FIRST WORKSHOP ON TRUSTED SMART CONTRACTS 2017 (WTSC17)

Malta, 7th April 2017
http://fc17.ifca.ai/wtsc/

Workshop associated to Financial Cryptography and Data Security 2017
http://fc17.ifca.ai/

WTSC17 Program Committee

Massimo	Bartoletti	Universiry of Cagliari IT	
Andrea	Bracciali	University of Stirling UK	chair
Eimear	Byrne	University College Dublin IR	
Martin	Chapman	King’s College London UK	
Tiziana	Cimoli	University of Cagliari IT	
Nicola	Dimitri	University of Siena IT	
Stuart	Fraser	Wallet.Services UK	
Laetitia	Gauvin	ISI Foundation IT	
Davide	Grossi	University of Liverpool UK	
Iain	Henderson	Jlink Lab UK	
Yoichi	Hirai	Ethereum DEV DE	
Camilla	Hollanti	Aalto University FI	
Ioannis	Kounelis	Joint Research Centre (JRC) European Commission	
Loi	Luu	National University of Singapore	
Michele	Marchesi	University of Cagliari IT	
Peter	McBurney	King's College London UK	
Neil	Mclaren	Avaloq Innovation Ltd UK	
Philippe	Meyer	Avaloq Innovation Ltd UK	
Mihail	Mihaylov	Vrije Universiteit Brussel BE	
Sead	Muftic	KTH Royal Institute of Technology SE	
Igor	Nai Fovino	Joint Research Centre (JRC) European Commission	
Daniela	Paolotti	ISI Foundation IT	
Federico	Pintore	University of Trento IT	
Massimiliano	Sala	University of Trento IT	chair
Ilya	Sergey	University College London UK	
Jason	Teutsch	University of Chicago US	
Roberto	Tonelli	University of Cagliari IT	
Yaron	Velner	Hebrew University IL	
Luca	Vigano	King's College London UK	

WTSC17 Overview

These informal proceedings collect the papers and posters accepted at the 1st Workshop on Trustable Smart Contracts (WTSC17) associated to the Financial Cryptography 2017 conference held in Malta in April 2017. WTSC17 focuses on smart contracts, i.e. self-enforcing agreements in the form of executable programs that are deployed to and run on top of blockchains. These technologies introduce a novel programming framework and execution environment, which are not satisfactorily understood at the moment. Multidisciplinary and multifactorial aspects affect correctness, safety, privacy, authentication, efficiency, sustainability, resilience and trust in smart contracts. WTSC17 aims to gather together researchers from both academia and industry to address the scientific foundations of Trusted Smart Contract engineering, i.e. contracts that enjoy some verifiable "correctness" properties, and to discuss open problems, proposed solutions and the vision on future developments.

This first edition of WTSC17 received nineteen submissions, of which nine were accepted as full papers and three as posters, after peer review managed by a Programme Committee featuring members from academia, institutions and industry from eleven countries, who kindly accepted to support the event. Accepted papers will be published with Springer. WTSC17 also enjoyed an invited talk by Vitalik Buterin (Ethereum Foundation), a prominent contributor to the world of smart contracts, who kindly accepted our invitation.

Andrea Bracciali
Federico Pintore
Massimiliano Sala

WTSC17 Organisers

WTSC17 Programme

13:55 Opening remarks

Session 1: Invited Talk
14:00-15:00

Blockchain and Smart Contract Mechanism Design Challenges

Vitalik Buterin

15:00 Coffee break with poster session

Fostering Consumers' Energy Market through Smart Contracts
Ioannis Kounelis, Gary Steri, Raimondo Giuliani, Dimitrios Geneiatakis, Ricardo Neisse, and Igor Nai Fovino

Scripting smart contracts for distributed ledger technology
Pablo Lamela Seijas, Simon Thompson and Darryl McAdams

ZeroTrade: Privacy Respecting Assets Trading System based on Public Ledger
Lei Xu, Lin Chen, Nolan Shah, Zhimin Gao, Yang Lu and Weidong Shi

Session 2
15:30-16:00

Findel: Secure Derivative Contracts for Ethereum
Alex Biryukov, Dmitry Khovratovich and Sergei Tikhomirov

Decentralized Execution of Smart Contracts: Agent Model Perspective and Its Implications
Lin Chen, Lei Xu, Nolan Shah, Zhimin Gao, Yang Lu and Weidong Shi

Session 3
16:00-17:20

A Concurrent Perspective on Smart Contracts
Ilya Sergey and Aquinas Hobor

An empirical analysis of smart contracts: platforms, applications, and design patterns
Massimo Bartoletti and Livio Pompianu

Trust in Smart Contracts is a Process, As Well
Firas Al Khalil, Tom Butler, Leona O'Brien and Marcello Ceci

Defining the Ethereum Virtual Machine for Interactive Theorem Provers
Yoichi Hirai

17:20 Break
Session 4
17:30-18:30

A Proof-of-Stake protocol for consensus on Bitcoin subchains
Massimo Bartoletti, Stefano Lande and Alessandro Sebastian Podda

On the feasibility of decentralized derivatives markets
Shayan Eskandari, Jeremy Clark, Moe Adham and Vignesh Sundaresan

SmartCast: An Incentive Compatible Consensus Protocol Using Smart Contracts.pdf
Abhiram Kothapalli, Andrew Miller and Nikita Borisov

WTSC17 Papers

Blockchain and Smart Contract Mechanism Design Challenges

WTSC17 Keynote Talk

Vitalik Buterin

Ethereum Foundation

Abstract. Arguably, the true genius behind the success of Bitcoin, Ethereum and similar systems was not the specific design of their blockchain, or their use of algorithms that resemble forms of distributed consensus in order to maintain security; rather, it is the innovation of cryptoeconomics - the art of combining cryptographic techniques and economic incentives defined and administered inside a protocol in order to encourage users to (correctly) participate in certain roles in the protocol, and thereby preserve and maintain certain desired properties of the protocol. I describe the key ideas in the abstract, then apply them to Bitcoin proof of work, the Schellingcoin oracle, Casper, as well as describing several key open problems in blockchain-based system design.

Fostering Consumers' Energy Market through Smart Contracts

Ioannis Kounelis, Gary Steri, Raimondo Giuliani,
Dimitrios Geneiatakis, Ricardo Neisse, and Igor Nai Fovino

European Commission, Joint Research Centre (JRC)
Cyber and Digital Citizens' Security Unit
Via Enrico Fermi 2749, 21027 Ispra, Italy
{firstname.lastname}@ec.europa.eu

Micro-generation is the capacity for consumers to produce electrical energy in-house or in a local community. The concept of "market" indicates the possibility of trading the electricity that has been micro-generated among producers and consumers, where a user acting both as a producer and consumer is called a "prosumer". Traditionally, this market has been served by pre-defined bilateral agreements between prosumers and retail energy suppliers. Until now, electricity-generating prosumers have not had real access to the energy market, which remains a privileged playing field for the institutionalised energy suppliers.

Indeed, the main options considered so far by the technical literature, were completely centralised and their viability was challenged as they introduce additional management fees and costs and assume the intervention of a trusted third party reducing once again the potential gains of end-users. New approaches should be developed enabling end-users to have free access to the energy market. In this regard we propose a solution that utilizes the advantages of using a blockchain for handling automatically the energy exchange. According to our approach, self-generated electricity could normally be either consumed within the house, accumulated in next-generation batteries for later use, or simply given back to the grid, where, thanks to the distributed and pervasive nature of the blockchain, the produced energy could be redeemed elsewhere.

Exploiting the potentialities of blockchains and distributed ledgers, with our work we propose a solar energy production and distribution architecture that uses smart contracts to support automatic and distributed energy exchange, thus allowing the development of an energy micro-generation market more open and fruitful, from an end-user perspective. More in detail we introduce a platform named Helios that facilitates micro-generators to exchange energy freely in a limited geographical area. In this setup a custom made Internet of Things smart meter is used to account and register the micro-generated energy in the blockchain, while the smart contract supports the monitoring and accounting of energy exchange in terms of a financial transaction. The model has been implemented and validated through an in-house developed test-bed composed by a real physical energy infrastructure and the related control and ICT layers. To the best of our knowledge, Helios is among the first solutions built on off the shelf devices and open source technologies, enabling prosumers to access the energy market.

Scripting smart contracts for distributed ledger technology

Pablo Lamela Seijas[1], Simon Thompson[1], and Darryl McAdams[2]

[1] University of Kent, UK
{pl240,S.J.Thompson}@kent.ac.uk
[2] IOHK
darryl.mcadams@iohk.io

Distributed Ledger Technology (DLT) offers a way of maintaining a synchronised log in a non-centralised, distributed way; notably, this allows the implementation of cryptocurrencies and, more recently self-enforcing smart contracts. Bitcoin is the first widely-used implementation of a cryptocurrency but it has very limited scripting capabilities in practice. Ethereum allows smart contracts to contain arbitrary time-bounded turing-computable code that is executed and validated in a virtual machine. Nxt moves scriptability to clients and provides a delimited functionality through an API.

Because smart contracts can control money and potentially other assets, it is crucial that they behave as expected, not only in normal conditions, but also when attacked by malicious agents. In particular, contracts must be *reentrant* if they call unknown code, they must gracefully handle all kinds of *execeptions*, they must not expect agents to collaborate (in some cases by including *rewards and penalties* to deter attacks).

Designers of smart-contract languages and cryptocurrencies may mitigate the likelihood of errors being made by their users by carefully designing them to be intuitive, explicit, and by providing well-tested artefacts. Some examples of effort in this direction include: the use of *zero-knowledge proofs* for providing anonymity (see Zerocash); the use of *SNARKS* to hide private inputs (Hawk allows to design contracts by separating private and public parts); and allowing the use and enforcement of *higher-level specifications*, like the use of polymorphic types, combinators, finite-state machines (FSMs), or domain specific languages (DSLs). Additionally, there are many open challenges that are specific to DLT systems, like the design of ways for *amending the rules* (see Tezos), the *unpredictability* of the initial execution state derived from the decentralisation, the need for a safe *source of randomness*, the *cost of validating* the contracts (which could be mitigated through the use of verifiable computation), the amount of work required by *proof-of-work* (see *proof-of stake*), and the need to preserve the delicate *equilibrium of incentives* that keeps block-chains secure.

In the full paper[3], we provide references for all the work mentioned here, we survey these and other representative examples of the advanced use of cryptocurrencies and blockchains beyond their basic usage as a payment method, and we analyse existing scripting solutions, their strengths and weaknesses, and some existing solutions for known problems with them. Through our work, we have seen that, while there have been many diverse efforts in different directions, there are still many open questions, no universal solutions, and lots of room for future research and experimentation.

[3] Pablo Lamela Seijas, Simon Thompson, and Darryl McAdams. *Scripting smart contracts for distributed ledger technology*. 2016. URL: `https://eprint.iacr.org/2016/1156.pdf`

ZeroTrade: Privacy Respecting Assets Trading System based on Public Ledger

Lei Xu, Lin Chen, Nolan Shah, Zhimin Gao, Yang Lu, and Weidong Shi

University of Houston, Houston TX 77004, USA

Motivation. Public ledger is a decentralized book keeping technology and is believed to have the potential to revolutionize many areas. Besides handling crypto-currency, public ledger can be used to tokenize arbitrary assets, and then support trading of these asset tokens in a decentralized manner. With public ledger based token trading system, users do not necessarily convert their assets to currencies, but can exchange assets directly. It also avoids unnecessary transportation as the asset only needs to be physically transferred to its last owner. Furthermore, utilization of the public ledger does not require that users have to trust each other in order to trade tokens safely. However, using decentralized public ledger for trading asset tokens raises serious privacy concerns. Because token ownership information is stored on the public ledger and disclosed to the public, anyone can uncover users trading activities and history. For a token based asset trading platform, all tokens are unique and transactions are usually two-ways or multi-ways. In response to these challenges, we propose ZeroTrade, a privacy respecting heterogeneous assets trading system that leverages various cryptography tools to conceal the exchange trace of asset tokens and takes advantage of public ledger for guaranteeing fairness of asset token exchange.

Solution. ZeroTrade involves trusted hubs that are responsible for converting assets to tokens and back, where trusted means that hubs will generate/accept valid tokens, and uses the public ledger to record all token exchange information. When two or more users want to exchange tokens with each other, each user picks an agent for the exchange. Asset tokens are first poured into a pool and users leverage agents to obliviously retrieve tokens from the pool in order to finish the exchange. The oblivious retrieving process cut off the connection between the original user and the agent. Therefore, one cannot determine the relationship between the original users who want to exchange tokens by observing information recorded on the public ledger.

To implement the design, ZeroTrade leverages different cryptography tools including zero-knowledge proof and various encryption techniques. Considering various demands in token trade, ZeroTrade also supports operations like partial token trade and revocation. A preliminary evaluation of the performance shows that ZeroTrade only adds limited burden on top of the public ledger. More detailed information can be found in the full version of the paper.

Conclusion. ZeroTrade provides a privacy friendly platform for asset trading based on public ledger. For the next step, we plan to implement a fully functional prototype and considering more complex token trading scenarios.

Findel: Secure Derivative Contracts for Ethereum

Alex Biryukov[1], Dmitry Khovratovich[2], Sergei Tikhomirov[2]

[1] alex.biryukov@uni.lu
SnT, University of Luxembourg
[2] {khovratovich,sergey.s.tikhomirov}@gmail.com
SnT, University of Luxembourg

Abstract. Blockchain-based smart contracts are considered a promising technology for handling financial agreements securely. In order to realize this vision, we need a formal language to unambiguously describe contract clauses. We introduce Findel – a purely declarative financial domain-specific language (DSL) well suited for implementation in blockchain networks. We implement an Ethereum smart contract that acts as a marketplace for Findel contracts and measure the cost of its operation. We analyze challenges in modeling financial agreements in decentralized networks and outline directions for future work.

Keywords: blockchain, smart contracts, financial engineering, domain-specific language

1 Introduction

Financial derivatives – contracts defined in terms of other contracts – play a major role in modern economy[3]. Financial industry lacks a universal domain-specific language. Natural language is unsuitable for expressing contracts due to its inherent ambiguity. An influential paper [JES00] is one of many attempts to create a rigorous DSL that would mitigate disputes and stimulate automated processing of complex derivatives. It leverages ideas from functional programming and uses a succinct set of basic building blocks to express financial agreements. A key feature of this notation is composability: new indefinitely complex derivatives can be defined based on existing ones. Due to their nested structure, contracts in this DSL are well-suited for automated processing, including valuation. The authors do not specify an enforcement mechanism though: execution is performed by an implicit environment. This work forms the basis for research [Gai11] [Sch14] and commercial [FSNB09] [Mor16] projects.

