomous Organizations) do require the absolute trustlessness and public verifiability of a blockchain-based Smart Contract, the scalability issues of a blockchain-based Smart Contract solution makes it unsuitable for most commercial-level applications.

Researchers and developers within the blockchain ecosystem are attempting to solve this problem through parallel expansion of current linear blockchain environments. Faster consensus, off-chain solutions such as state

channels, state compression through new cryptographic technologies like ZK-SNARKs/STARKs [3], and new ledger structures are also being proposed as solutions.

While these innovations in blockchain technology can promise higher transaction throughput, they inevitably add logical complexity to the system. Thus, while a newly proposed system might have improved throughput with parallel expansion, it gets difficult to obtain human-readable data for system admins and users to independently verify, which can lead to centralization issues. Because the FLP Impossibility of Consensus theorem (1985) [4] shows that a computerized consensus system cannot guarantee both safety and liveness, the ability for humans to independently verify data and code execution on a decentralized network is crucial for commercial-level systems.

Distributed ledgers are also difficult to work with for application developers due to its structural rigidity. Even though the blockchain's rigid data structure is key to ensuring its integrity, this often makes it difficult for developers to build on it. For one thing, developers have limited ability to control data and code being recorded, deployed or executed on a blockchain. Even though this may be useful in some cases, the additional efforts required to build on a distributed ledger may result in a poor developer experience, even when abstracted and automated.

What we need is a new solution not based on distributed ledgers that does not solely rely on a fixed "world state" concept, and **enabling stateless, yet verifiable computing.** This new solution must be flexible enough for developers to have confidence and control over their development and deployment environment, while retaining trustlessness and integrity that can be achieved with DLT-based systems. It needs to be lightweight enough to run on mainstream level devices, but also be able to utilize existing blockchain platforms to run tasks that require public verifiability. A solution like this will enable developers to focus on writing decentralized apps that prioritize user experience, while still taking advantage of distributed ledgers and Smart Contract systems for public and trustless tasks.

# 2. A parallel interpretation of world state

Blockchain researchers and developers often use separate blockchains cryptographically tied to its parent in order to improve transaction throughput. This often enables code to run its state transition on a child chain, and only periodically submit cryptographic proof of multiple state transitions to the parent chain. Plasma [5], Sharding [6], Mimblewimble [7][8] and Coda [9] are examples of such scalability approaches.

This scaling approach of processing state means that we can view world state on a root blockchain as a **collection of multiple "world states"** independently co-existing on multiple, different blockchains. We define this "sub-world state" on such a network as **local state**. Thus, we can define world state as a **set of multiple, independent local states.**

$$\sigma_{WORLD} = \{\sigma^{\alpha}_{LOCAL}, \sigma^{\beta}_{LOCAL}, \sigma^{\delta}_{LOCAL}, \dots\}$$

Each local state is also a collection of state transitions occurring on each individual node, which actually mines the transaction. We define state transitions executed on an individual node for a particular Smart Contract as a **local substate**. Therefore, each **local state** can be defined as a **set of multiple, independent local substates**. For instance, local state $\sigma^{\alpha}_{LOCAL}$ for a child blockchain $\alpha$, which includes nodes Alice, Bob and Cindy, may be defined as follows:

$$\sigma^{\alpha}_{LOCAL} = \{\sigma^{\alpha}_{ALICE}, \sigma^{\alpha}_{BOB}, \sigma^{\alpha}_{CINDY}, \dots\}$$

Under a single blockchain, which must share the same state, it is assumed that state transitions for individual nodes must be the same, even if it is not executed or validated locally. Thus the above definition is impossible for a blockchain network under a single world or local state, which should practically be $\sigma^{\alpha}_{LOCAL} = \sigma^{\alpha}_{ALICE} = \sigma^{\alpha}_{BOB} = \sigma^{\alpha}_{CINDY} = \cdots$.

