



The Kadena Public Blockchain

Project Summary Whitepaper

Version 1.2 November, 2018

Will Martino
will@kadena.io

Stuart Popejoy
stuart@kadena.io

Introduction

In this paper, we give a high-level overview of the technology behind the Kadena Public Blockchain: the smart contract language, *Pact*, and the new parallel-chain Proof-of-Work architecture, *Chainweb*. Together, these technologies allow for high transaction throughput, as well as smart contract upgrades without a hard fork. This provides the necessary assurance to provide enhanced business workflows and governance mechanisms for the modern age.

Contents

- The Pact smart contract language and its formal verification system
- Built-in governance mechanisms for smart contracts
- Native support for oracles, REST APIs and database integrations
- Chainweb: a new multi-chain Proof-of-Work architecture

Safer smart contracts with Pact

Pact was designed to provide a safe solution for implementing high-value business workflows on a blockchain. Languages like Ethereum's Solidity lack critical features that are part of the day-to-day operations of business applications, such as enforcing business rules with unambiguous error messages on failure, modeling and maintaining database schemas, and authorizing users to perform sensitive operations. However, each of these processes requires advanced expertise. Leaving the design and implementation of these critical features to developers harms productivity, and most importantly, invites bugs and exploits. Pact incorporates these essential features into the language itself, making code easier to write, test, and understand.

Pact follows the philosophy that smart contracts need to be *readable by humans, and verifiable by computers*. We strongly disagree with the usage of bytecode-based interpreters like the EVM, since they require the storage and invocation of large streams of low-level bytecodes that are unintelligible by even advanced users. Instead, Pact is an *interpreted* language, where the code stored within the Blockchain is exactly what the application developers wrote, and is always legible in its original form. This additionally allows Pact to be accessible to both developers and non-developers alike, such that

technically savvy lawyers and business executives can easily review the business logic of their smart contracts.

Pact's focus on safety is inspired by Bitcoin scripts, which were designed with a minimal feature-set to provide the most assurance possible during coin transfers. With this in mind, Pact was purposefully made Turing-incomplete — recursion is not possible in Pact¹, and unbounded looping is not available. However, Pact retains the capability for terminating loops by using familiar list-processing functional concepts, such as map, filter, and fold. We believe that the vast majority of viable blockchain use-cases do not require Turing-complete functionality, while those that genuinely need it (path-finding, complex pricing schemes, etc) are ill-suited for the resource-constrained blockchain environment. Such tasks require an unpredictable amount of compute power. For instance, in Ethereum they could “run out of gas”² and fail to complete at any time.

Formal Verification of Contract Code

Due to the notoriety of failures and exploits that emerged in the Ethereum ecosystem, the risks of automating central business processes with smart contracts have become apparent. *Formal Verification* offers a powerful tool to vastly increase safety and assurance of smart contracts. With Formal Verification, code is transformed into a mathematical model of its functionality, which is then used to prove that properties of that model satisfy certain conditions. This is radically different than normal software testing, as this technology can validate correct behavior over any and all possible inputs and program states. By comparison, usual software development practices (e.g. unit testing) can only test for known situations.

Formal Verification and SMT are highly specialized areas of computer science research that usually require expertise generally beyond that of the average programmer. Pact's verification system leverages Pact's Turing-incomplete design to first assert that the program provably terminates and typechecks, after which it can be directly compiled into the SMT-LIB2 language used by the Z3 theorem prover.

Pact then offers a powerful, succinct “mini-language” that is embedded in the smart contract code itself, identified by a “@model” tag, to express simple *properties* that the Z3 environment will attempt to violate. For example, we can express a balance transfer function that disallows double-spending, by enforcing that total balances debited and

¹ The Ethereum DAO hack is a notable recursion-based exploit.

² “Gas” refers to the payment of cryptocurrency to regulate compute and data usage on Ethereum. Pact contracts will also be regulated by some gas model on the Kadena public blockchain.

credited must sum to zero. This DSL is easy to read, and allows non-technical stakeholders to check verification rules for completeness.³

Smart Contracts with Built-in Governance

Typically, blockchains run on *protocol client software* that must be upgraded to provide protocol-level updates. To achieve this, the blockchain undergoes a *hard fork* wherein everyone in the network is forced to switch over to a new version of the system. Hard forks have the power to change any aspect of the blockchain ledger or protocol; they are only restricted by the informal consensus of community members.