The idea of smart contracts – computer programs for (semi-)automatic enforcement of agreements – dates back to mid-1990s [Sza97]. Blockchain networks,

[3] The derivatives market is comparable in size to the world's GDP. The gross market value of all outstanding over-the-counter derivatives is \$20.7 trillion [Bis16] (2016). The world GDP in 2015 is \$73,9 trillion [Wor16].

notably Ethereum, became the first practical implementation of this idea and fueled interest in the concept [dC16]. Ethereum is a network of mutually distrusting nodes, which nevertheless establish consensus on the results of computations without the need of a trusted third party.

An obvious use case for blockchain-based smart contracts is to securely manage financial agreements. A naive approach to doing so is to encode the entire logic of an agreement inside a smart contract. Expressing complex clauses in a general-purpose programming language, like Ethereum's Solidity, is error-prone [ABC16] [Sir16]. We propose a safer approach that separates the description of a contract from its execution. A user only defines what a contract *is* ("I owe you $10 tomorrow"), not *how* it is executed ("if the timestamp is greater than $t_0, \dots$"). The entire execution logic is implemented inside a smart contract, which is executed by nodes of a blockchain network. Thus we take the best of both worlds: unambiguity and composability of a concise declarative DSL, and trustless execution of blockchain-based smart contracts.

We introduce **Findel** (Financial Derivatives Language) – a declarative financial DSL (Section 2) capable of expressing most common derivatives (Appendix A). We implement an Ethereum contract that manages Findel contracts (Section 3) and prove our approach viable in terms of cost (Section 4).

2 Findel contracts syntax

2.1 Definitions

Definition 1. *A* ***Findel contract***[4] *C is a tuple (D, I, O), where D is the* ***description****, I is the* ***issuer****, and O is the* ***owner*** *(collectively called parties).*

Definition 2. *A* ***description*** *of a Findel contract is a tree with* ***basic primitives*** *as leaves and* ***composite primitives*** *as internal nodes. The following BNF grammar defines primitives:*

$\langle basic \rangle$ ::= `Zero` | `One` ($\langle currency \rangle$)

$\langle scale \rangle$ ::= `Scale` ($\langle number \rangle$, $\langle primitive \rangle$)

$\langle scaleObs \rangle$ ::= `ScaleObs` ($\langle address \rangle$, $\langle primitive \rangle$)

$\langle give \rangle$ ::= `Give` ($\langle primitive \rangle$)

$\langle and \rangle$ ::= `And` ($\langle primitive \rangle$, $\langle primitive \rangle$)

$\langle or \rangle$ ::= `Or` ($\langle primitive \rangle$, $\langle primitive \rangle$)

$\langle if \rangle$ ::= `If` ($\langle address \rangle$, $\langle primitive \rangle$, $\langle primitive \rangle$)

$\langle timebound \rangle$ ::= `Timebound` ($\langle timestamp \rangle$, $\langle timestamp \rangle$, $\langle primitive \rangle$)

[4] We may refer to Findel contracts simply as contracts, when the distinction between them and Ethereum smart contracts is clear from the context.

$\langle composite \rangle$ ::= $\langle scale \rangle \mid \langle scaleObs \rangle \mid \langle give \rangle \mid \langle and \rangle \mid \langle or \rangle \mid \langle if \rangle \mid \langle timebound \rangle$

$\langle primitive \rangle$ ::= $\langle basic \rangle \mid \langle composite \rangle$

We distinguish between composite and basic primitives, because the former contain other primitives as sub-nodes while the latter do not. *Currency*, *number*, *address*, and *timestamp* are implementation dependent data types. D and I can not be modified after a contract is created.

A financial company typically has templates for common contracts. Parties who wish to sign an agreement write their names on a copy of a template and sign it, making it unique and legally binding. In our model, Findel contracts represent signed copies while their descriptions represent blank templates.

Traditional contracts usually contain clauses that regulate sub-ideal situations, i.e., a breach of contract. Findel does not distinguish between "ideal" and "sub-ideal" situations. All right and obligations are expressed uniformly. Section 3.3 discusses issues related to contract enforcement.

Table 1 informally defines the primitives' execution semantics.

Primitive	**Informal semantics**
Basic	
Zero	Do nothing.
One(*currency*)	Transfer 1 unit of *currency* from the issuer to the owner.
Composite	
Scale(k, c)	Multiply all payments of c by a constant factor k.
ScaleObs($addr, c$)	Multiply all payments of c by a factor obtained from $addr$.
Give(c)	Swap parties of c.
And(c_1, c_2)	Execute c_1 and then execute c_2.
Or(c_1, c_2)	Give the owner the right to execute either c_1 or c_2 (not both).
If($addr, c_1, c_2$)	If b is true, execute c_1, else execute c_2, where b is a boolean value obtained from $addr$.
Timebound(t_0, t_1, c)	Execute c, if the current timestamp is within $[t_0, t_1]$.

Table 1. Findel contract primitives

Table 2 illustrates the composability of Findel[5].

2.2 Execution model

Findel contracts have the following lifecycle:

1. The first party **issues** the contract by specifying D, becoming its issuer. This is a mere declaration of the issuer's desire to conclude an agreement and entails no obligations.

[5] INF is a symbol representing infinite time, i.e., $t_0 < INF$ for every t_0. δ is an implementation dependent constant intended for handling imperfect precision of time signal in distributed networks.

Contract	**Definition**
$\text{At}(t_0, c)$	$\text{Timebound}(t_0 - \delta, t_0 + \delta, c)$
$\text{Before}(t_0, c)$	$\text{Timebound}(now, t_0, c)$
$\text{After}(t_0, c)$	$\text{Timebound}(t_0, INF, c)$
$\text{Sell}(n, CURR, c)$	$\text{And}(\text{Give}(\text{Scale}(n, \text{One}(CURR))), c)$

Table 2. Examples of custom Findel contracts

2. The second party **joins** the contract, becoming its owner. As a result, both parties accept certain rights and obligations.
3. The contract is **executed** immediately as follows:
   (a) Let the root node of the contract's description be the current node.
   (b) If the current node is either `Or` or `Timebound` with $t_0 > now$, postpone the execution: issue a new Findel contract with the same parties and the current node as root. The owner can later demand its execution.
   (c) Otherwise, execute all sub-nodes recursively[6].
   (d) Delete the contract.

The execution outcome is fully determined by description D, execution time t, and external data $\mathcal{S}$ retrieved at time t.

2.3 Example

Suppose Alice sells to Bob a zero-coupon (i.e., paying no interest) bond that pays \$11 in one year for \$10:

$$c_{zcb} = \text{And}(\text{Give}(\text{Scale}(10, \text{One}(USD))), \text{At}(\text{now+1 years}, \text{Scale}(11, \text{One}(USD))))$$

We now show how c_{zcb} is executed step by step.

1. And executes; Bob temporarily owns two new contracts:

Alice's contracts	
Alice's balance	100
Bob's contracts	$\text{Give}(\text{Scale}(10, \text{One}(USD)))$ $\text{At}(\text{now + 1 years}, \text{Scale}(11, \text{One}(USD)))$
Bob's balance	10

2. Give executes; Alice owns a new contract:

Alice's contracts	$\text{Scale}(10, \text{One}(USD))$
Alice's balance	100
Bob's contracts	$\text{At}(\text{now + 1 years}, \text{Scale}(11, \text{One}(USD)))$
Bob's balance	10

[6] In case of `Or`, execute exactly one of the sub-nodes, according to the owner-submitted value indicating the choice; delete the other one. It is the only primitive that requires an additional user-supplied argument for execution.

3. Scaled One transfers \$10 go from Bob to Alice:

Alice's contracts	
Alice's balance	110
Bob's contracts	$\text{At}(\text{now} + 1\ \text{years}, \text{Scale}(11, \text{One}(USD)))$
Bob's balance	0

4. In one year Bob claims \$11 from Alice:

Alice's contracts	
Alice's balance	99
Bob's contracts	
Bob's balance	11

3 Implementation

We develop an Ethereum smart contract, referred to as marketplace, that keeps track of users' balances and lets them create, trade, and execute Findel contracts. The Findel DSL is network-agnostic and can be implemented on top of any blockchain with sufficient programming capabilities.

3.1 Ethereum overview

Ethereum is a decentralized smart contracts platform [But14] [Woo14]. Ethereum full nodes store data, perform computations, and maintain consensus about the state of all accounts using a proof-of-work mechanism similar to that in Bitcoin. Programs (Ethereum smart contracts) are stored on the blockchain as Ethereum virtual machine (EVM) bytecode, a Turing-complete language. Programmers write contracts in high-level languages targeting EVM, most popular being Solidity and Serpent (we use the former).

A contract can call other contracts' functions and send them units of Ether – the Ethereum native cryptocurrency. To launch a particular function of a contract, a user must send a well-formed transaction to the Ethereum network.

Each EVM operation has a fixed cost in *gas*. A user pays upfront for the maximum amount of gas the computation is expected to consume and gets a partial refund after a successful execution. If an exception (including "out of gas") occurs, all changes are reverted, but the gas is not refunded.

3.2 Implementation details

Users and balances We implement the objects defined in Section 2.1 with struct data types `Description` and `Fincontract`. We also introduce the `User` type that contains the user's Ethereum address and balances in all supported currencies. Users, descriptions and contracts are stored in their respective mappings (a generic key-value storage type in Solidity) in the marketplace's storage.

The ultimate effect of every financial agreement is changing the parties' balances (with clauses specifying when and under what conditions it should occur). We stick to a naive approach: each user is assigned an array of balances for each supported currency. Although easily implementable, it introduces a single point of failure: the marketplace holds users' deposits.

The only primitive that actually transfers value is One. The `enforcePayment` function implements its execution. It subtracts a given amount in a given currency from the issuer's balance and adds it to the owner's balance. Our current implementation does not enforce any constraints on users' balances that would prevent them from building up too much debt.

Ownership transfer In addition to `issuer` and `owner` (see Definition 1), a `Fincontract` contains an auxiliary `proposedOwner` field. On contract creation, `issuer`, `owner`, and `proposedOwner` are initialized to `msg.sender`. To transfer ownership, the owner sets `proposedOwner` either to the address of the proposed new owner or to `0x0`. Only the proposed owner can (but does not have to) `join` the contract; `0x0` means anyone can do so[7].

Data sources and gateways Ethereum contracts are intentionally isolated from the broader Internet and can not pull data from the Web, as it can not be consistently replicated [Gre16]. Asynchronous requests usually solve the problem: a smart contract records an Ethereum event with request parameters properly encoded. A daemon process at an Ethereum node listens for such events, parses requests, and sends them to the Web. The responses are then sent to the requesting smart contract on behalf of an Ethereum account affiliated with the daemon. The submitted data may be accompanied by a proof of authenticity (say, digital signature on a pre-approved public key)[8].

Financial derivatives often use external data. To prevent a malicious or careless user from creating a Findel contract using untrusted sources, we need to guarantee data authenticity.

Definition 3. *A **gateway** is a smart contract that conforms to the API:*

- ***int getValue()*** *Get the latest observed value*[9].
- ***uint getTimestamp()*** *Get the timestamp at which the latest value was observed.*
- ***bytes getProof()*** *Get the authenticity proof for the latest value.*
- ***update()*** *Update the value.*

[7] Beware of front-runners: Bob can monitor the network and try to join a contract as soon as he sees Alice's attempt to do so. Depending on the network latency and miner's behavior, either transaction can be confirmed.

[8] BTCRelay is a prominent example: users submit Bitcoin block headers to a smart contract, which implies their authenticity from the validity of easily verifiable proof-of-work. After a header is stored on the Ethereum blockchain, users check with a Merkle proof that the Bitcoin block contains a given transaction.

[9] For simplicity, we only consider 256-bit integers as observable values. Boolean values can be trivially simulated via integers.

A gateway connects to an external data source and stores the latest value observed along with the time of observation, and, optionally, a cryptographic proof of authenticity. We do not specify the type of proof a gateway provides. Possible options include Oraclize [Ora16] / TLSNotary [Tls16] and Reality Keys [Rea16].

The marketplace queries a gateway at execution time, if necessary. If the value is fresh and the proof is valid, the execution proceeds, otherwise it is aborted and all changes are reverted. Since a Findel contract may use multiple gateways, the owner is advised to `update` them all shortly before execution.

A possible improvement would be for a gateway to store not only the latest observed value, but a sequence of historical data. This would allow for more straightforward modeling of derivatives that depend on multiple data points, such as barrier options (execute either c_1 or c_2 depending on whether an observable value touches a pre-defined threshold between acquisition and maturity).

We assume that the original data sources (e.g., feeds of reputable financial media) are trustworthy. An extra safety catch would be to query multiple sources, exclude outliers and return an aggregated value. Authenticity of data sources is guaranteed by a secure connection (e.g., TLS) and the existing PKI for authentication ([CF14] and [LC16] propose blockchain-based PKI architectures).

Gateways without publicly available source code should not be trusted.

Execution implementation The `executeRecursively` function implements the execution logic defined in Section 2.2 and returns `true` if executed completely (without creating new contracts) and `false` otherwise. The execution of an expired contract ($t_0 < now$) returns `true` unconditionally[10] and deletes the contract[11]. Every step in the life cycle of a Findel contract issues a system-wide notification (`Event`), allowing users to keep track of contracts they are interested in.

Our implementation deviates from the model (Section 2.1) in that the execution of contracts is not guaranteed. Ethereum contracts can not act on their own: the owner must issue a transaction to trigger execution. The owner may be unable to do so due to either opportunistic behavior, or technical problems, such as loss of connectivity or lack of ether. Thus we presume that Findel contracts are not guaranteed to execute[12]. We discuss this issue in Section 3.3.

We model unbounded Findel contracts (i.e., with INF as the upper time bound) using a global *expiration* constant inside the marketplace contract. Every Findel contract in the Ethereum implementation can only be executed within *expiration* time units after creation (e.g., 10 years).

[10] By definition, an expired contract is equivalent to Zero.

[11] An expired contract should also be deleted even if its owner is offline forever. Our current implementation does not handle the latter case, though it may be considered an attack vector due to increasing storage usage. A possible approach is for a marketplace to offer rewards for keeping track of expired contracts and triggering their deletion.

[12] Compare to [JES00]: "If you acquire *(c1 or c2)* you must immediately acquire either *c1* or *c2* (but not both)". We can not force a user to make this decision.

3.3 Possible improvements

We now discuss the shortcomings of our model and ways to improve it.