However, if Alice, Bob and Cindy **are all executing different parts of a single application**, the above definition for local state $\sigma^{\alpha}_{LOCAL} = \{\sigma^{\alpha}_{ALICE}, \sigma^{\alpha}_{BOB}, \sigma^{\alpha}_{CINDY}, \dots\}$ **may be interpreted differently**. In such a case, the system for it **should not be a blockchain or a distributed ledger**, as it enforces a single state across all participating nodes.

In order to verify multiple state transitions occurring on different nodes, we need a way to (i) compress all state transitions on multiple nodes into a single value; (ii) share this single value across all nodes running the same application; and (iii) quickly and effectively verify all state transitions on execution nodes only using this verification value. Also, to achieve this concept as a whole, an application must be divided into different parts that can (i) execute on its own, and (ii) generate state transitions as proof of code execution.

# 3. Functions and Applications

Combining the above two definitions of world, local and local-sub states, world state can be represented as:

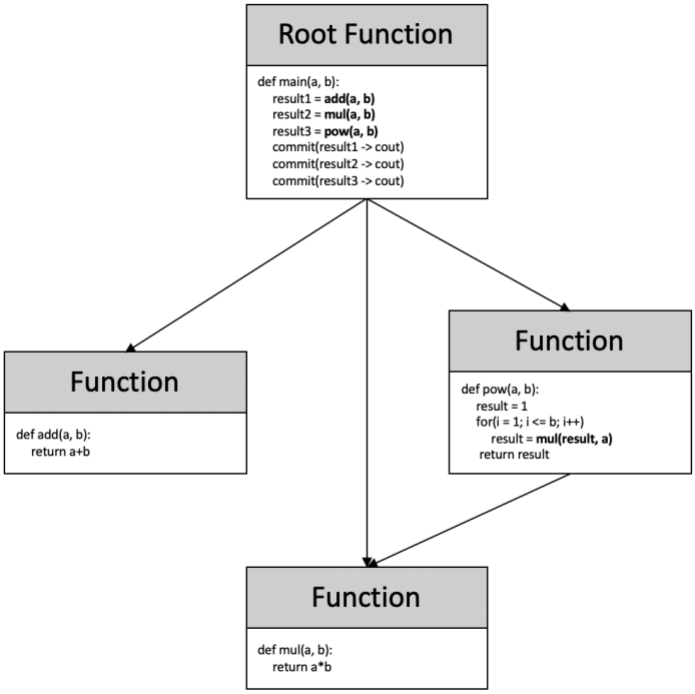
$$\sigma_{WORLD} = \begin{bmatrix} \sigma_A^{\alpha} & \sigma_B^{\alpha} & \sigma_C^{\alpha} & \cdots \\ \sigma_D^{\beta} & \sigma_E^{\beta} & \sigma_F^{\beta} & \cdots \\ \sigma_G^{\gamma} & \sigma_H^{\gamma} & \sigma_I^{\gamma} & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

with each row representing a separate local state.

With Gemer, a **function** is the basic building block for an **application**. Each **application** is defined as a **collection of call-and-return relationships between two or more functions**. Thus, in its raw executable form, Gemer applications are **purely functional**; this means that all operations are considered **stateless**, unless the **root function** initially called by a user **explicitly changes the application state**.

We define each **local substate** as a **cryptographic representation of a function's parameter and return values**. This is required because functions stay stateless unless specified, and an execution result value is required in order to verify a node's honesty.

Thus, a **local state** can be defined as the **combined state of a Gemer application**. In Gemer, local state for an application is represented as the **root of a callstack graph** represented as a **Merkle-Patricia tree**. While this callstack graph is shared by all nodes running the same application, we only need to compute the values that can be linked to a meaningful state transition.



# 4. The Codex & Keychain

To accommodate both the functions consisting an application and its corresponding local substates, we need a new data structure that can be shared across nodes running the app. This application data structure used with Gemer is called the **codex**.

The **codex** is a **cyclic graph that represents the call-and-return relationships between different functions**. Each relationship is cryptographically tied as those of a blockchain. Individual "blocks" store the full bytecode of each function, which are shared across all execution nodes.

