kadena

scalable blockchain | smarter contracts

The Kadena Public Blockchain

Project Summary Whitepaper

August 28, 2017

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 combination of the smart contract language Pact, the working *formal verification* solution Yoke, and the new *parallel-chain* Proof-of-Work protocol Chainweb provide distributed applications and services with the necessary assurance to host robust business workflows, governance mechanisms that allow smart contract upgrades without a hard fork, and throughput capable of executing upwards of 10,000 transactions per second.

Contents

Safer, better 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 operation of business applications: enforcing business rules (with unambiguous error messages on failure); modeling and maintaining database schemas; and authorizing users to perform sensitive operations. Leaving these as tasks for each developer to create from scratch requires advanced expertise, 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 reflects the firm belief that smart contracts need to be *readable by humans and verifiable by computers*. We strongly disagree with the use of virtual machines that require the storage and invocation of illegible bytecode and instead designed Pact as an interpreted language where the code is always available. Technically-savvy lawyers and business executives can easily review the business logic of the smart contract code, and the language becomes easier to learn for a broad base of developers and accessible even to some non-programmers.

Pact's focus on safety is inspired by Bitcoin scripts, which were designed with a minimal feature-set to provide the most assurance possible for coin transfers. Pact prohibits recursion and unterminated loops, eschewing a radically more dangerous *Turing-complete* model and eliminating an entire class of bugs[1]. Pact retains sufficient power for iterative computations using functional list processing techniques (map, filter, fold). We observe that use-cases that truly require Turing-complete functionality are ill-suited for the compute-constrained blockchain environment: tasks such as path-finding or analytic queries require an unpredictable amount of gas[2] and can fail to complete in the Ethereum VM.

Formal Verification of User Code with Yoke

High-visibility failures and exploits that have emerged in the Ethereum ecosystem—the DAO theft being the prime example – and the high risk of automating central business processes with smart contracts reveal an acute need for *formal verification*. In this process, code is transformed to yield a mathematical model of its functionality, which is then used to prove that it operates correctly in all cases. This is radically different than normal software testing, as this technology can validate correct behavior over infinite input spaces and states. By comparison, standard software development practices can only test for known situations.

We are very proud to offer Yoke, a major advance in smart contract technology that works today. Yoke completes our vision of offering smart contracts that are readable by humans and verifiable by computers. The Yoke formal verification system starts by compiling Pact smart contract code directly into the SMT-LIB2 language. This code can then be loaded into the Z3 theorem prover, yielding a system usable by experts in SMT techniques.

Formal verification and satisfiability modulo theories (SMT) are highly specialized areas of computer science research that require expertise generally beyond that of production programmers. In order to spread the high-assurance environment of Pact-based smart contracts as widely as possible, Yoke offers a simple domain specific language (DSL) that can be inserted directly into Pact code to express inviolable business rules. For example, double-spends can be prohibited by declaring that a column tracking coin balances must always be net zero. Like Pact, Yoke is easy to read and allows non-technical stakeholders to check verification rules for completeness.[3]

[1] The Ethereum DAO hack is notably a 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.

[3] There is no system available or proposed today that can offer Yoke'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.

Cryptocharters: Smart Contracts with Governance

Blockchains run on *client software* that must be upgraded to provide protocol-level updates. To achieve this, the blockchain undergoes a *hard fork* wherein 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 agreement of what the community deems appropriate.

The advent of smart contracts on Ethereum has led to a serious abuse of hard forks. These contracts are *immutable:* once installed the code cannot be changed[4], requiring the use of hard forks to fix catastrophic smart contract exploits. While protocol-only hard forks can be contentious, 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 ethos.[5]

We assert that any mature smart contract system must support the ability to upgrade contracts without requiring a hard fork at any point in its lifecycle in order to resolve critical issues and deploy strategic enhancements. In Pact, every smart contract is a *cryptocharter,* which requires contracts to specify a governance function. Cryptocharters can be fully autonomous, operate under a board of directors, or grant every member a vote. For the first time, contract authors will have the freedom to model governance structures however they wish.[6]

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 and returns data authenticated by proofs of provenance like 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* interaction, 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

[4] Software techniques to introduce indirection in invoking smart contract code can ameliorate 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.

in the database, handling non-responses, managing escrow payments, and cleanup 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 commit. If steps fail, rollbacks are specified to reverse changes post-failure.

Beyond 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. 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 protocol comprised of "sister" chains that mine the same currency and transfer liquidity between chains. Unlike existing Proof-of-Work implementations, 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 to add Bitcoin's Simple Payment Verification capability (SPV) to smart contracts. Simple Payment Verification, which is also known as *light client* support, allows users to verify a particular transaction without processing the entire blockchain by obtaining a *Merkle proof* from any blockchain node.

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

To support these exchanges, Pact must have native support for Bitcoin Merkle proofs and separate support for ether and other currencies. In fact, Pact can even support Kadena Merkle proofs, too. With Kadena-to-Kadena pacts, *two Kadena blockchains could run in parallel,* each minting different coins and exchanging them

trustlessly with SPV.[7] With two parallel chains, overall transaction throughput doubles.

However, some unsolved problems remain. For cross-currency exchanges, one needs to verify the authenticity of the Merkle proof by "linking" the proof's root to the roots of the external cryptocurrency's longest branch. Because smart contracts cannot access the internet, an oracle must be trusted to provide the Merkle roots. But for Kadena to Kadena transfers, each chain can publish a prior 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 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 "rope" of chains emerges that can only be attacked as a unit: to maliciously fork either chain, an attacker must hash *both chains* faster than the honest miners.

By extending the two-chain design to *n* chains, we arrive at Chainweb. Sister 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 "rope" to confirmation depth converges on the cumulative hash rate of every chain running.

Importantly, in Chainweb 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.

[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