Enforcement As mentioned in Section 3.2, Findel contracts are not guaranteed to execute. At first sight, it is a major problem, as contract must impose obligations on parties. In traditional finance, a trusted third party and, ultimately, the state law enforcement are responsible for punishing violators. The closest we can arguably get to enforcement is a conditional penalty implemented inside a Findel contract itself.

Assume Alice issues and Bob joins the following contract:

$$C = Before(t_0, \mathrm{Or}(\mathrm{Give}(\mathrm{One}(USD)), \mathrm{Give}(\mathrm{One}(EUR))))$$

C obliges Bob to give Alice either \$1 or €1 before time t_0. If Bob fails to make a choice on time, Alice does not get the money she was planning to receive[13]. To prevent it, Alice attaches a ”penalty” clause:

$$P = After(t_0, \mathrm{If}(c_{executed}, \mathrm{Zero}, \mathrm{Scale}(2, \mathrm{One}(USD))))$$

$c_{executed}$ is the address of a gateway that indicates whether a particular Findel contract was executed. When Bob joins $C_{penalty} = \mathrm{And}(C, \mathrm{Give}(P))$, Alice obtains the right to claim \$2 from Bob if he fails to fulfil his obligations.

Note that $C_{penalty}$ references $C_{executed}$, which in turn must be aware of $C_{penalty}$. Thus the gateway should be either adjustable (with Alice tuning the gateway with a special transaction) or generic (reports the state of a Findel contract taking its id as an argument).

Defaulting on debt A concise financial DSL does not prevent borrowers from defaulting on their debt. It is up to a marketplace to solve this problem.

Requiring a 100% guarantee deposit seems safe, but is questionable from an economical standpoint. People and organizations borrow money to invest it. The no-arbitrage principle states that there is no guaranteed way to make a profit. The investor reward, e.g. interest, is the premium for taking the inevitable risk of business failure. Thus, this approach hardly makes economical sense.

A marketplace can also mimic the fractional reserve banking model by requiring users to always be able to pay at least $n\%$ of their debt and punishing violators (e.g., by withholding their guarantee deposit). It does not solve the problem of defaults completely though. In legacy finance, users have a fixed government-issued identity, allowing banks to maintain a common database of their credit history. In a decentralized setting, users can create a practically indefinite number of identities. A production-ready marketplace should therefore take measures to combat Sybil attacks.

[13] In this particular case, an equivalent contract $\mathrm{Give}(\mathrm{Or}(\mathrm{One}(USD), \mathrm{One}(EUR)))$ solves the issue. In more complex cases this is not necessarily the case.

Modeling balances with Tokens A more refined approach to modeling users' balances is to use **tokens** – a de-facto standard API [Tok16] for implementing transferable units of value in Ethereum. Tokens are primarily used to represent company shares during so-called initial coin offerings [Ico17]. We assume that tokens can be freely exchanged to any currency the marketplace operates with. Given the address $\mathcal{T}$ of the Ethereum token contract, any Ethereum contract can query the balance of any user U, and transfer its tokens (if it has any) to an arbitrary address. Suppose Alice and Bob are token holders. Alice calls a standard API function `approve` to allow Bob to withdraw a certain amount of tokens from her account. Bob later calls `transferFrom` to transfer the tokens. The transfer succeeds if Alice has enough funds.

We suggest the following procedure. A Findel contract's issuer approves the marketplace with the number of tokens he is potentially liable with. The marketplace implements `enforcePayment` by calling `transferFrom` thus trying to withdraw tokens from the issuer and send them to the owner. Certainly, for the execution to complete, the owner must either have enough tokens in the account, or execute another Findel contract to fill it up. Thus we delegate the banking functionality to the token smart contract and free the marketplace from holding and transferring money [Kho16].

Multi-party contracts We might want to extend the Findel contracts model to support more than two parties. An example of a three-party contract is buying a car with insurance. A user can only buy a car while simultaneously signing an insurance contract. We can express the two contracts (buyer – car dealer, buyer – insurance company) in Findel DSL, but executing them atomically is non-trivial. A possible way would be to use a gateway that keeps track of the state of Findel contracts. If *insuranceSigned* indicates whether a user joined the insurance contract, then buying with insurance looks like this (assuming CAR is a token representing the ownership over a car):

$$\text{If}(\mathit{insuranceSigned}, \text{And}(\text{Give}(\text{Scale}(P, \text{One}(USD)), \text{One}(CAR))), \text{Zero})$$

Local client In order to communicate with a Findel marketplace, users need client-side software. Besides communicating with the Ethereum network, it might also implement other functions:

- Create and store Findel contracts locally.
- Calculate the current value and other properties of Findel contracts based on assumptions about external data (e.g., the € / \$ exchange rate is between 1.0 and 1.2) or valuation techniques such as the lattice binomial model [CRR79].
- Keep track of relevant Findel contracts and perform actions depending on their state (e.g., if c_1 gets executed, join c_2).
- Store a predefined list of addresses of trusted gateways, similar to a list of trusted certificate authorities in web browsers.

3.4 Platform limitations

A Turing-complete programming language does not mean that all a programmer can think of can be implemented inside an Ethereum contract. Gas costs aside, the Ethereum network architecture implies certain limitations.

Lack of precise clock Timing is important for almost all financial contracts. Clock synchronization is a hard problem in decentralized systems, even more so if participants can profit from manipulating timestamps. Blocks in Ethereum are produced every 15 seconds; block timestamps provide causal ordering. Solidity contains keywords for time units, but timestamps are ultimately controlled by miners.

Imperative paradigm Functional programming paradigm is well suited for developing embedded DSLs [Gib13]. The original papers by Peyton Jones et al. as well as all existing implementations of their DSL use functional languages (Haskell [JES00] [JE03] [vS07], OCaml [Lex00], Scala [Wal12] [Cha15]). In contrast, Solidity and Serpent are imperative. Functional languages for Ethereum are in a very early stage of development [FpE17].

Underdeveloped type system Ethereum supports neither decimal nor floating-point types[14], which often model amounts of money and currency exchange rates respectively. The only numeric data types in Solidity are integers of various bit lengths. Moreover, Solidity lacks type parameters, which could be useful for Gateways (i.e., `Gateway<int>`).

4 Gas costs

Every computational step in Ethereum is charged in terms of gas. Despite the use of expensive permanent storage operations, the cost of running our implementation is not prohibitively high for a proof-of-concept.

We measure gas costs of managing common Findel contracts as assessed by the Browser-solidity compiler [Bro16][15] for a marketplace supporting two currencies (referred to as USD and EUR and not tied to any asset). The difference between transaction and execution cost is that the former includes the overhead of creating a transaction (i.e., a call from a client) and the latter does not (i.e., a call from another contract) [Rev16].

4.1 Setup and helper functions

Registering a user implies initializing the user's balances to zero for all supported currencies. For testing purposes, we implement a gateway that uses the current timestamp as data source and calculates a single `keccak256` hash as a dummy authenticity proof.

[14] A likely rationale: rounding issues break consensus.
[15] Solidity version: 0.4.4+commit.4633f3de.Emscripten.clang

Operation	Transaction cost	Execution cost
Create a marketplace smart contract	2221599	1681095
Register a user	79462	58190
Check user's balance	47667	26395
Get contract info	24407	959
Get description info	24706	1258
Update a gateway	36922	15650

Table 3. Cost of setup and helper functions (in gas units)

4.2 Managing common derivatives

In our measurements, we omit cases where parties split the execution cost. We assume that the issuer only pays for contract creation and issuance whereas the owner pays for the execution. For simple Findel contracts, two Ethereum transactions (one from each party) represent the whole lifecycle of a Findel contract. In more complex cases, when a contract executes in multiple steps, we sum up all costs that the owner bears to execute it completely. We also do not account for gateway update costs.

Operation	Create and issue		Join and execute	
	Tx cost	**Exec cost**	**Tx cost**	**Exec cost**
One	184239	177967	58493	93602
Currency exchange (fixed rate)	663149	656877	101878	138430
Currency exchange (market rate)	300842	294570	59822	96196
Zero-coupon bond	373783	367511	143891	201750
Bond with two coupons	939566	933294	346871	477100
European option	519628	513356	278191	411103
Binary option	402359	396087	59826	96204

Table 4. Cost of handling Findel contracts for common derivatives (in gas units)

As of January 2017, the gas cost 10^{-9} ether per unit [Eth17]; the price of ether fluctuated around \$10 [Wor17]. That brings the cost of a typical Findel contract operation (10^5 – 10^6 gas units) to 1.8 – 18 US cent.

5 Related work

[Sch13] and [Hvi10] review financial DSLs and related projects. [STM16] and [CBB16] explore approaches to smart contract programming languages.

5.1 Composable contracts by Peyton Jones et al.

Our work is inspired by the composable contracts as defined in [JE03], from which we borrow some of our primitives (Zero, One, Scale, And, Or). It turns out though that this notation is not directly transferable to blockchain environments (at least to Ethereum) due to the way it formalizes temporal conditions (*when*, *until*). Blockchains differ substantially from traditional centralized marketplaces in how they model conditions. For this reason we introduced If and Timebound primitives to express causal and temporal conditions respectively.

5.2 Logic Portfolio Theory by Steffen Schuldenzucker

Steffen Schuldenzucker in [Sch16] proposes an axiomatic approach to proving no-arbitrage relationships between contracts based on the notation from [JE03]. Using a rigorously defined algebra of contracts, he proves well-known financial theorems, such as the put-call parity. Formal semantics of Findel can be introduced using a similar approach. This would enable formal verification techniques that could substantially increase confidence in the safety of our language.

5.3 Preliminary draft by Nick Szabo

Smart contracts pioneer Nick Szabo in [Sza02] presents "a mini-language" that can be characterized as a middle ground between programming and legal speak. The basic building block is a *right* (e.g., to receive \$100 now). Rights are combined using well-defined operators (*when*, *then*, *also*, *with* – analogous to our primitives) and *performed* depending on external events. Parties are assumed to have a trusted source of real-world information. The language is not purely declarative: contracts may perform calculations and save values in state variables, which allows for more flexibility[16].

6 Conclusion

Smart contracts in public blockchain networks seem to be a perfect match for modeling financial agreements. Their unique value proposition is trustless execution, which reduces counterparty risks. We introduced Findel – a declarative financial DSL built upon ideas from previous research in financial engineering. Formalizing contract clauses using Findel makes them unambiguous and machine-readable. We proved Ethereum to be a suitable platform for trading and executing Findel contracts.

Nevertheless, the whole smart contract field is still in its infancy. Programmers who wish to implement a usable smart contract for handling financial agreements need to be aware of the forthcoming challenges: from fundamental limitations of the blockchain network architecture to imperfect development environment.

[16] Szabo makes a case against state variables in general, stating that "they should be avoided unless utterly necessary".

References

ABC16. Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts. *IACR Cryptology ePrint Archive*, 2016:1007, 2016.

Bis16. Statistical release. otc derivatives statistics at end-june 2016, 2016. `https://www.bis.org/publ/otc_hy1611.pdf`.

Bro16. Browser-solidity online compiler, 2016. `https://ethereum.github.io/browser-solidity/`.

But14. A next-generation smart contract and decentralized application platform, 2014. `https://github.com/ethereum/wiki/wiki/White-Paper`.

CBB16. Christopher D. Clack, Vikram A. Bakshi, and Lee Braine. Smart contract templates: foundations, design landscape and research directions. *CoRR*, abs/1608.00771, 2016.

CF14. Sophia Yakoubov Conner Fromknecht, Dragos Velicanu. A decentralized public key infrastructure with identity retention. Cryptology ePrint Archive, Report 2014/803, 2014. `http://eprint.iacr.org/2014/803`.

Cha15. Shahbaz Chaudhary. Adventures in financial and software engineering, 2015. `https://falconair.github.io/2015/01/30/composingcontracts.html`.

CRR79. John C Cox, Stephen A Ross, and Mark Rubinstein. Option pricing: A simplified approach. *Journal of financial Economics*, 7(3):229–263, 1979.

dC16. Michael del Castillo. Jp morgan, credit suisse among 8 in latest bank blockchain test, 2016. `http://www.coindesk.com/jp-morgan-credit-suisse-among-8-in-latest-bank-blockchain-test/`.

Eth17. Ethstats, 2017. `https://ethstats.net/`.

FpE17. Functional programming for ethereum, 2017. `https://github.com/fp-ethereum/fp-ethereum`.

FSNB09. Simon Frankau, Diomidis Spinellis, Nick Nassuphis, and Christoph Burgard. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming*, 19(01):27–45, 2009.

Gai11. Jean-Marie Gaillourdet. A software language approach to derivative contracts in finance. 2011. `http://ceur-ws.org/Vol-750/yrs06.pdf`.

Gib13. Jeremy Gibbons. Functional programming for domain-specific languages. In *CEFP*, volume 8606 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2013.

Gre16. Gideon Greenspan. Why many smart contract use cases are simply impossible, 2016. `http://www.coindesk.com/three-smart-contract-misconceptions/`.

Hvi10. Tom Hvitved. A survey of formal languages for contracts. In *Fourth Workshop on Formal Languages and Analysis of Contract–Oriented Software (FLACOS10)*, pages 29–32. Citeseer, 2010.

Ico17. Icos, token sales, crowdsales, 2017. `https://www.smithandcrown.com/icos/`.

JE03. Simon L. Peyton Jones and Jean-Marc Eber. How to write a financial contract. *The Fun of Programming*, 2003.

JES00. Simon L. Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: an adventure in financial engineering, functional pearl. In *ICFP*, pages 280–292. ACM, 2000.

Kho16. Dmitry Khovratovich. debt.sol, 2016. `https://gist.github.com/khovratovich/45f68082b556b45eb64e8e1c3eb82892`.

LC16. Karen Lewison and Francisco Corella. Backing rich credentials with a blockchain pki, 2016. `https://pomcor.com/techreports/BlockchainPKI.pdf`.

Lex00. Ocaml at lexifi, 2000. `https://www.lexifi.com/blogs/ocaml`.

Mor16. Sofus Mortensen. Universal contracts, 2016. `https://github.com/corda/corda/tree/master/experimental/src`.

Ora16. Oraclize, 2016. `http://www.oraclize.it/`.

Rea16. Reality keys, 2016. `https://www.realitykeys.com/`.

Rev16. Raine Rupert Revere. What is the difference between transaction cost and execution cost in browser solidity?, 2016. `https://ethereum.stackexchange.com/q/5812/5113`.

Sch13. Todd Schiller. Financial domain-specific language listing, 2013. `http://www.dslfin.org/resources.html`.

Sch14. Steffen Schuldenzucker. Decomposing contracts. 2014. `http://www.ifi.uzh.ch/ce/people/schuldenzucker/decomposingcontracts.pdf`.