The codex also includes a **state chain**, also cryptographically tied to each function code block in a directive fashion. This stores the state transition data generated when each function executes. Only the callstack graph's tree root is stored in the state chain header, and the full tree for each state transition is only stored by (i) the nodes that executed this particular state transition session, and (ii) the nodes that verified it.

The codex accommodates a separate data structure called the **keychain**. The keychain is a **cryptographic chain** of **device-specific public keys** that are currently joining the network. A device without its public key registered on the keychain is not considered to be a valid node on the network. Each state transition and verification stages must be signed by the public key of the device that performed it. The keychain also holds the public key of the entire application, which may be used for cross-application communication workloads.

A group of nodes that share the same codex is called the **application container**. All nodes within the container must have its **public device key registered** on the container's codex keychain. Unlike blockchains and distributed ledgers, a Gemer container is **semi-permissive**; while anyone can **join** an application container, once their public keys are **revoked** as a result of malicious behavior, they are **no longer considered a part of that container**.

The **bare codex**, which only contains raw bytecode without the state chain and keychain, may be packaged alongside with the release application binary distributed for end users. During network initialization, each end device can (i) rely on the application developer's **release keys** used to sign the **application binary**, and (ii) compare its **fingerprint** against other devices within the same container to confirm its integrity.

Bare Codex
prevFunctHash: null
Root Function
def main(a, b):
result1 = add(a, b)
result2 = mul(a, b)
result3 = pow(a, b)
commit(result1 -> cout)
commit(result2 -> cout)
commit(result3 -> cout)
prevFuncHash:
19Mqagdd8QaXAQ27tQ4sRdEKkSQgY
DWbY2
Function
def add(a, b):
return a+b
prevFuncHash:
19Mqagdd8QaXAQ27tQ4sRdEKkSQgY
DWbY2 (main)
prevFuncHash:
1BYJSJZ9WN883wtyhEjhjeBA33unf3awQ (pow)
Function
def mul(a, b):
return a*b
prevFuncHash:
19Mqagdd8QaXAQ27tQ4sRdEKkSQgY
DWbY2
Function
def pow(a, b):
result = 1
for(i = 1; i <= b; i++)
result = mul(result, a)
return result
Sign
deviceKey:
1CDqXRWdhgk
n5acQSA2BUsL
rHTZRnQYPwP
deviceKey:
1Fft6cjqGhN4N
9EhNhrtVrDavi
AjBHa6K
prevFuncHash:
19Mqagdd8QaXAQ27tQ4sRdEKkSQgY
DWbY2
stateHash:
1AS1wMZW6gTE79Noy26GMtJbXA1U
f1Gi7
Committed State:
cout:
3 (invoked by:
1FZ3yg9FEUJ7EPYbr8QoMCFZPIDejTKduD)
2 (invoked by:
1CSRMmECzRdsFVCs89huhnqoDFEANfnicU)
1 (invoked by:
1BYJSJZ9WN8B3wtyhEjhjeBA33unf3awQ)
Params:
input:
a = 1, b = 2
returns:
null
References:
1FZ3yg9FEUJ7EPYbr8QoMCFZPIDejTKduD
1CSRMmECzRdsFVCs89huhnqoDFEANfnicU
1BYJSJZ9WN8B3wtyhEjhjeBA33unf3awQ
prevFuncHash:
1BYJSJZ9WN8B3wtyhEjhjeBA33unf3a
wQ
stateHash:
1DgqUEubETuoKMjmxTdDMF6N2ASU
UKhtmn
Params:
input:
a = 1, b = 2
returns:
1
References:
1CSRMmECzRdsFVCs89huhnqoDFEANfnicU
prevChainHash:
16oG1h1j6NTgMyPznYPmHJATDZjNT4
CdYU
stateHash:
12rsMoCqbRBHDuAPPQRcafAjQdxZf8
BuPo
Committed State:
cout:
7 (invoked by:
1FZ3yg9FEUJ7EPYbr8QoMCFZPIDejTKduD)
12 (invoked by:
1CSRMmECzRdsFVCs89huhnqoDFEANfnicU)
81 (invoked by:
1BYJSJZ9WN8B3wtyhEjhjeBA33unf3awQ)
Params:
input:
a = 3, b = 4
returns:
null
References:
1FZ3yg9FEUJ7EPYbr8QoMCFZPIDejTKduD
1CSRMmECzRdsFVCs89huhnqoDFEANfnicU
1BYJSJZ9WN8B3wtyhEjhjeBA33unf3awQ
prevFuncHash:
1D2wy99rQyvdmxSRzNKBXuQWbcoLro
nZnR
stateHash:
1FPWDoDhkVjzNGQFRVwYy2SF5FL85t
PUQZ
Params:
input:
a = 3, b = 4
returns:
81
References:
1CSRMmECzRdsFVCs89huhnqoDFEANfnicU
State Chain(s)