In Ethereum, smart contracts loaded onto the system are referenced by *address* (based on the public key of the uploader) and cannot be upgraded once installed, ever.⁴ (At one point Ethereum was popularizing this notion with the moniker “code is law”). However, in reality, the inability to upgrade smart contracts led to a serious abuse of protocol-level hard forks, in order to fix catastrophic exploits that emerged on the blockchain by overwriting old ledger data. While protocol-only hard forks can be contentious (like Bitcoin’s issues with Segwit and block sizes), hard forks to *modify ledger entries* (where contract code is stored) assert a centralized authority over the ledger data, which is antithetical to blockchain’s trustless, decentralized philosophy.⁵

We assert that any mature smart contract system must support the ability to upgrade contracts *without* requiring a hard fork. This must be allowed at any point in the contract’s life cycle, in order to resolve critical issues and deploy strategic enhancements. Pact requires smart contracts to specify governance, currently using a *keyset* (a rule for allowing public-key signers to authorize operations), and soon a generalized governance function. While keysets offer single- and multiple-key signing operations, governance functions will extend this to offer fully decentralized models, such as stakeholder voting. Pact contract authors will have the freedom to model governance structures however they wish.⁶

³ There is no system available or proposed today that can offer Pact’s combination of formal verification with simple, easy-to-understand code and proof expression. By comparison, Tezos requires developers to write smart contracts in a formally-specified low-level bytecode and construct proofs using the Coq theorem prover.

⁴ Software techniques to introduce indirection in invoking smart contract code can help with this issue, but increase complexity, and still only serve to “work around” the fundamental problem.

⁵ Current approaches to address this issue, like Bancor’s use of a “pilot phase” where bug bounties are made available and addressed, are half-measures: after the pilot phase the contract is made permanently non-upgradeable, requiring a hard fork to address critical issues.

⁶ Pact governance functions operate as a pass-fail on upgrade attempts, perhaps by validating some voting process that has transpired and been recorded in the database.

Oracles and Services

In a robust blockchain service environment, smart contracts need a way to retrieve data from *oracles*; trusted sources in the outside world. In turn, an *oracle process*, such as obtaining a stock price or running an intensive computation, executes off-chain. These processes return data authenticated by proofs of provenance, such as public key signatures. In a *push-based* approach, the oracle source publishes periodic data submissions that can be queried by a contract. In a *pull-based* approach, a smart contract initiates an oracle process by requesting external information.

Existing smart contract languages require custom code to implement pull-based oracle processes — a complicated programming task that is often beyond the skill of an average developer. These interactions can require tracking outstanding requests in the database, handling non-responses, managing escrow payments, or cleaning up after the process is complete.

Pact makes the automation of oracle processes easy and safe with *pacts*, functions that can *yield* and *resume* at distinct *steps* to provide a form of multi-phase transaction. If steps fail, rollbacks are specified to reverse all changes that have occurred.

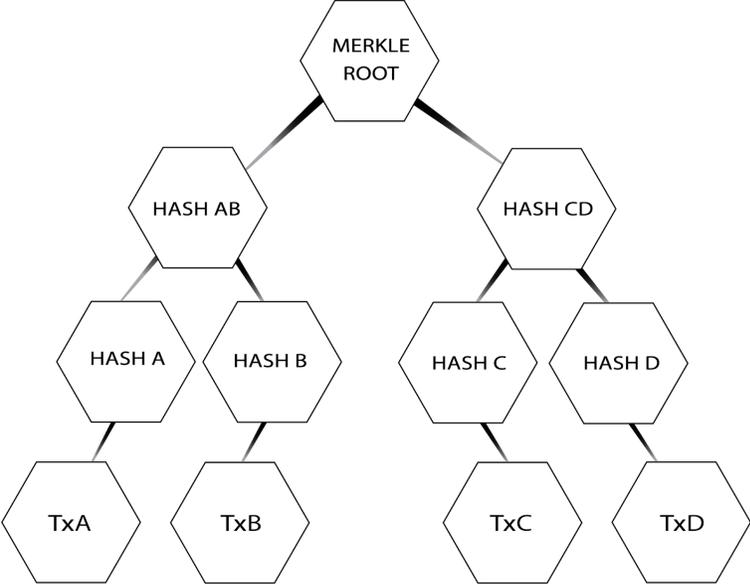
In addition to oracle support, Pact offers an “omnidirectional” approach to service design. Functions in a Pact smart contract automatically become front-end REST API endpoints with native JSON representations of all Pact data. Name-based resolution facilitates a “horizontal” service API to other smart contracts, directly calling functions on each other within a blockchain transaction. Finally, all data can be written directly to relational database back-ends, making it easy to export data for integration with downstream systems and further analysis.