Sch16. Steffen Schuldenzucker. An axiomatic framework for no-arbitrage relationships in financial derivatives markets. 2016. `http://www.ifi.uzh.ch/ce/publications/LPT.pdf`.

Sir16. Emin Gün Sirer. Thoughts on the dao hack, 2016. `http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/`.

STM16. Pablo Lamela Seijas, Simon Thompson, and Darryl McAdams. Scripting smart contracts for distributed ledger technology. Cryptology ePrint Archive, Report 2016/1156, 2016. `http://eprint.iacr.org/2016/1156`.

Sza97. Nick Szabo. Formalizing and securing relationships on public networks, 1997. `http://journals.uic.edu/ojs/index.php/fm/article/view/548`.

Sza02. Nick Szabo. A formal language for analyzing contracts, 2002. `http://nakamotoinstitute.org/contract-language/`.

Tls16. Tlsnotary, 2016. `https://tlsnotary.org/`.

Tok16. Ethereum improvement proposal: Token standard, 2016. `https://github.com/ethereum/EIPs/issues/20`.

vS07. Anton van Straaten. Composing contracts, 2007. `https://web.archive.org/web/20130814194431/http://contracts.scheming.org`.

Wal12. Channing Walton. Scala contracts project, 2012. `https://github.com/channingwalton/scala-contracts/wiki`.

Woo14. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2014. `http://gavwood.com/paper.pdf`.

Wor16. Gross domestic product 2015, 2016. `http://databank.worldbank.org/data/download/GDP.pdf`.

Wor17. Worldcoinindex, 2017. `https://www.worldcoinindex.com/coin/ethereum`.

A Examples

– A **fixed-rate currency exchange**: the owner sells €10 for $11.

$$\text{And}(\text{Give}(\text{Scale}(10, \text{One}(EUR))), \text{Scale}(11, \text{One}(USD))$$

– A **market-rate currency exchange**: the owner sells €10 at market rate as reported by the gateway at *addr*.

$$\text{Scale}(10, \text{And}(\text{Give}(\text{One}(EUR)), \mathit{ScaleObs}(addr, \text{One}(USD))))$$

– A **zero-coupon bond**: the owner receives \$100 at t_0.

$$\text{Timebound}(t_0 - \delta, t_0 + \delta, \text{Scale}(100, \text{One}(USD)))$$

– A **bond with coupons**: the owner receives \$1000 (face value) in three years (maturity date) and two coupon payments of \$50 at regular intervals before the maturity date.

$$\text{And}(\text{At}(\text{now} + 3 \text{ years}, c_{face}), \text{And}(\text{At}(\text{now} + 1 \text{ years}, c_{cpn}), \text{At}(\text{now} + 1 \text{ years}, c_{cpn})))$$

where

$$c_{face} = \text{Scale}(1000, \text{One}(USD)), \quad c_{cpn} = \text{Scale}(50, \text{One}(USD))$$

– A **future** (a **forward**[17]): parties agree to execute the underlying contract c at t_0.

$$\text{Timebound}(t_0 - \delta, t_0 + \delta, c)$$

– An **option**: the owner can choose at (European option) or before (American option) time t_0 whether to execute the underlying contract c.

$$\text{Timebound}(t_0 - \delta, t_0 + \delta, \text{Or}(c, \text{Zero}))$$

$$\text{Timebound}(now, t_0 + \delta, \text{Or}(c, \text{Zero}))$$

– A **binary option**: the owner receives \$10 if a predefined event took place at t_0 and nothing otherwise.

$$\text{If}(addr, \text{Scale}(10, \text{One}(USD)), \text{Zero})$$

[17] In traditional finance, a future is a standardized contract while a forward is not. This distinction is not relevant for our model.

Decentralized Execution of Smart Contracts: Agent Model Perspective and Its Implications

Lin Chen, Lei Xu, Nolan Shah, Zhimin Gao, Yang Lu, and Weidong Shi

Department of Computer Science, University of Houston, TX 77054, USA

Abstract. Smart contracts are one of the most important applications of the blockchain. Most existing smart contract systems assume that for executing contract over a network of decentralized nodes, the outcome in accordance with the majority can be trusted. However, we observe that users involved with a smart contract may strategically take actions to manipulate execution of the contract for purpose to increase their own benefits. We propose an agent model, as the underpinning mechanism for contract execution over a network of decentralized nodes and public ledger, to address this problem and discuss the possibility of preventing users from manipulating smart contract execution by applying principles of game theory and agent based analysis.

Keywords: Smart Contract, Blockchain, Public Ledger, Game Theory

1 Introduction

In recent years, there have been papers and articles focusing on improving our understanding of blockchain based crypto-currency using game theory [7, 8, 16]. The assumption behind these crypto-currency systems, e.g., Bitcoin, is that participating users are financially driven. If a user has no interest in gaining rewards from the system (e.g., mining, executing contract), he/she has no incentive of staying in the system. Therefore, users should not be considered as merely machines that have resources to execute the protocols of such system. By nature, they are more like players/economic agents who attempt to maximize their profits through participation. This motivates the use of game theory to study blockchain-based smart contract and transaction systems. For instance, in this line of research, a recent paper of Kiayias et al. studies mining as a game in Bitcoin and analyzes the best strategy for users [7]. However, little research has been done for understanding the behaviors of smart contract execution over decentralized blockchain and public ledger under agent based model, which is the main focus of this paper.

As such, we consider the strategic behavior of users in smart contracts. Briefly speaking, a smart contract is a computerized transaction protocol that executes the terms of a contract [19]. It could be viewed as a counterpart to a physical-world contract in a decentralized system. Like a contract in the physical world, a smart contract may specify different conditions and define the payoffs for users under each condition. The following is a simple example: if a random dice returns 0, then A pays one coin to B; if it returns 1, then B pays A one coin. Though electronic commerce applications or contracts can be supported using centralized systems, smart contract mostly relies on decentralized network of participants where no single participant is necessarily trusted. A hallmark of smart contracts is that enforcement is achieved through consensus.

A smart contract can involve multiple users/participants and large amounts of crypto-currency. Thus, it has the potential to be more critical than mining in pure crypto-currency systems (e.g., Bitcoin), in which only a fixed reward is paid to successful miners. The amount of crypto-currency involved in a contract may be many times and significantly higher than the cost of running the contract itself. Therefore, users involved in a smart contract may strategically take actions to maximize their own profits, which can cause significant problems and cast doubt to the fundamental assumption of smart contract execution model based on consensus or majority accepted outcome.

Considering the example mentioned above, suppose that A represents a set of users. If the random dice returns 0, A has the incentive of lying and claiming that it returns 1, and plays strategically according to the protocols of the system. If the system applies Byzantine agreement protocols or alike to reach consensus, then A plays as the set of malicious nodes in the Byzantine problem who attempt to prevent a consensus on 0 (i.e., A tries to impose a consensus on the wrong value 1 or prevent the entire system from reaching a consensus at all). If the system allows temporary branches and uses the longest chain rule to eventually resolve branches, then A adds a block containing the wrong value of the dice and tries to make it into the longest chain. The strategies that A may take are dependent on the protocols of the system. In this paper, we do not necessarily restrict our attention to one specific protocol or one specific embodiment of smart contract system. Therefore we do not specify the actions of A but rather say whether A lies or not. When we say A lies, we mean A plays strategically to produce contract execution outcome that favors him/her financially regardless the true result of the contract. Otherwise, we say A does not lie or A tells the truth - always producing or accepting the outcome based on truthful execution of the contract. The goal of this paper is to discuss the possibility and feasible strategies strategies to prevent users involved in smart contracts from lying or manipulating contract execution outcome for personal financial gains.

It is worth pointing out that the risk of accepting the rogue outcome of contract execution increases when a large percentage of nodes of a smart contract system have direct or indirect financial involvement in a smart contract. Even for contracts only directly involving few or just two participants, there is a possibility that a subset of these directly involved participants can manipulate the outcome by creating dependent contracts that distributes financial rewards to other nodes of the system if they accept certain contract execution result, a form of bribing in contract execution and outcome confirmation. There is no trivial solution or prevention mechanism to this problem. In the worst case, every node may have either direct or indirect conflict of interests in terms of contract execution. In addition, the anonymous nature of smart contract users/accounts and crypto-currency wallets make it almost impossible to detect conflict of interests when comes to contract execution.

Our contributions. We suggest that participants of a smart contract based system using blockchain and public ledger be considered as economic agents. As a consequence, execution of smart contract over a network of untrusted nodes using blockchain is better to be understood and studied under the framework of agents with the assumption that their participation is motivated by self-interests and financial benefits. When participants of a smart contract system (e.g., miners, nodes for executing contracts) are involved in a smart contract, they may have incentives and engage in negative behaviors (e.g., lying or manipulation) to maximize their own interests. These include producing or accepting contract execution outcome that favors themselves by ignoring or discarding results of truthful execution of the contract. Furthermore, we discuss the feasibility of preventing such behaviors through proper design of smart contract based systems.

We show that, in general, there is no guaranteed way to prevent users from lying or engaging in bad behaviors in a smart contract system, and there exist scenarios where lying on outcome of contract execution could be the dominant strategy for a user (i.e., the user will lie regardless of the actions of other users). To solve this problem, we introduce payment in the game, that is, we discuss the scheme that can penalize a node by fining him/her some amount of coins if the result of a smart contract execution is different from that of the majority. This is a straightforward approach that works for many problems in game theory. However, we show that, if all users are not only rational but also fall into a class called *superrationality*, then there exist scenarios in which they will always lie or behave badly regardless of how high the penalty or fine would be.

Our negative results rely heavily on the rationality assumption of the users and participants of a smart contract system. However, rationality is a debatable concept in game theory. There exists a line of research focusing on irrational behaviors of people. It suggests that a person, even with perfect rationality of himself/herself, might not fully trust the rationality of others. We show that the problem changes significantly if we assume that users are not fully confident in the rationality of others. We also characterize the amount of the penalty that can prevent users from lying on contract execution outcome given that the users' belief in the rationality of others is reflected by some known probability distribution.

The remainder of the paper is organized as follows: In Section 2 we give a short review of smart contract and describe the problem we address in this paper. Section 3 describes the agent model for smart contract execution over a network of decentralized participants and the role of penalty. In Section 4 we discuss the way to implement penalty in decentralized smart contract execution environment. Section 5 discusses related work, and we conclude the paper in Section 6.

2 Smart Contract and Problem Statement

We begin by defining smart contracts. The definition provided by Szabo in 1997 [18] is:

Definition 1. *A smart contract is a set of promises, specified in a digital form, including protocols within which the parties perform on these promises.*

However, this definition potentially covers a broad range of already existing centralized and client-server based e-commerce systems (e.g., Ebay), which fundamentally distinguishes from blockchain based smart contracts that rely on a decentralized network of untrusted nodes/participants and crypto-currency (e.g., Ethereum [1]). Blockchain can enforce smart contracts in a decentralized way without assuming any single trusted party. This is especially attractive in scenarios where users involved in a contract do not necessarily trust each other. As long as the entire blockchain system is "trusted" as a whole, it is guaranteed that execution results of a smart contract could be trustworthy. Most of existing works assume that when the majority of participating nodes in a blockchain system are honest, the system is trusted.

However, the situation is more complex in reality. Each node of the blockchain may adopt different action strategies for different smart contracts to maximize their own interests. This makes smart contract execution process more like an economic game. We use the definition of a normal form game by Osborne [13]:

Definition 2. *A* normal form game Γ consists of:

- A finite set N of players (agents).
- A nonempty set Q_i of strategies available for each player $i \in N$.
- A preference relation $\preceq_i$ on $Q = \times_{j \in N} Q_j$ for each player i.

We restrict our attention to normal form games in this paper. For simplicity, when we say a game, we mean a normal form game.

A strategy $q_i \in Q_i$ is called a (weakly) dominant strategy for player i if no matter what strategies are chosen by other players, choosing q_i always gives i an outcome that is not worse than any other strategy.

The agent model for smart contract. We consider the following model, which we call an agent model for smart contracts. There is a smart contract which involves N users (players). Each user j has a weight w_j. The smart contract specifies a set of possible future states of the system, depending on which each user either gains or loses coins (crypto-currency). For simplicity we assume that there are only two possible states S_0 and S_1. If a state S_i occurs ($i = 0$ or 1), user j will get z_j^i coins (specifically, if $z_j^i < 0$, then it means that user j loses $-z_j^i$ coins). Once the smart contract starts to be executed, the state of the system is unique and clear to all users/participants, and we call this state as the true state. In a decentralized system for contract execution and confirmation, however, all the users shall agree to a certain state based on which the smart contract is executed; and this state may not necessarily be the true state because of the

agent assumption. We assume that every user will vote for/accept one state, and if users who vote for/accept a certain state S_i have a total weight at least αW where $W = \sum_{j=1}^{N} w_j$, then the smart contract will be executed based on the state S_i. We discuss, under the described agent model, the possibility of preventing users from lying on contract execution outcome by voting for/accepting incorrect state.
Remark on the model. A user may have different identities (pseudonyms) in a public blockchain based smart contract system. For simplicity, in this paper, we assume that each user owns exactly one identity, whereas identities and users are used interchangeably. Depending on the protocols used in a blockchain based contract system, parameters may have different meanings. For example, if the system uses proof of work and longest chain rule (e.g., Bitcoin), then w_j corresponds to the computation power of user j, and voting for a state S_i means generating a block that executes the smart contract based on S_i (this may yield a branch, though), and keeping adding blocks to make it into the longest chain. For ease of presentation, we assume that there are only two possible states S_0 and S_1. However, our result can be easily extended to the case where there are more possible states.

3 An Agent Model for Smart Contract Execution with Penalty

We start with the following simple observation.

Observation 1 *In the agent model, voting for the state that the user most prefers is the dominant strategy.*

Consider an arbitrary scenario in which every user votes for S_0 or S_1. If user j prefers S_1 most and does not vote for S_1, then he/she can simply switch and vote for S_1 instead. Switching only decreases the utility of j if originally S_1 is the state based on which the smart contract is executed, and after switching it becomes S_0. However, this is impossible. Hence the observation is true. Note that if S_1 is not the true state, then user j always lies.

A common approach that prevents agents from lying in a game is to introduce payments. We consider the most straightforward way of adding the payment to the agent model, that is, if a user votes for a state that is different from the state based on which the smart contract is executed, he/she will be penalized, i.e., he/she will be fined a certain amount of coins.