# 5. Executing and Verifying the Codex

The steps to execute a codex within an application container are as follows:

(i) Randomly **assign** multiple **execution nodes** to each **function**. **VRFs** may be used for node assignment to ensure public verifiability of allocation randomness, as proposed by Algorand [10];

(ii) **Execute** functions & **write** the resulting **local substates** to its corresponding **state chain**;

(iii) **Pass** all resulting **local substates** to its corresponding **child processes** as a **stateless parameter**;

(iv) Cryptographically **sign** all corresponding **execution proofs** tied to each **local substate**, using the **unique public key** of the device (node) that executed the **corresponding function**;

(iv) Repeat (iii) and (iv) for a **randomized portion** of all executing devices deemed **statistically significant** to ensure BFT;

(v) **Expand** the resulting **state chain** from (ii) ~ (iv) into an **AST-style callstack graph**. A callstack graph generated from a state update **must**:

- be a temporary structure generated on a per-request or process basis;
- represent the execution flow cryptographically, involving all state chains involved in a single execution session;
- have its leaf values be a **compressed state representation** of all its parent processes;
- be the form of a **Merkle-Patricia tree**, having its root as a representation of the whole.

(vi) If at least one of the resulting leaf values returned by (v) for all state chains from (iv) does not match, run a **recursive tree search**. Halt at the last execution point that does not have a valid state hash.

- Look up the **device public key** used to sign the last faulty execution point on the **keychain,** and **penalize** the corresponding execution node.
- Penalties may be recorded on the **keychain**, or used to revoke all keys related to that device under certain circumstances.
- Repeat (i) ~ (vi) until the resulting callstack graph meets all the conditions of (vii).

(vii) Else, if all resulting leaf values returned by (v) for all state chains from (iv) match, **commit** all state changes invoked by this session to the **state chain**.

- When a state change is committed to the state chain, devices and/or users involved in that particular execution session may sign committed data with their public device key to approve **finalization** and **irreversibility**.
- Access to permanent storage is achieved through an **externalized state model**, as explained with later sections of this paper.