Chainweb

Chainweb is a new *parallel-chain* Proof-of-Work architecture comprised of braided chains that mine the same currency and transfer liquidity between them. Unlike existing Proof-of-Work architectures, Chainweb offers massive throughput. Starting with 1,250 chains executing upwards of 10,000 transactions per second, while maintaining Proof-of-Work’s unmatched resilience against fraud and censorship.

The design of Chainweb begins with the idea of adding Bitcoin's Simple Payment Verification capability (SPV) to Kadena’s smart contracts. SPV, also known as *light client* support, allows users to verify a particular transaction without processing the entire blockchain. This is done by querying the blockchain for a *Merkle proof*. In such a proof, hashes of a transaction and its history are stored as *leaf* and *branch* nodes of a tree-like

data structure. Starting from the transactions stored in the leaves (the bottom-most nodes), adjacent transaction hashes are merged together to form new nodes in the tree, with the most recent transaction proof forming the *root*.



In Kadena's public blockchain, smart contract SPV allows for automated *cross-currency exchanges* with multi-step pacts. Suppose that Alice wants to exchange her Kadena coin with Bob's Bitcoin. She initiates a pact by escrowing her Kadena coin, and Bob responds with a Merkle proof of the corresponding transfer to her Bitcoin address. The pact then validates the proof of the transaction history, and releases the escrowed funds to Bob's Kadena account.

To allow these exchanges, Pact requires direct support for the Merkle proofs of the other popular currencies, such as Bitcoin and Ether. For Kadena-to-Kadena pacts, *two Kadena blockchains may be ran in parallel*, each minting different coins and exchanging them trustlessly with SPV.⁷ With two parallel chains, overall transaction throughput doubles.

Chainweb Security

For cross-currency exchanges, one must verify the authenticity of the Merkle proof by "linking" the proof's root to the roots of the longest branch of an external cryptocurrency. Since smart contracts cannot access the internet, an oracle must be trusted to provide the Merkle roots. However, for Kadena-to-Kadena transfers, each chain can publish a prior

⁷ To avoid duplicate accounts, the transfers destroy coin on the debited chain and create it on the credited one, conserving coins "globally."

Merkle root of the opposite chain in each block header, effectively using chain consensus to establish one chain as an “oracle” of the other’s Merkle roots.

Additionally, it is not enough to simply prove transactions using the other chain’s roots. Global conservation of coins demands enforcing a single view of the transaction history across the chains. To achieve this, each chain hashes the Merkle roots of the opposite chain into its own, and inspects the roots to validate that the branches do not diverge. A “braid” of chains emerges that can only be attacked as a unit. To maliciously fork either chain, an attacker must hash *both chains* faster than honest miners.

By extending the two-chain design to many chains, we arrive at Chainweb. Braided chains incorporate Merkle proofs from adjacent chains in a graph layout that ensures that proofs quickly propagate to every other chain in the system (within some maximum block depth). Solutions to the degree-diameter problem reveal that a layout of 1,250 chains reaches global propagation in just 3 blocks.⁸ Larger configurations are possible with increases in confirmation depth. With each additional chain, throughput increases linearly and the hash rate required to sustain a malicious fork of the Chainweb “braid” to confirmation depth converges on the cumulative hash rate of every running chain.

Importantly, chains can be *specialized* for particular operations, such as supporting file storage applications. Developers can provision their throughput requirements by choosing which chains their smart contracts run on.

Together, these innovations can support unexpected spikes in demand, operate more efficiently with the same resilience, and enable an entirely new class of business applications.

⁸ https://en.wikipedia.org/wiki/Table_of_the_largest_known_graphs_of_a_given_diameter_and_maximal_degree