Adding payment might prevent some users from lying on execution outcome. Specifically, if the number of extra coins that a user gets by outputting wrong outcome or lying is less than the penalty, he/she may choose to vote for the true state. However, it is still possible that users are lying no matter how large the penalty is. Consider the following scenario: The true state is S_0. There are users who strictly prefer S_1 than S_0. Let U be the set of them and suppose $\sum_{j \in U} w_j \geq \alpha W$. Focusing on users in U, there are two Nash equilibria, every user in U voting for S_0 or every user in U voting for S_1. Consider an arbitrary user $j \in U$. When making his/her own decision, user j guess the decisions of other players. If j is optimistic and assumes every other player in U are voting for S_1, he/she will vote for S_1, otherwise if he/she is pessimistic and assumes every other player in U are voting for S_0, he/she will vote for S_0. In such a scenario, users may still lie. Furthermore, we have the following claim.

Theorem 2. *In the agent model with penalty, if j is superrational and knows that $\sum_{j \in U} w_j \geq \alpha W$, then no matter how high the penalty is, j will always lie.*

We provide the definition of superrationality as follows.

Definition 3 ([6]). *A player (agent) is called superrational if he/she has perfect rationality (and thus maximize his/her own utility), assumes that all other players are superrational, and that a superrational player will always come up with the same strategy as any other superrational player when facing the same problem.*

We remark that, superrationality is also called renormalized rationality in literature. According to the definition, if j is superrational, then he/she assumes that any other user in U would behave in the same way as he/she does, in this case, he/she will always vote for S_1, hence Theorem 2 is true.

Our above arguments show that, in general, introducing payment does not prevent users from lying. There exist scenarios in which users lie regardless of how high the penalty is. However, superrationality or rationality may not apply to real world application scenarios. As we have discussed, the incentive of lying relies crucially on a user's belief in certain behaviors of others. Specifically, he/she believes that other users are all rational. However, rationality itself is one of the most debatable issues in game theory in the sense that it seems to contradict a lot of laboratory experiments, which suggests that people often fail to conform to some of the basic assumptions of rationality. The "Centipede Game" , which was constructed by Rosenthal [15] in 1982, is one of the most well-known examples that illustrate such a phenomenon.

The centipede game is carried out between two players, say, A and B in a fixed number of rounds which is known to both players. Initially both A and B own 1 coin. At the beginning of round i, let a_i and b_i be the number of coins owned by A and B respectively. If i is odd, A makes the decision of yes or no, otherwise, B makes the decision. If A or B decides on yes, then the game moves to round $i+1$, $a_{i+1} = a_i + 1$, $b_{i+1} = b_i + 1$. If A or B decides on no, then the game stops. If it is A that decides on no (i.e., i is odd), then $a_{i+1} = a_i + 2$, $b_{i+1} = b_i - 1$. Otherwise it is B that decides on no, then $a_{i+1} = a_i - 1$, $b_{i+1} = b_i + 2$.

Assuming that A is rational and he/she believes the rationality of B, then A will decide on no at round 1 and the centipede game ends at the beginning. The reasoning is that at the last round regardless of whose turn it is, the decision will be no. Therefore, at the second to last round the opponent will decide no to make sure that the number of his/her coins does not decrease. Iteratively carrying out this argument we get the conclusion. However, this does not coincide with the experiment results. For example, McKelvey and Palfrey [10] reported that only 15% of the players chose to end the game at the beginning in the experiments they carried out. That means, in most of these experiments, people do exhibit behaviors that contradict the traditional rationality assumptions in game theory. More experimental results and discussions on the centipede game and irrationality could be found in [11, 20].

The experimental results suggest that people often do not have fully trust in the rationality of the others. Notice that even if player A has perfect rationality, however, if he/she does not believe in the rationality of B, then A may still choose to continue the centipede game. Users involved in a smart contract may encounter a similar situation. Consider user $j \in U$, whether j votes for S_1 or not depends on his/her belief in the other users. Following the studies on irrationality in centipede game [2], we define the parameter $\tau_j(k)$, which indicates user j's belief in a certain behavior of user k, that is, user j believes that with probability $\tau_j(k)$, user k will vote for S_1, and with probability $1-\tau_j(k)$, user k will vote for S_0. Based on such assumptions, user j's decision is based on the following.

For $k \neq j$, we define X_k as a 0-1 random variable such that:

$$Pr(X_k = 1) = \tau_j(k), \quad Pr(X_k = 0) = 1 - \tau_j(k).$$

Suppose user j votes for S_1, then based on j's belief, the probability that the smart contract is executed based on S_1 is $Pr(\sum_{k\neq j} X_k + w_j \geq \alpha W)$. Let p_j be the penalty if the smart contract is executed based on S_0, then the expected reward of j by lying (voting for S_1) is

$$\begin{aligned} &z_j^1 Pr(\sum_{k\neq j} w_j X_j \geq \alpha W - w_j) - p_j(1 - Pr(\sum_{k\neq j} X_j \geq \alpha W - w_j)) \\ =\ &(z_j^1 + p_j) Pr(\sum_{k\neq j} w_j X_j \geq \alpha W - w_j) - p_j \end{aligned}$$

The expected reward of j by telling the truth is

$$z_j^0 Pr(\sum_{k\neq j} w_j(1 - X_j) \geq \alpha W - w_j) = z_j^0 Pr(\sum_{k\neq j} w_j X_j \leq (1-\alpha)W)$$

Therefore, as long as

$$z_j^0 Pr(\sum_{k\neq j} w_j X_j \leq (1-\alpha)W) \geq (z_j^1 + p_j) Pr(\sum_{k\neq j} w_j X_j \geq \alpha W - w_j) - p_j,$$

is true, the rational user j will not lie. This means, if j does not fully believe in the rationality of other users, then sufficient penalty can prevent j from lying. Overall, the following is true:

Theorem 3. *In the agent model with penalty, if a user does not fully believe in the rationality of others, then a sufficient penalty can prevent him/her from outputting incorrect contract execution outcome or lying.*

4 Implementation of Contract Execution with Penalty

Penalty plays a central role in the agent model of smart contract execution as shown in the previous section's analysis. We discuss the enforcement of penalty in this section.

There are several strategies to eliminate disagreement in blockchain branches. These strategies are also used to determine smart contract execution results when there is disagreement. Common rules include longest-chain which is used by Bitcoin [12], and GHOST which is used by Ethereum [17]. No matter what strategy is used, we add following functions to support penalty in a decentralized smart contract system:

- Recording users' choices. Existing blockchain systems usually records only one identity for each block and ignores supporters of the block. Recording supporters is necessary for implementing penalty schemes. When a user accepts a block, he/she should generate a signature of the block and broadcast it to the network. Therefore, everyone can track users' choices of the smart contract execution outcome;
- Distribution of penalty. When a group of users supporting the wrong result need to be penalized, users supporting the correct result can submit a penalty request to the blockchain. The collected fine is distributed to them.

5 Related Work

We provide a brief overview on blockchain based smart contract and game theory studies on these systems.

Ethereum is the most popular smart contract system [1]. It is based on proof-of-work, but is planning to move to proof-of-stake. Luu et.al. proposed a formal method to analyze Ethereum smart contracts to detect potential vulnerabilities [9].

The consequence of decentralization is subtle. Garay [5] and Pass et al. [14] showed that, several important security properties defined in the work of Nakamoto [12] are true, given the assumption that the majority of mining power in the Bitcoin system is controlled by the honest miners. Without such an assumption, however, security is not guaranteed. However, the assumption itself is questionable. For example, in 2014, the mining pool GHash.io exceeded 50% of the computational power in Bitcoin [3]. Thus, it becomes important to understand the behavior of users that participate in the system and study mechanisms that would motivate them to behave in an honest way.

There are a series of studies focusing on game theory aspects of users involved in mining. From a game theory perspective, Eyal and Sirer [4] showed that even a majority of honest miners is not enough to guarantee the security of the Bitcoin protocol. Sapirshtein et al. [16] and Kiayias et al. [7] studies mining as a game in Bitcoin and analyzes the best strategy of users.

6 Conclusion and Future Work

In this paper, we establish an agent based framework to model smart contract execution over a decentralized network of nodes/participants using blockchain and public ledger. In contrast to the commonly accepted assumption that smart contract execution outcome accepted by the majority can be trusted, agent based model of smart contract execution assumes that nodes may have incentive to manipulate or lie on outcome of contract execution in return for personal benefits or financial gains even they are not directly involved in a contract. We observe that users who are directly or indirectly involved in a smart contract may strategically take actions to manipulate smart contract execution outcome (e.g., produce or accept outcome that favors their own interests). In accordance with agent based model, we discuss the possibility of preventing users from engaging in bad behaviors in terms of contract execution or lying on contract outcome. We provide negative results for general smart contract execution models. We also show that if penalty is introduced in contract execution and assume that users are not fully confident in the rationality of other participants, then it is plausible to prevent users from lying on outcome or manipulating result of contract execution. Furthermore, we believe that, irrationality is an important subject that would contribute to better understanding of user behaviors in a decentralized cryptocurrency or smart contract system. A systematic investigation of irrationality in the context of smart contract execution and consensus is an important open problem. Another interesting open problem is whether it is possible to use other mechanisms, rather than financial penalty, to prevent users from lying on contract outcome when it favors them the most.

References

1. Buterin, V.: A next-generation smart contract and decentralized application platform. white paper (2014)
2. Dunbar, G., Tu, J., Wang, R., Wang, X., et al.: Rationalizing irrational beliefs. Queens Economics (2006)
3. Duong, T., Fan, L., Zhou, H.S.: 2-hop blockchain: Combining proof-of-work and proof-of-stake securely (2016)
4. Eyal, I., Sirer, E.G.: Majority is not enough: Bitcoin mining is vulnerable. In: International Conference on Financial Cryptography and Data Security. pp. 436–454. Springer (2014)
5. Garay, J., Kiayias, A., Leonardos, N.: The bitcoin backbone protocol: Analysis and applications. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 281–310. Springer (2015)
6. Hofstadter, D.R.: Dilemmas for superrational thinkers, leading up to a luring lottery. Scientific American 6, 267–275 (1983)
7. Kiayias, A., Koutsoupias, E., Kyropoulou, M., Tselekounis, Y.: Blockchain mining games. In: Proceedings of the 2016 ACM Conference on Economics and Computation. pp. 365–382. ACM (2016)
8. Lewenberg, Y., Bachrach, Y., Sompolinsky, Y., Zohar, A., Rosenschein, J.S.: Bitcoin mining pools: A cooperative game theoretic analysis. In: Proceedings of the

2015 International Conference on Autonomous Agents and Multiagent Systems. pp. 919–927. International Foundation for Autonomous Agents and Multiagent Systems (2015)

9. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. ACM (2016)
10. McKelvey, R.D., Palfrey, T.R.: An experimental study of the centipede game. Econometrica: Journal of the Econometric Society pp. 803–836 (1992)
11. McKelvey, R.D., Palfrey, T.R.: Quantal response equilibria for extensive form games. Experimental economics 1(1), 9–41 (1998)
12. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
13. Osborne, M.J., Rubinstein, A.: A course in game theory. MIT press (1994)
14. Pass, R., Seeman, L., Shelat, A.: Analysis of the blockchain protocol in asynchronous networks. IACR Cryptology ePrint Archive 2016, 454 (2016)
15. Rosenthal, R.W.: Games of perfect information, predatory pricing and the chain-store paradox. Journal of Economic theory 25(1), 92–100 (1981)
16. Sapirshtein, A., Sompolinsky, Y., Zohar, A.: Optimal selfish mining strategies in bitcoin. arXiv preprint arXiv:1507.06183 (2015)
17. Sompolinsky, Y., Zohar, A.: Secure high-rate transaction processing in bitcoin. In: International Conference on Financial Cryptography and Data Security. pp. 507–527. Springer (2015)
18. Szabo, N.: Formalizing and securing relationships on public networks. First Monday 2(9) (1997)
19. Tapscott, D., Tapscott, A.: Blockchain Revolution: How the Technology Behind Bitcoin Is Changing Money, Business, and the World. Penguin (2016)
20. Zauner, K.G.: A payoff uncertainty explanation of results in experimental centipede games. Games and Economic Behavior 26(1), 157–185 (1999)

A Concurrent Perspective on Smart Contracts

Ilya Sergey[1] and Aquinas Hobor[2]

[1] University College London, United Kingdom
`i.sergey@ucl.ac.uk`
[2] Yale-NUS College and School of Computing, National University of Singapore
`hobor@comp.nus.edu.sg`

Abstract. In this paper, we explore remarkable similarities between multi-transactional behaviors of smart contracts in cryptocurrencies such as Ethereum and classical problems of shared-memory concurrency. We examine two real-world examples from the Ethereum blockchain and analyzing how they are vulnerable to bugs that are closely reminiscent to those that often occur in traditional concurrent programs. We then elaborate on the relation between observable contract behaviors and well-studied concurrency topics, such as atomicity, interference, synchronization, and resource ownership. The described *contracts-as-concurrent-objects* analogy provides deeper understanding of potential threats for smart contracts, indicate better engineering practices, and enable applications of existing state-of-the-art formal verification techniques.

1 Introduction

Smart contracts are programs that are stored on a blockchain, a distributed Byzantine-fault-tolerant database. Smart contracts can be triggered by blockchain transactions and read and write data on their blockchain [38]. Although smart contracts are run and verified in a distributed fashion, their semantics suggest that one can think of them as of *sequential* programs, despite the existence of a number of complex interaction patterns including *e.g.*, reentrancy and recursive calls. This mental model simplifies both formal and informal reasoning about contracts, enabling immediate reuse of existing general-purpose frameworks for program verification [5,16,31,32] that can be employed to verify smart contracts written in *e.g.* Solidity [15] with only minor adjustments.

Although all computations on a blockchain are deterministic,[3] a certain amount *non-determinism* still occurs due to races between transactions themselves (*i.e.* which transactions are chosen for a given block by the miners). We will show in that non-determinism can be exploited by adversarial parties and makes reasoning about contract behavior particularly subtle, reminiscent to known challenges involved in conventional concurrent programming.

In this paper we outline a model of smart contracts that emphasizes the properties of their *concurrent* executions. Such executions can span *multiple*

[3] This requirement stems from the way the underlying Byzantine distributed ledger consensus protocol enables all involved parties to agree on transaction outcomes.

blockchain transactions (within the same block or in multiple blocks) and thereby violate desired safety properties that cannot be stated using only the contract's implementation and local state—precisely what the existing verification methodologies focus on [5, 32]. To facilitate the reuse of the common programming intuition, we propose the following analogy:

Accounts using smart contracts in a blockchain
are like
threads using concurrent objects in shared memory.