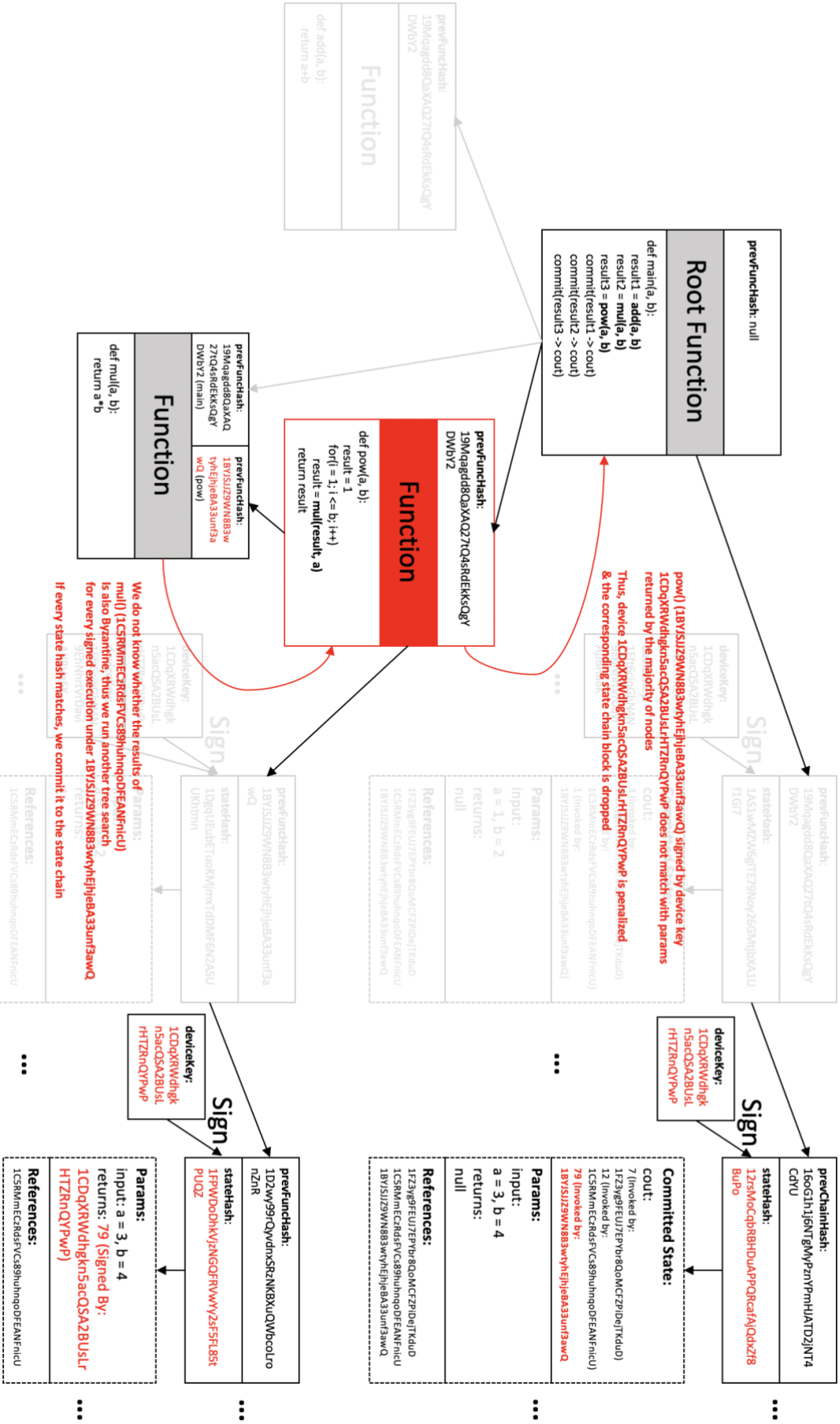
(viii) The cryptographic fingerprint of the resulting **state chain** and the **keychain** is **periodically submitted to an external blockchain**. In the case of Turing-complete platform blockchains, **Smart Contracts** may be used to automate this process on-chain.

(ix) **State Flush & Container Exit**: Under certain circumstances, such as a compromised codex or a state chain overflow, an application may *exit* to another container, thus flushing most prior state chains stored within the codex. In such a case, the application must meet the following *exit conditions*:

- When an application exits to a new container, the signature of the previous application container recorded on the **keychain** and through an **external state provider** must be **revoked**, and be **replaced with the new container's signature.**

- A cryptographic representation of the previous codex container's **state chain** must be recorded as proof while existing to a new container codex. This value should be recorded as a genesis link of the new container's state chain.

