Kadena

The Kadena Public Blockchain

Project Summary Whitepaper

Version 1.3 February, 2019

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, safe and simple smart contract construction, and integrated smart contract upgrades without a hard fork. These features provide a trustworthy ecosystem for distributed applications, safely shared business workflows, and flexible governance mechanisms for the modern age.

Contents

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 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 feature 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 inspiration in mind, Pact was purposefully made Turing-incomplete—recursion is not possible in Pact[1] 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, and in Ethereum, for instance, they could "run out of gas"[2] and fail to complete at any time.

Formal Verification of Contract Code

The notorious failures and exploits that emerged in the Ethereum ecosystem have exposed the risks of automating central business processes with smart contracts as they exist today. *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 approach 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 uses 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 domain-specific language 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 an upgrade, 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 and 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 can never be upgraded once installed.[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 frequent abuse of protocol-level hard forks in order to fix by overwriting old ledger data catastrophic exploits that emerged on the blockchain. 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 the trustless, decentralized philosophy of a public blockchain.[5]

We assert that any mature smart contract system must support the ability to upgrade contracts *without* requiring a hard fork. Upgrades 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 with either a *keyset* (a rule for allowing public-key signers to authorize operations) or a generalized governance function. While keysets offer single- and multiple-key signing operations, governance functions extend governance to 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.

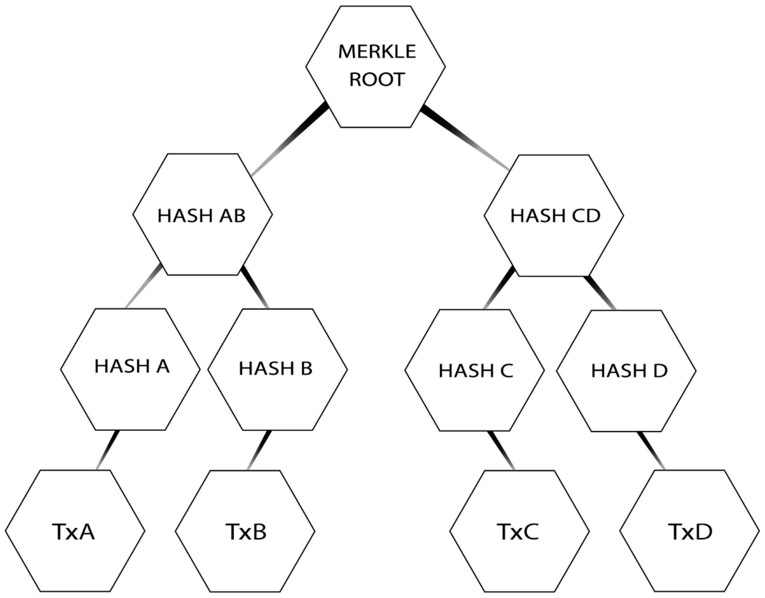
Pact makes the automation of oracle processes easy and safe with *pacts*, functions that can *yield* and *resume* at distinct *steps* to provide a form of multi-phase transaction. If steps fail, rollbacks are specified to reverse all changes that have occurred.

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 all mine the same native currency and transfer liquidity between each other. Unlike existing Proof-of-Work architectures, Chainweb offers massive throughput without significantly increasing hash power. Chainweb has the potential to grow to at least 1,250 chains executing upwards of 10,000 transactions per second while still maintaining the unmatched resilience against fraud and censorship of Proof of Work.

The design of Chainweb began 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 verification 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 Chainweb, multiple blockchains run in parallel, each minting different coins and exchanging them trustlessly with SPV.[7] With two parallel chains, overall transaction throughput doubles. Each chain publishes a prior Merkle root of its peer chains in each block header, effectively using chain consensus to establish each chain as an “oracle” of the others’ Merkle roots.

The native Kadena token can be moved from chain to chain via SPV, and to achieve a single view of transaction history across all chains, each chain hashes the Merkle roots of its peer chains into its own and inspects the roots to validate that the branches do not diverge. From the hash links emerge a braid of chains that can only be attacked as a whole. To maliciously fork any chain, an attacker must hash all chains faster than honest miners.

In Chainweb, the braided chains incorporate Merkle proofs from adjacent chains in a fixed graph layout that ensures that proofs quickly propagate to every other chain in the system within some maximum block depth. The fixed graph structure for Chainweb will follow solutions to the degree-diameter problem, which allow for maximum propagation of information with a minimum number of hops and a minimum number of messages between chains. For example, a layout of 1,250 chains reaches global propagation in just 3 blocks.[8] Larger configurations for Chainweb are possible with increases in confirmation

[7] To avoid duplicate accounts, the transfers destroy coin on the debited chain and create it on the credited one, conserving coins “globally.”
[8] https://en.wikipedia.org/wiki/Table_of_the_largest_known_graphs_of_a_given_diameter_and_maximal_degree

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.

Chains in Chainweb have the potential to be specialized for particular operations, such as supporting file storage applications. Developers can provision their throughput requirements by choosing on which chains to run their smart contracts. Together, these innovations of throughput, security, and specialization can support unexpected spikes in demand, operate more efficiently with the same resilience, and enable an entirely new class of business applications.

Chainweb Interoperability

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 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. This interoperability allows for Kadena to provide the computational backbone for many different kinds of applications.

Pact and Chainweb together is a powerful combination that allows developers and businesses to use a public blockchain in a safe, secure, and stable way that they know they'll be able to trust. Smart contracts should be useful, human readable, and verified to behave as intended, and a public blockchain should be fast, stable, and capable of handling the throughput necessary for real applications. With Kadena's public blockchain ecosystem we're bringing blockchain application development into a stronger, more effective paradigm.