Threads using concurrent objects in shared memory. By *concurrent objects* we mean the broad class of data structures that are employed to exchange data between and manage the interaction of multiple *threads* (processes) running concurrently [20]. Typical examples of concurrent objects are locks, queues, and atomic counters—typically used via popular libraries such as `java.util.concurrent`. At runtime, these concurrent objects are allocated in a block of *shared memory* that is accessible to the running threads. The behavior resulting from the threads accessing the objects simultaneously—*i.e. interference*—can be extremely unpredictable and thus extremely difficult to reason about.

Concurrent objects whose implementation does not utilize proper synchronization (*e.g.*, with *locks* or *barriers*) can manifest *data races*[4] under interference leading to a loss of memory integrity. Even for race-free objects the observed behavior under interference may be erroneous from the perspective of one or more clients. For example, a particular thread may not "foresee" the actions taken by the other threads with a shared object and thus may not expect for that object to change in all of the ways that it does change under interference.

Accounts using smart contracts in a blockchain. Smart contracts are analogous to concurrent objects. Instead of residing in a shared memory they live in the blockchain; instead of being used by threads they are invoked by *accounts* (users or other contracts). Like concurrent objects, they have internal mutable state, manage resources (*e.g.* funds), and can be accessed by multiple parties both within a block and in multiple blocks. Unlike traditional concurrent objects, a smart contract's methods are atomic due to the transactional model of computation. That is, a single call to a contract (or a chain of calls to a series of contracts calling each other), is executed *sequentially*—without interrupts—and either terminates after successfully updating the blockchain or aborts and rolls back to its previous configuration before the call.

The notion of "atomicity for free" is deceptive, however, as concurrent behavior can still be observed *at the level of the blockchain*:

– The order of the transactions included to a block is not determined at the moment of a transaction execution, and, thus, the outcome can largely depend on the ordering with respect to other transactions [27].

[4] That is, unsynchronized concurrent accesses by different threads to a single memory location when at least one of those accesses is a write.

– Several programming tasks require the contract logic to be spread across several blockchain transactions (*e.g.*, when contracts "communicate" with the world outside of the blockchain), enabling true concurrent behavior.
– Calling other contracts can be considered to be a kind of *cooperative multitasking.* By cooperative multitasking we mean that multiple threads can run but do not get interrupted unless they explicitly "yield". That is, a call from contract A to contract B can be considered to be a yield from contract A's perspective, with contract B yielding when it returns. The key point for smart contracts is that **contract B can run code that was unanticipated by contract A's designer**, which makes the situation much closer to a concurrent setting than a typical sequential one.[5] In particular, contract B can modify state that contract A may assume is unchanged during the call. This is the essence of The DAO bug [9], in which contract B made a call back into contract A to modify A's local state before returning [27]. However, reentrancy is not the only way this kind of error can manifest, since:
– It is not difficult to imagine a scenario in which a certain contract is used as a *service* for other parties (users and contracts), managing the access to a shared resource and, in some sense, serving as a concurrent library. As multi-contract transactions are becoming more ubiquitous, various interference patterns can be observed and, thus, should be accounted for.

Our goals and motivation. Luckily, the research in concurrent and distributed programming conducted in the past three decades provides a large body of theoretical and applied frameworks to code, specify, reason about, and formally verify concurrent objects and their implementations. The goal of this paper is thus twofold. First, we are going to provide a brief overview of some known concurrency issues that can occur in smart contracts, characterizing the problems in terms of more traditional concurrency abstractions. Second, we are aiming to build an intuition for "good" and "bad" contract behaviors that can be identified and verified/detected correspondingly, using existing formal methods developed for reasoning about concurrency.

2 Deployed Examples of *Concurrentesque* Behavior

Here we discuss two contracts that have been deployed on the Ethereum blockchain that each illustrate different aspects of concurrent-type behavior. The BlockKing contract, like many others on the Ethereum blockchain today, implements a simple gambling game [2]. Although BlockKing is not heavily used, we study it because it showcases a potential use of the Oraclize service [4], which is a service that allows contracts to communicate with the world outside of the blockchain and thus invites true concurrency. Since the early adopters of the Oraclize service wrote it as a demonstration of the service and has made its source code freely available, it is likely that many other contracts that wish to use Oraclize will mirror it in their implementations.

The second example we discuss is the widely-studied bug in the DAO contract [1]. The DAO established an owner-managed venture capital fund with

[5] A better term would be "uncooperative multitasking" under the circumstances.

more than 18,000 investors; at its height it attracted more than 14% of all Ether coins in existence at that time. The subsequent attack on it cost investors approximately 3.6 million Ether, which at that time was worth approximately USD 50 million. The DAO employed what we call "uncooperative multitasking", in that when the DAO sent money to a recipient then that recipient was able to run code that interfered (via reentrancy) with the DAO's contract state that the DAO assumed would not change during the call.

2.1 The BlockKing contract

The gamble in BlockKing works as follows. At any given time there is a designated "Block King" (initially the writer of the contract). When money is sent to the contract by a sender s, a random number j is generated between 1 and 9. If the current block number modulo 10 is equal to j then s becomes the new Block King. Afterwards, the Block King gets sent a percentage of the money in the contract (from 50% to 90% depending on various parameters), and the writer of the contract gets sent the balance.

Generation of good quality random numbers is often difficult in deterministic systems, especially in a context in which all data is publicly stored—and in which there are financial incentives for attackers. Accordingly, BlockKing utilizes the services of a trusted party, Wolfram Alpha, to generate its random numbers using the Oraclize service. Assuming Oraclize is well-behaved, this strategy for random number selection should be very difficult for attackers to predict.

The code for BlockKing is 365 lines long, but the lines of particular interest are given in Figure 1; line numbers here refer to the actual source code of the contract as given by Etherscan [2]. The `enter` function is called when money is sent to the contract. It sets some contract variables (lines 299–301) and then sends a query to the Oraclize service (line 303).

The `oraclize_query` function raises an event visible in the "real world" before returning to its caller, which then exits (line 304). In the real world the Oraclize servers monitor the event logs, service the request (in this case by contacting the Wolfram Alpha web service), and then make a fresh call into the originating contract at a designated callback point (line 306 in BlockKing). Between the event and its callback, many things can occur, in the sense that the the blockchain can advance several blocks between the call to `oraclize_query` and the resumption of control at `__callback`. During this time the state of the blockchain, and even of the BlockKing contract itself, can have changed drastically. In other words, *this is true concurrent behavior on the blockchain.*

What can go wrong? Suppose that multiple gamblers wish to try their luck in a short period of time (even within the same block). The contract makes no attempt to track this behavior. Accordingly, each new contestant will overwrite the previous one's data (the critical `warriorBlock` and `warrior` variables) in lines 299–301. When the callbacks do eventually occur, the last contestant in the batch will enjoy multiple chances to win the throne curtesy of the earlier contestants in that batch who payed for the other callbacks! The culprit is lines 339–347 from the `process_payment` function, called as the last line of the `__callback` function in line 309.

```
293  function enter() {
294   // 100 finney = .05 ether minimum payment otherwise refund payment and stop contract
295   if (msg.value < 50 finney) {
296    msg.sender.send(msg.value);
297    return;
298   }
299   warrior = msg.sender;
300   warriorGold = msg.value;
301   warriorBlock = block.number;
302   bytes32 myid =
303    oraclize_query(0,"WolframAlpha","random number between 1 and 9");
304  }
305
306  function __callback(bytes32 myid, string result) {
307   if (msg.sender != oraclize_cbAddress()) throw;
308   randomNumber = uint(bytes(result)[0]) - 48;
309   process_payment();
310  }
311
312  function process_payment() {

     ...

339   if (singleDigitBlock == randomNumber) {
340    rewardPercent = 50;
341    // If the payment was more than .999 ether then increase reward percentage
342    if (warriorGold > 999 finney) {
343     rewardPercent = 75;
344    }
345    king = warrior;
346    kingBlock = warriorBlock;
347   }
```

Fig. 1. BlockKing code fragments [2].

Each time the `process_payment` function is called the least significant digit of `warriorBlock` is computed and stored into the variable `singleDigitBlock`.[6] Each time the `process_payment` function is called by `__callback` he has a new chance to match the random number in line 339. If the numbers do match, then that final contestant is crowned on line 345.

2.2 The DAO contract

The source code for the DAO is 1,239 lines and markedly more complex than BlockKing [23]. Since much has already been written about this bug (*e.g.* [9,27]), we present in Figure 2 only the key lines. The problem is the order of line 1012, which (via a series of further function calls) sends Ether to `msg.sender`, and line 1014, which zeros out the balance of `msg.sender`'s account.

In a sequential program, reordering two independent operations has no effect on the ultimate behavior of the program. However, in a concurrent program the effect of a sequentially-harmless reorder can have significant effect since the order in which operations occur can affect how the threads interfere. In the DAO, sending the Ether in line 1012 "yields" control, in some multitasking sense, to any arbitrary (and thus potentially malicious) contract located at `msg.sender`.

[6] For reasons that seem rather strange to us, this modulus is computed very inefficiently in lines 315–338 of the contract, which we elide to save space.

```
1010    // Burn DAO Tokens
1011    Transfer(msg.sender, 0, balances[msg.sender]);
1012    withdrawRewardFor(msg.sender); // be nice, and get his rewards
1013    totalSupply -= balances[msg.sender];
1014    balances[msg.sender] = 0;
1015    paidOut[msg.sender] = 0;
1016    return true;
1017   }
```

Fig. 2. DAO code fragment [23].

Unfortunately, the DAO internal state still indicates that the account is funded since its account balance has not yet been zeroed out in line 1014. Accordingly, a malicious `msg.sender` can initiate a second withdrawal by calling back into the DAO contract, which will in turn send a second payment when control reaches line 1012 again. In fact, the malicious `msg.sender` can then initiate a third, fourth, *etc.* withdrawal, all of which will result in payment. Only at the end is his account zeroed out, after being paid many multiples of its original balance.

Previous analyses of this bug have indicated that the problem is due to recursion or unintended reentrancy. In a narrow sense this is true, but in a wider sense what is going on is that sequential code is running in what is in many senses a concurrent environment.

3 Interference and Synchronization

Having showed that concurrent-type behavior exists and causes problems in real contracts on the Blockchain, we will now examine other ways that our *concurrent-objects-as-contracts* viewpoint can help us understand how contracts can behave on the blockchain.

3.1 Atomic updates in shared-memory concurrency

Figure 3 depicts a canonical example (presented in a Java 8-like pseudocode) of a wrongly used concurrent object, which is supposed to implement an “atomic” counter with methods `get` and `set`. The implementation of the concurrent counter on the left is obviously *thread-safe* (*i.e.*, *data race*-free), thanks to the use of `synchronized` primitives [17]. What is problematic, though, is how an instance of the `Counter` class is used in the multithreaded client code on the right.

Specifically, with two threads running in parallel and their operations interleaving, the call to `incr()` within `thread2`’s body could happen, for instance, between the assignment to `a` and the call `c.set(a + 1)` within the `incr()` call of `thread1`. This would invalidate the condition in the following `assert` statement, making the overall program fail *non-deterministically* for a certain execution!

The issue arises because the implementation of `incr()` on top of `Counter` does not provide the *atomicity guarantees*, expected by the client code. Specifically, the code on the right is implemented in the assumption that there will be *no interference* between the statements of `incr()`, hence the counter `c` is going to be incremented by `1`, and `a` and `b` will be the same by the end of its execution. Indeed,

```
class Counter {
  private int x = 0;

  /** Return current value */
  synchronized int get() {
    return x;
  }

  /** Set x to be v */
  synchronized int set(int v) {
    int t = x;
    x = v;
    return t;
  }
}
```

```
final Counter c = new Counter();

void incr() {
  int a = c.get();
  int b = c.set(a + 1);
  assert (a == b);
}

// In the main method
Runnable thread1 = () ->
  { incr(); }

Runnable thread2 = () ->
  { incr(); }

thread1.run(); thread2.run();
```

Fig. 3. A concurrent counter (left) and its two-thread client application (right).

this is not always the case in the presence of concurrently running `thread2`, and not only `a` and `b` will be different, the later call to `c.set()` will also "overwrite" the result of the earlier one.

A better designed implementation of `Counter` could have instead provided an *atomic* implementation of `incr()`, implemented via a version of *fetch-and-increment* operation [20, § 5.6], via explicit locking, or by means of Java's `synchronized` keyword. However, given the only two methods, `get` and `set`, the implementation of `Counter` has synchronization properties of an atomic register whose *consensus number* [20, § 5.1] (*i.e.*, the number of concurrent threads that can unambiguously agree on the outcomes of `get` and `set`) is exactly 1. Therefore, it is fundamentally impossible to implement an atomic incrementation of `c` by using only `get` and `set`, and without relying on some additional synchronization, by giving priorities to certain preordained threads.

Perhaps a bit surprisingly, even though the implementation of `Counter` from Figure 3 is not flawed by itself, its weak atomicity properties render it quite useless in the presence of an unbounded number of threads, making it virtually impossible to make any *stable* (*i.e.*, resilient with respect to concurrent changes) assumptions about its internal state.

3.2 Atomic updates in concurrent blockchain transactions

The left part of Figure 4 shows a smart contract, implemented in Solidity [15], with functionality and methods reminiscent to those of an atomic concurrent counter. The function `get` allows one to query the contract for the current balance, associated with some fixed address `id`, whereas the `set` function allows one to update balance with the new balance, taken from the message via `msg.value`, sending back the old amount and returning it as a result.

Since the bodies of both `get` and `set` are going to be executed sequentially in the course of some transactions, neither there is any need to synchronized them,

```
contract Counter {
  address public id;
  uint private balance;

  function get() returns (uint) {
    return balance;
  }

  function set() returns (uint) {
    uint t = balance;
    balance = msg.value;
    msg.sender.send(t);
    return t;
  }}
```

```
// ...
// Same code as in Counter

function testAndSet(uint expected)
  returns (uint) {
  uint t = balance;
  if (t == expected) {
    balance = msg.value;
    msg.sender.send(t);
    return t;
  } else {
    throw;
  }
}
```

Fig. 4. A counter contract (left) and a synchronizing `testAndSet` method (right).

nor there is any explicit way to do so in Solidity. However, it is not difficult to observe that as an implementation of the simplest possible storage (*e.g.*, for some `id`-related funds), used by multiple different parties to update it's balance, the `Counter` contract is as useless as its Java counterpart from Figure 3.