- **Statistically no more than 33%** of all participating devices should be against the proposed application container exit.


prevFuncHash: null
Root Function
def main(a, b):
result1 = add(a, b)
result2 = mul(a, b)
result3 = pow(a, b)
commit(result1 -> cout)
commit(result2 -> cout)
commit(result3 -> cout)
prevFuncHash:
19Mqagdd8QaXAQ27tQ4sRdEKkSqBY
DWbY2
Function
def pow(a, b):
result = 1
for(i = 1; i <= b; i++)
result = mul(result, a)
return result
prevFuncHash:
19Mqagdd8QaXAQ27tQ4sRdEKkSqBY
DWbY2 (main)
prevFuncHash:
1BYiSUZ9WN8B3wtyhEjhjeBA33unf3awQ
(pow)
Function
def mul(a, b):
return a*b
We do not know whether the results of mul() (1C5RMmECzRdsFVCs89huhnqoDFEANFnicU) is also Byzantine, thus we run another tree search for every signed execution under 1BYiSUZ9WN8B3wtyhEjhjeBA33unf3awQ
If every state hash matches, we commit it to the state chain
pow() (1BYiSUZ9WN8B3wtyhEjhjeBA33unf3awQ) signed by device key 1CDqXRWdhgkn5acQSA2BUsLrHTZRnQYPwP does not match with params returned by the majority of nodes
Thus, device 1CDqXRWdhgkn5acQSA2BUsLrHTZRnQYPwP is penalized & the corresponding state chain block is dropped
prevChainHash:
16oG1h1j6NTgMyPznYPmHJATD2JNT4CdYU
stateHash:
1ZrsMoCqbRBHDuAPPQRcafAJQdXZf8BuPo
Committed State:
cout:
7 (invoked by:
1FZ3yg9FEUJ7EPYbr8QoMCFZPiDejTKduD
12 (invoked by:
1C5RMmECzRdsFVCs89huhnqoDFEANFnicU)
79 (invoked by:
1BYiSUZ9WN8B3wtyhEjhjeBA33unf3awQ
Params:
input:
a = 3, b = 4
returns:
null
References:
1FZ3yg9FEUJ7EPYbr8QoMCFZPiDejTKduD
1C5RMmECzRdsFVCs89huhnqoDFEANFnicU
1BYiSUZ9WN8B3wtyhEjhjeBA33unf3awQ
deviceKey:
1CDqXRWdhgk
n5acQSA2BUsL
rHTZRnQYPwP
Sign
prevFunchash:
1D2wy99rQyvdnxSRzNKBXuQWbcoLro
nZnR
stateHash:
1FPWDoDhkVjzNGQFRVwWy2sF5FL8St
PUQZ
Params:
input: a = 3, b = 4
returns: 79 (Signed By:
1CDqXRWdhgkn5acQSA2BUsLr
HTZRnQYPwP)
References:
1C5RMmECzRdsFVCs89huhnqoDFEANFnicU

# 6. Web of Trust and Externalized State

The *local state model* we proposed with this paper proposed a new approach to the blockchain's scalability problem, while maintaining asynchronous BFT under particular conditions. However, in order for individual local states to be fully fault tolerant, there are additional factors that should be considered for production-level applications running under an asynchronous environment:

- **The container forking problem.** While fault tolerance may be achieved within an application container with the *codex model* as described in the previous sections, the authenticity of a container in itself cannot be established when communicating between different applications. In such a case, a malicious attacker may be able to conduct a *container forking attack*, in which a remote attacker may fork the entire codex and its corresponding state chain to inject malicious code into its clients.

- **Accessing external permanent storage.** When applications require access to data stored on an external permanent storage service, such as IPFS or centralized cloud services, the trust of such data cannot be established when trustlessness is assumed.

With Gemer, those problems may be largely addressed with two different approaches: **web of trust** and **externalized state**. While some significant issues involving container forking and storage access may be resolved with the web of trust paradigm, an **externalized state model** based on public, decentralized ledgers may be necessary under certain circumstances.

I. **The web of trust (WoT) model**: The concept of WoT was first proposed by Phillip Zimmermann [11] with the release of PGP. As described with the PGP User's Manual, the idea for web of trust is:

*"As time goes on, you will accumulate keys from other people that you may want to designate as trusted introducers. Everyone else will each choose their own trusted introducers. And everyone will gradually accumulate and distribute with their key a collection of certifying signatures from other people, with the expectation that anyone receiving it will trust at least one or two of the signatures. This will cause the emergence of a decentralized fault-tolerant web of confidence for all public keys."* [11]

Thus, the same idea may be applied for a Gemer container. As we already have a **list of all device keys** and a **container-wide signature** within an application container included with its codex, this concept may be used to enable trustlessness for external, trusted data.

Trustless permanent storage access may be achieved with the following process:

(i) **Randomly** selected devices first access data from a given source;
(ii) A **cryptographic representation** of the data blob is calculated, and written to the state chain;
(iii) Each device **signs** the data that it has accessed with its device key;
(iv) **Compare** the cryptographic representation of the signed data. If a signed data blob's hash is different from the one signed by the majority, the device holding the public key that signed it may be **penalized**.

Inter-application data exchanges may be established in a similar fashion. Because each application container holds its own unique container key tied to its codex and keychain, **a well-known container key may be deemed as trustable under the web of trust paradigm.**

The problem with this approach is that it **may not be able to handle container exits effectively**. When the state chain is flushed and the application exits to another container, its public container key changes. This means that the container forking problem will be an issue, as malicious attackers may try to imitate the new container due to the fact that its public key is not yet well known.

**II. The externalized state model**: This approach relies on one or more external blockchains – or "**world state providers**" – to submit the container public key and the container keychain hash periodically as a method to verify a container's integrity and identity. A cryptographic representation of the codex and state chains may also be periodically submitted to a world state provider, when the integrity of application execution is of top priority.

While most blockchain and cryptocurrency networks may be utilized for this purpose, Turing-complete platform chains such as Ethereum may be able to automate this process on-chain. This reduces the additional security risks involved when processing such cryptographic proof submissions off-chain.

# 7. Tokenomics & Gas Fee

Gemer is not a blockchain, nor a distributed ledger network by itself; **it is a software framework that defines common interfaces that independent distributed applications can operate and talk with each other. This is not a single network tied with common state data.** Also, lightweight applications may not make sense as an economic market large enough for a dedicated tokenomics model to make sense at all. In such a case, token-based rewarding will only harm the overall user experience, as there are enough non-financial incentives that can cancel out each other under supply and demand.

Thus, **Gemer does not, and technically cannot, offer a native platform token by itself.** However, **for resource-intensive applications, economic incentives may be required**, as end-user devices often do not have the compute and bandwidth capabilities to process massive computation workloads.

While Gemer does not have its native tokenomics model, individual developers may write additional code to incorporate the concept for this purpose. Consider the contents of this section as a **guideline** rather than an absolute rule to follow; under this proposed model, **application developers are fully responsible for any cryptocurrency and/or tokenomics model being included within their own applications.**

A Gemer application may use any ERC-20 or equivalent standard token contract running on an external blockchain as their own native token for gas fee and rewarding mechanisms. For the best user experience, we recommend using well established cryptocurrencies such as Bitcoin [2] and Ethereum [1], or a trusted stablecoin solution available as a token contract.

To reward computation nodes performing computationally heavy workloads, we may need to incorporate the concept of **gas** in some cases. Gas is calculated on a **per-function basis**; in other words, only function blocks invoked by a single execution session is taken into account when paying for computation gas.

Gas fees are paid for computation nodes "as-is"; this means that tokens should not be minted for the sole purpose of computation rewards. However, when an application needs to accept multiple cryptocurrencies, or fiat payments to maintain the payment user experience, the following solutions may be integrated:

- **Decentralized Exchanges (DEX) as a service.** A decentralized exchange, which can swap various cryptocurrencies into one another, may be used to accept multiple cryptocurrencies within a single app when consistency in currency is required for computation gas and/or other types of payment.