For instance, imagine that two parties, unaware of each other try to increment the amount, stored by an instance of `Counter` by a certain value. Since the contract does not provide a way for them to do it in one operation, they will have to first query the amount via `get` and then try to change it via `set` function, following the same pattern as the implementation of `incr` from Figure 3. Indeed, both these calls can be accomplished in a single transaction, which would make the execution sequential. However, because of the limited gas requirement,[7] it is ill-advised to call more than one external contract in the course of execution. Furthermore, the call to `get` can be performed by a client, external to the blockchain, which would mean that the consecutive calls to `get` and `set` will end up in *two different* transactions. If this is the case, those calls might interfere with other transactions, launched by multiple parties trying to modify `Counter` at the same time, making us face the familiar problem: the result of calling the function `set` cannot be predicted out of the local observations.

The cause of the described problem, both in the shared-memory and blockchain cases, is the lack of *strong synchronization primitives*, allowing one to simultaneously observe and manipulate with the counter in the presence of concurrent executions. One solution to the problem, which would make it possible to increment the counter atomically, is to enhance the counter with the `testAndSet` function (right part of Figure 4). This function implements the check/update logic similar to the *compare-and-swap* primitive [20, § 5.8], (known as `CMPXCHG`, on the Intel x86 and Itanium architectures), as a way to implement synchronization between multiple threads. The consensus number of `testAndSet` (and

[7] This is a standard way in Ethereum to ensure that execution of a contract terminates: by supplying it with a limited amount of "gas", used as a fuel for execution steps.

```
contract Counter {
  address public owner;
  uint private balance;

  modifier byOwner() {
    if (msg.sender != owner) throw;
    _
  }

  function get() external byOwner
    returns (uint) {
    return balance;
  }

  function set() external byOwner
    returns (uint) {
    uint t = balance;
    balance = msg.value;
    msg.sender.send(t);
    return t;
  }}
```

```
// Same declarations as in Counter

mapping (address => bool) readers;

// Initialized with 0x0
address writer;

modifier canRead() {
  if (msg.sender != writer ||
      !readers[msg.sender]) throw;
    _
}

modifier canWrite() {
  if (msg.sender != writer) throw;
  _
}

function acquireReadLock() returns (bool) {
  if (writer == 0x0) {
    readers[msg.sender] = true;
  } else return false;
}

// ... Other synchronization primitives
```

Fig. 5. An exclusively-owned (left) and Read/Write-locked (right) contract.

some other similar *Read-Modify-Write* primitives) is known to be ∞, hence it is strong enough to allow an arbitrary number of concurrent parties agree on the outcome of the operation.

Notes on formal reasoning and verification. The modern formal approaches for runtime concurrency verification, based on exploring dynamic execution traces and summarizing their properties, provide efficient tools for detecting the violations of atomicity assumptions, and the lack of synchronization [26]. For instance, by translating our contract to the corresponding shared-memory concurrent object, one would be able to use the existing tools to summarize its traces [13], thus, making it possible to observe undesired interaction patterns.

4 State Ownership and Permission Accounting

A different way to prohibit the unwelcome interference on a contract's state is to engineer a tailored permission accounting discipline, controlling the set of operations allowed for different parties.

Let us first notice that the problems exhibited by the two-thread example in Figure 3 and preventing one from asserting anything about its state `x` could be avoided if we enforced a restricted access discipline: for instance, by stating that at any moment at most one thread can query/modify its state. This would grant the corresponding thread an exclusive *ownership* [30] over the object, thus, justifying any assertions made locally from this thread about the object's state.

The unique ownership is traditionally ensured in Ethereum's contracts by disallowing any other party, but a dedicated *owner*, make critical changes in the contract state. For instance, Figure 5 (left) shows an altered version of the `Counter` contract, so no other party can interact with it but its "owner". The ownership discipline is enforced by Solidity's mechanism of `modifier`s, allowing one to provide custom dynamically checked pre-/postconditions for functions. In our example, the `byOwner` modifier will enforce that the functions `get` and `set` will be only invoked on behalf of a fixed party—the `owner` of the contract.

This is a rather crude solution to the interference problem, as it would mean to exclude any concurrent interaction at a contract whatsoever. It is quite illuminating, though, from a perspective on thinking of contracts as concurrent objects, allowing us to immediately apply our analogy: *accounts are threads*. Indeed, by imposing a specific ownership discipline on a contract as shown in Figure 5 is similar to enhancing its Java counterpart with an explicit check of `Thread.currentThread().getId()`.

Let us now try to push the analogy between accounts and threads a bit further by designing a version of a counter with more elaborated access rights. In particular, we are going to ensure that as long as there are accounts (aka "threads") "interested" in having its value immutable (as their internal logic might rely on its immutability), no other party may be allowed to modify it. Similarly, if at the moment there is exactly one party that holds a unique permission to modify the counter, no other parties may be allowed to read it. The solution to this synchronization problem is well-known in a concurrency community by the name *Read/Write lock* [6]. Its implementation requires keeping track of threads currently reading and writing to the shared object, so a thread should explicitly *acquire* the corresponding permission before performing a read/write operation, and then should *release* it upon finishing.

The right part of Figure 5 shows the essential fragments of the Read/Write-locked contract implementation. The two new fields, `readers` and `writer` keep track of the currently active readers and writers. The new modifiers `canRead` and `canWrite` are to be used for the omitted `get` and `set` operations correspondingly. Finally, `acquireReadLock` allows its caller to acquire the lock as long as there is no active writer in the system, by registering it in the `readers` mapping.

As we can see, the accounts-as-threads is a rather powerful analogy, suggesting a number of solutions to possible synchronization problems that can be taken verbatime from the concurrency literature. The only drawback of the presented solution is the fact that it is rather monolithic: the contract now combines the functionality of the data structure (*i.e.*, the counter) and that of a synchronization primitive (*i.e.*, a lock). We will discuss possible ways to improve the modularity of the implementation in Section 5.

Notes on formal reasoning and verification. Formal reasoning about permission accounting and separation of state access is a long studied topic in the shared-memory concurrency literature (see, *e.g.*, [8] for an overview). Formalisms, such as Concurrent Separation Logic and [30] Fractional/Counting permissions [6] provide a flexible way to define the abstract ownership discipline and verify that a particular implementation follows it faithfully. For instance, our Read/Write lock contract can be formally proven *safe* (*i.e.*, prohibiting concurrent write-modifications) using a formal model of permissions by Bornat *et al.* [6].

5 Discussion

5.1 Composing the contracts

The locking contract "pattern", considered in Section 4, has a significant drawback: its design is *non-modular*. That is, the locking machinery is implemented by the contract itself rather than by a third-party library. This is at odds with good practices of software engineering, in which it is advised to implement synchronization primitives, such as ordinary and reentrant locks, as standalone libraries, which can be used for managing access client-specific resources.

But once the lock logic is factored out of the contract, the reasoning about the contract's behavior becomes significantly more difficult, as, in order to prove the preservation of its internal invariants, one needs to be aware of the properties of the extracted locking protocol, such as, *e.g.*, uniqueness of a writer, which are external to the contract. In other words, verification of a contract can no longer be conduced in an *isolated* manner and will require building a model that allows reasoning about a contract interacting with other, rigorously specified contracts. The idea of disentangling the logic of contracts is not inherent to our concurrent view and is paramount in the existing good practices of contract development. For instance, the same idea is advocated as a way to implement *upgradable* contracts in Ethereum through introducing and additional level of indirection [11]. Having a "contract factory", implemented as another contract, which can be invoked by any party, poses verification challenges similar to those of proving the safety properties of *higher-order* concurrent object (*i.e.*, an object, that is manipulating with other objects) [19].

The idea of compositional reasoning and verification of mutually-dependent and higher-order concurrent objects using concurrency logics has been a subject of a large research body in the past decade [12, 33, 34, 37]. Most of those approaches focus on a notion of *protocol*, serving as an abstract interface of an object's behavior in the presence of concurrent updates, while hiding low-level implementation details (*i.e.*, the actual code). We believe, that by leveraging our analogy, we will be able to develop a method for modular verification of such multi-contract interactions.

5.2 Liveness properties

With the introduction of locks and exclusive access, another concurrency-related issue arises: reasoning about *progress* and *liveness* properties of contract implementations. For instance, it is not difficult to imagine a situation, in which a particular account, registered as a "reader" in our example from Figure 5, might never release the reader-lock, thus, blocking everyone else from being able to change the contract's state in the future. The liveness in this setting would mean that *eventually something good happens*, meaning that any party is properly incentivised to release the lock. In a concurrency vocabulary, such an assumption can be rephrased as *fairness* of the system scheduler, making it possible to reuse existing proof methods for modular reasoning about progress [25] and termination [18] in of single- and multi-contract executions.

6 Related Work

Formal reasoning about smart contracts is an emerging and exciting topic, and suitable abstractions for describing a contract's behavior are a subject of active research. In this section, we relate our observations to the existing results in formalizing and verifying contract properties, outlining promising areas that would benefit from our concurrency analogy.

6.1 Verifying contract implementations

Since the DAO bug [9], the Ethereum community has been focusing on preventing similar errors, with the aid of general-purpose tools for program verification.

At the moment, contracts written in Solidity can be annotated with Hoare-style pre/postconditions and translated down to OCaml code [32], so they become amenable to verification using the Why3 tool, which uses automation to discharge the generated verification conditions [16]. This approach is efficient for verifying basic safety properties of Solidity programs, such as particular variables always being within certain array index boundaries, and preservation of general contract invariants (typically stated in a form if linear equations over values of `uint`-valued variables) at the method boundaries and before performing external contract calls—precisely what was violated by the DAO contract.

Bhargavan *et al.* have recently implemented a translation from a subset of Solidity (without loops and recursion) [5] into $\mathrm{F}^\star$—a programming language and verification framework, based on dependent types [35]. They also provided a translator from EVM bytecode to $\mathrm{F}^\star$ programs. Both these approaches made it possible to use $\mathrm{F}^\star$ as a uniform tool for verification of contract properties, such as invariant preservation and absence of unhandled exceptions, which were encoded as an effect via $\mathrm{F}^\star$'s support for indexed Hoare monad [36]. A similar approach to specify the behavior of contracts and based on dependent types has been adopted by Pettersson and Edström [31], who implemented a small effect-based contract DSL as a shallow embedding into Idris [7], with the executable code extracted to Serpent [14], a Python-style contract language.

Hirai has recently formalized the entire specification of Ethereum Virtual Machine [22] in Lem [28] with extraction to the Isabelle/HOL proof assistant, allowing mechanized verification of contracts, compiled to EVM bytecode, for a number of safety properties, including assertions on mutable state and the absence of potential reentrancy. Unlike the previous approaches, Hirai's formalization does not provide a syntactic way to construct and compose proofs (*e.g.*, via a Hoare-style program logics), and all reasoning about contract behavior is conducted out of the low-level execution semantics [38].

In contrast with these lines of work, which focus predominantly on *low-level* safety properties and invariant preservation, our observations hint a more high-level formalism for capturing the properties of a contract behavior and its communication patterns with the outside world. In particular, we consider communicating state-transition systems (STSs) [29] with abstract state as a suitable formalism for proving, *e.g.*, trace and liveness properties of contract executions using a toolset of established tools, such as TLA+ [24]. In order to connect such an abstract representation with low-level contract code, one will have to prove a *refinement* [3] between the high-level and the low-level representations, *i.e.*, between an STS and the code. In some sense, finding a suitable contract invariant and proving it via Why3 or F* may be considered as proving a refinement between a *one-state* transition system, such that the only state is what is described by the invariant, and an implementation that preserves it. However, we expect more complicate STSs will be required in order to reason about contracts with preemptive concurrency.

6.2 Reasoning about global contract properties

The observation about some contracts being prone to unintentional or adversarial misuse due to the interference phenomenon has been made by Luu *et al.* [27]. They characterised the problem similar to what's exhibited by our counter example in Section 3 as *transaction-ordering dependency* (TOD), which under our concurrency analogy can be generalized as a problem of unrestricted interference. The solution to the TOD-problem, suggested by Luu *et al.*, required changing the semantics of Ethereum transactions, providing a primitive, similar to our `testAndSet` from Figure 4. While the advantage of such an approach is the absence of the need to modify the already deployed contracts (only the client code interacting with them needs to be changed), it requires all involved users to upgrade their client-side applications, in order to account for the changes. In essence, Luu *et al.*'s solution targets a very specific concurrency pattern: strengthening synchronization, provided by atomic registers, by adding a blockchain-supported *read-modify-write* primitive. Realizing the nature of the problem, hinted by our analogy, might instead suggest alternative *contract-based* solutions, such as, *e.g.*, engineering a locking proxy contract. The disadvantage of this approach is, however, the need to foresee this behavior at the moment of designing and deploying a contract. That said, such an ability to model this behavior is precisely what, we believe, our analogy enables.

7 Conclusion

We believe that our analogy between *smart contracts* and *concurrent objects* can provide new perspectives, stimulate research, and allow effective reuse of existing results, tools, and insights for understanding, debugging, and verifying complex contract behaviors in a distributed ledger. As any analogy, ours should not be taken verbatim: on the one hand, there are indeed issues in concurrency, which seem to be hardly observable in contract programming; on the other hand, smart contract implementers should also be careful about notions that do not have direct counterparts in the concurrency realm, such as gas-bounded executions and management of funds.

To conclude, we leave the reader with several speculations, inspired by our observations, but neither addressed nor disproved:

- A common concurrency challenge in non garbage-collected languages is to track the uniqueness of heap locations, which can be later reclaimed and repurposed—an issue dubbed *the ABA problem* [10]. With the lack of due caution, the ABA problem may lead to the violation of the object's state integrity. Can we imagine a similar scenario in a multi-contract setting?

– Continuing the analogy, if one sees a blockchain as a shared state, then the mining protocol defines the priorities for scheduling. Can we leverage the insights from efficient concurrent thread management in order to analyze and improve the existing distributed ledger implementations?
– *Linearizability* [21] (aka *atomicity*) is a standard notion of correctness for specifying high-level behavior of lock-free concurrent objects. What would be an equivalent de-facto notion of consistency for composite contracts with multi-transactional operations, such as BlockKing?

Acknowledgements. Sergey's research is supported by EPSRC grant EP/P009271/1. Hobor's research is funded by Yale-NUS College R-607-265-045-121.