- **Centralized payment gateways.** Because Gemer applications can deal with centralized data and services as described with the previous sections, centralized payment gateways may also be integrated as a payment solution. Developers may directly accept payments through those providers when computation gas is not involved, or use an exchange-as-a-service to change fiat currency paid through centralized payment methods, such as credit cards, into cryptocurrencies in the background.

# 8. Conclusion

In this paper, we proposed Gemer, a parallel decentralized application programming framework for stateless computing. Using this framework, commercial application developers can build fully decentralized applications without being constrained by scalability limitations of a distributed ledger. Developers can also take advantage of decentralized infrastructures, such as user data privacy and self-sovereignty, while being able to easily incorporate, embrace and transform existing centralized infrastructure, user interface and codebases into decentralized applications. We hope this work will help developers to choose decentralized infrastructures over centralized ones as a foundation for their next big thing.

# 9. Future work

- **Efficient way for detecting & penalizing Byzantine nodes**: In the future, newer state compression solutions such as zk-roll-up [12], Mimblewimble [7][8] and Coda [9] may greatly improve Byzantine verification performance in a codex state chain.

- **Translating existing codebases into the codex model**: This requires a new virtual machine and runtime compatibility layer that can compress state on a stack machine and pass it to another, without largely rewriting existing code to be stateless and/or FP-complete. We are planning to discuss potential solutions to this problem in the future.

- **Better randomizable methods for selecting nodes**: While pseudo-random functions may be sufficient for random execution and edge node assignment, adopting VRFs [10] and other latest randomizable methods should greatly decrease the risk of an execution node being predicted in the long term.

- **Precise logic for tokenomics & payment**: The current version of this paper does not directly integrate a tokenomics model into Gemer, and leaves it up for the developers as a guideline. However, in case we cannot control device behavior as intended solely through supply and demand, we may have to integrate a tokenomics and gas payment model directly into the system.

- **Protecting user data**: In the current version of this paper, we did not discuss using zero-knowledge proofs to protect user data while all parameters are recorded on the state chain for fault tolerance. This will be added in a future version of this paper.

# References


[1] Ethereum, “Ethereum”, https://www.ethereum.org.

[2] Satoshi Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System”, https://bitcoin.org/bitcoin.pdf, Oct 2008.

[3] Jens Grothm, “On the Size of Pairing-based Non-interactive Arguments”, https://eprint.iacr.org/2016/260.pdf, 2016.

[4] Michael J. Fischer, Nancy A. Lynch & Michael S. Paterson, “Impossibility of Distributed Consensus with One Faulty Process”, https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf, 1985.

[5] Joseph Poon & Vitalik Buterin, “Plasma: Scalable Autonomous Smart Contracts”, https://plasma.io/plasma.pdf, 2017.

[6] Vitalik Buterin, “Ethereum Sharding FAQ”, https://github.com/ethereum/wiki/wiki/Sharding-FAQ.

[7] Tom Elvis Jedusor, “Mimblewimble”, https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.txt, July 2016.

[8] Andrew Poelstra, “Mimblewimble”, https://download.wpsoftware.net/bitcoin/wizardry/mimblewimble.pdf, Oct 2016.

[9] Izzak Meckler & Evan Shapiro, “Coda: Decentralized cryptocurrency at scale”, https://cdn.codaprotocol.com/v2/static/coda-whitepaper-05-10-2018-0.pdf, May 2018.

[10] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos & Nickolai Zeldovich, “Algorand: Scaling Byzantine Agreements for Cryptocurrencies”, https://people.csail.mit.edu/nickolai/papers/gilad-algorand-eprint.pdf.

[11] Phillip Zimmermann, “PGP User’s Guide, Volume I: Essential Topics”, https://web.pa.msu.edu/reference/pgpdoc1.html, Oct 1994.

[12] Vitalik Buterin, “On-chain scaling to potentially ~500 tx/sec through mass tx validation”, https://ethresear.ch/t/on-chain-scaling-to-potentially-500-tx-sec-through-mass-tx-validation/3477, Sep 2018.