References

1. The DAO. https://en.wikipedia.org/wiki/The_DAO_(organization).
2. BlockKing contract, 2016. https://etherscan.io/address/0x3ad14db4e5a658d8d20f8836deabe9d5286f79e1.
3. M. Abadi and L. Lamport. The existence of refinement mappings. In *LICS*, pages 165–175. IEEE Computer Society, 1988.
4. T. Bertani. Oraclize. http://www.oraclize.it, 2016.
5. K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *PLAS*, pages 91–96. ACM, 2016.
6. R. Bornat, C. Calcagno, P. W. O'Hearn, and M. J. Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270. ACM, 2005.
7. E. Brady. Programming and reasoning with algebraic effects and dependent types. In *ICFP*, pages 133–144. ACM, 2013.
8. S. Brookes and P. W. O'Hearn. Concurrent separation logic. *SIGLOG News*, 3(3):47–65, 2016.
9. V. Buterin. Critical Update Re: DAO Vulnerability. https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability.
10. D. Dechev, P. Pirkelbauer, and B. Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. In *ISORC*, pages 185–192. IEEE Computer Society, 2010.
11. E. Dimitrova. Writing upgradable contracts in Solidity. https://blog.colony.io/writing-upgradeable-contracts-in-solidity-6743f0eecc88, Last visited on February 3, 2017.
12. T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *ECOOP*, volume 6183 of *LNCS*, pages 504–528. Springer, 2010.
13. M. Emmi and C. Enea. Symbolic abstract data type inference. In *POPL*, pages 513–525. ACM, 2016.
14. Ethereum Foundation. The Serpent Contract-Oriented Programming Language. https://github.com/ethereum/serpent.
15. Ethereum Foundation. The Solidity Contract-Oriented Programming Language. https://github.com/ethereum/solidity.
16. J. Filliâtre and A. Paskevich. Why3 - Where Programs Meet Provers. In *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.
17. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

18. A. Gotsman, B. Cook, M. J. Parkinson, and V. Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL*, pages 16–28. ACM, 2009.
19. D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA*, pages 355–364, 2010.
20. M. Herlihy and N. Shavit. *The art of multiprocessor programming.* M. Kaufmann, 2008.
21. M. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Syst.*, 12(3):463–492, 1990.
22. Y. Hirai. Formalization of Ethereum Virtual Machine in Lem. `https://github.com/pirapira/eth-isabelle`, Last visited on February 3, 2017.
23. C. Jentzsch. The DAO, 2016. `https://etherscan.io/address/0xffbd72d37d4e7f64939e70b2988aa8924fde48e3`.
24. L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.
25. H. Liang and X. Feng. A program logic for concurrent objects under fair scheduling. In *POPL*, pages 385–399. ACM, 2016.
26. Y. Lin and D. Dig. CHECK-THEN-ACT misuse of java concurrent collections. In *ICST*, pages 164–173. IEEE Computer Society, 2013.
27. L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *CCS*, pages 254–269. ACM, 2016.
28. D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. In *ICFP*, pages 175–188. ACM, 2014.
29. A. Nanevski, R. Ley-Wild, I. Sergey, and G. A. Delbianco. Communicating state transition systems for fine-grained concurrent resources. In *ESOP*, volume 8410 of *LNCS*, pages 290–310. Springer, 2014.
30. P. W. O'Hearn. Resources, concurrency, and local reasoning. *Th. Comp. Sci.*, 375(1-3):271–307, 2007.
31. J. Pettersson and R. Edström. Safer Smart Contracts through Type-Driven Development. Master's thesis, Chalmers University of Technology, Department of Computer Science and Engineering, Sweden, 2016.
32. C. Reitwiessner. Formal Verification for Solidity Contracts. Last visited on February 3, 2017, `https://forum.ethereum.org/discussion/3779/formal-verification-for-solidity-contracts`.
33. I. Sergey, A. Nanevski, and A. Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87. ACM, 2015.
34. K. Svendsen, L. Birkedal, and M. J. Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, volume 7792, pages 169–188. Springer, 2013.
35. N. Swamy, J. Chen, C. Fournet, P. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, pages 266–278. ACM, 2011.
36. N. Swamy, C. Hritcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P. Strub, M. Kohlweiss, J. K. Zinzindohoue, and S. Z. Béguelin. Dependent types and multi-monadic effects in F*. In *POPL*, pages 256–270. ACM, 2016.
37. A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *ICFP*, pages 377–390. ACM, 2013.
38. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. `http://gavwood.com/paper.pdf`, 2014.

An empirical analysis of smart contracts: platforms, applications, and design patterns

Massimo Bartoletti and Livio Pompianu

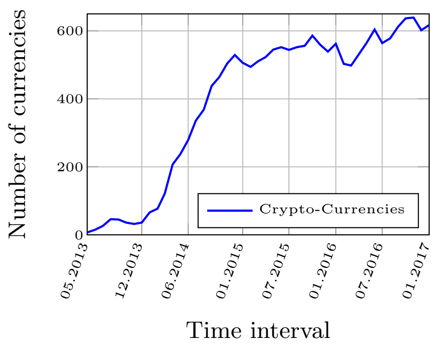
Università degli Studi di Cagliari, Cagliari, Italy
{bart,livio.pompianu}@unica.it

Abstract. Smart contracts are computer programs that can be consistently executed by a network of mutually distrusting nodes, without the arbitration of a trusted authority. Because of their resilience to tampering, smart contracts are appealing in many scenarios, especially in those which require transfers of money to respect certain agreed rules (like in financial services and in games). Over the last few years many platforms for smart contracts have been proposed, and some of them have been actually implemented and used. We study how the notion of smart contract is interpreted in some of these platforms. Focussing on the two most widespread ones, Bitcoin and Ethereum, we quantify the usage of smart contracts in relation to their application domain. We also analyse the most common programming patterns in Ethereum, where the source code of smart contracts is available.

1 Introduction

Since the release of Bitcoin in 2009 [38], the idea of exploiting its enabling technology to develop applications beyond currency has been receiving increasing attention [24]. In particular, the public and append-only ledger of transaction (the *blockchain*) and the decentralized consensus protocol that Bitcoin nodes use to extend it, have revived Nick Szabo's idea of *smart contracts* — i.e. programs whose correct execution is automatically enforced without relying on a trusted authority [45]. The archetypal implementation of smart contracts is Ethereum [26], a platform where they are rendered in a Turing-complete language. The consensus protocol of Ethereum ensures the all and only the valid updates to the contract states are recorded on the blockchain, so ensuring their correct execution.

Besides Bitcoin and Ethereum, a remarkable number of alternative platforms have flourished over the last few years, either implementing crypto-currencies or some forms of smart contracts [1,7,9,28,35]. For instance, the number of crypto-currencies hosted on coinmarketcap.com has incresed from 0 to more than 600 since 2012 and the number of github projects related to blockchains and smart contracts has reached, respectively, 2, 715 and 445 units (see Figure 1). In the meanwhile, industries and some national governments have started dealing with these topics [39, 46], also with significant investments.



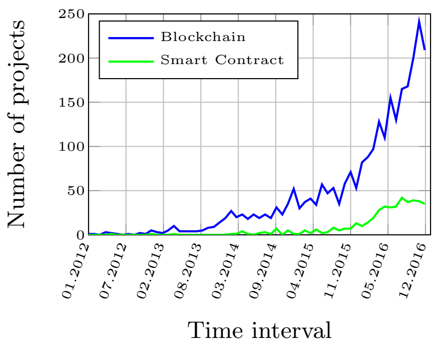


Fig. 1: On the left, monthly trend of the number of crypto-Currencies hosted on coinmarketcap.com. On the right, number of new projects related to blockchains and smart contracts which are created every month on github.com.

Despite the growing hype on blockchains and smart contracts, the understanding of the actual benefits of these technologies, and of their trustworthiness and security, has still to be assessed. In particular, the consequences of unsafe design choices for the programming languages for smart contracts can be fatal, as witnessed by the unfortunate epilogue of the DAO contract [13], a crowdfunding service plundered of $\sim 50M$ USD because of a programming error. Since then, many other vulnerabilities in smart contract have been reported [12, 17, 35].

Understanding how smart contracts are used, and how they are implemented, could help designers of smart contract platforms to create new domain-specific languages (not necessarily Turing complete [25,27,31,40]), which *by-design* avoid vulnerabilities as the ones discussed above. Further, this knowledge could help to improve analysis techniques for smart contracts (like e.g. the ones in [23,35]), by targeting contracts with specific programming patterns.

Contributions. This paper is a methodic survey on smart contracts, with a focus on Bitcoin and Ethereum — the two most widespread platforms currently supporting them. Our main contributions can be summarised as follows:

- in Section 2 we examine the Web for news about smart contracts in the period from June 2013 to September 2016, collecting data about 12 platforms. We choose from them a sample of 6 platforms which are amenable to analytical investigation. We analyse and compare several aspects of the platforms in this sample, mainly concerning their usage, and their support for programming smart contracts.
- in Section 3 we propose a taxonomy of smart contracts, sorting them into categories which reflect their application domain. We collect from the blockchains of Bitcoin and Ethereum a sample of 834 smart contracts, which we classify according to our taxonomy. We then study the usage of smart contracts, measuring the distribution of their transactions by category. This allows us

to compare the different usage of Bitcoin and Ethereum as platforms for smart contracts.
- in Section 4 we consider the source code of the Ethereum contracts in our sample. We identify 9 common design patterns, and we quantify their usage in contract, also in relation to the corresponding category. Together with the previous point, ours constitutes the first quantitative investigation on the usage and programming of smart contract in Ethereum.

All the data collected by our survey are availble online at: `goo.gl/pOswL8`.

2 Platforms for smart contracts

In this section we analyse various platforms for smart contracts. We start by presenting the methodology we have followed to choose the candidate platforms (Section 2.1). Then we describe the key features of each platform, pinpointing differences and similarities, and drawing some general statistics (Section 2.2).

2.1 Methodology

To choose the platforms subject of our study, we have drawn up a candidate list by examining all the articles of `coindesk.com` in the "smart contracts" category[1]. Starting from June 2013, when the first article appeared, up to the 15th of September 2016, 175 articles were published, describing projects, events, companies and technologies related to smart contracts and blockchains. By manually inspecting all these articles, we have found references to 12 platforms: Bitcoin, Codius, Counterparty, DAML, Dogeparty, Ethereum, Lisk, Monax, Rootstock, Symbiont, Stellar, and Tezos.

We have then excluded from our sample the platforms which, at the time of writing, do not satisfy one of the following criteria: (i) have already been launched, (ii) are running and supported from a community of developers, and (iii) are publicly accessible. For the last point we mean that, e.g., it must be possible to write a contract and test it, or to explore the blockchain through some tools, or to run a node. We have inspected each of the candidate platforms, examining the related resources available online (e.g., official websites, white-papers, forum discussions, *etc.*) After this phase, we have removed 6 platforms from our list: Tezos and Rootstock, as they do not satisfy condition (i); Codius and Dogeparty, which violate condition (ii), DAML and Symbiont, which violate (iii). Summing up, we have a sample of 6 platforms: Bitcoin, Ethereum, Counterparty, Stellar, Monax and Lisk, which we discuss in the following.

2.2 Analysis of platforms

We now describe the general features of the collected platforms, focussing on: (i) whether the platform has its own blockchain, or if it just piggy-backs on an

[1] `www.coindesk.com/category/technology/smart-contracts-news`

already existing one; (ii) for platforms with a public blockchain, their consensus protocol, and whether the blockchain is public or private to a specific set of nodes; (iii) the languages used to write smart contracts.

Bitcoin [38] is a platform for transferring digital currency, the bitcoins (BTC). It has been the first decentralized cryptocurrency to be created, and now is the one with the largest market capitalization. The platform relies on a public blockchain to record the complete history of currency transactions. The nodes of the Bitcoin network use a consensus algorithm based moderately hard *"proof-of-work"* puzzles to establish how to append a new block of transactions to the blockchain. Nodes work in competition to generate the next block of the chain. The first node that solves the puzzle earns a reward in Bitcoin.

Although the main goal of Bitcoin is to transfer currency, the immutability and openness of its blockchain have inspired the development of protocols that implement (limited forms of) smart contracts. Bitcoin features a non-Turing complete scripting language, which allows to specify under which conditions a transaction can be redeemed. The scripting language is quite limited, as it only features some basic arithmetic, logical, and crypto operations (e.g., hashing and verification of digital signatures). A further limitation to its expressiveness is the fact that only a small fraction of the nodes of the Bitcoin network can process transactions whose script is more complex than verifying a signature.[2]

Ethereum [26] is the second platform for market capitalization, after Bitcoin. Similarly to Bitcoin, it relies on a public blockchain, with a consensus algorithm similar to that of Bitcoin[3]. Ethereum has its own currency, denoted *ETH*. Smart contracts are written in a stack-based bytecode language [47], which is Turing-complete, unlike Bitcoin. There also exist a few high level languages (the most prominent being *Solidity*[4]), which compile into the bytecode language. Users create contracts and invoke their functions by sending transactions to the blockchain, whose effects are validated by the network. Both users and contracts can store money and send/receive ether to other contracts or to other users.

Counterparty [30] is a platform without its own blockchain; rather, it embeds its data into Bitcoin transactions. While the nodes of the Bitcoin network ignore the data embedded in these transactions, the nodes of Counterparty recognise and interpret them. Smart contracts can be written in the same language used by Ethereum. However, unlike Ethereum, no consensus protocol is used to validate the results of computations[5]. Counterparty has its own currency, which can be transferred between users, and be spent for executing contracts. Unlike Ethereum, nodes do not obtain fees for executing contracts; rather, the fees paid

[2] As far as we know, currently only the *Eligius* mining pool accepts more general transactions (called *non-standard* in the Bitcoin community). However, this pool only mines $\sim 1\%$ of the total mined blocks [19].

[3] The consensus mechanism of Ethereum is a variant of the GHOST protocol in [44].

[4] Solidity: `solidity.readthedocs.io/en/develop/index.html`

[5] See FAQ: How do Smart Contracts "form a consensus" on Counterparty?

by clients are destroyed, and nodes are indirectly rewarded from the inflation of the currency. This mechanism is called *proof-of-burn*.

Stellar [10] features a public blockchain with its own cryptocurrency, governed by a consensus algorithm inspired to federated Byzantine agreement [11]. Basically, a node agrees on a transaction if the nodes in its neighbourhood (that are considered more trusted than the others) agree as well. When the transaction has been accepted by enough nodes of the network, it becomes infeasible for an attacker to roll it back, and it can be considered as confirmed. Compared to *proof-of-work*, this protocol consumes far less computing power, since it does not involve solve cryptographic puzzles. Unlike Ethereum, there is no specific language for smart contracts; still, it is possible to gather together some transactions (possibly ordered in a chain) and write them atomically in the blockchain. Since transactions in a chain can involve different addresses, this feature can be used to implement basic smart contracts. For instance, assume that a participant A wants to pay B only if B promises to pay C after receiving the payment from A. This behaviour can be enforced by putting these transactions in the same chain. While this specific example can be implemented on Bitcoin as well, Stellar also allows to batch operations different from payments, as create a new account. Stellar features special accounts, called *multisignature*, that can be handled by several owners. To perform operations from these accounts, a threshold of consensus must be reached among the owners. Transaction chaining and multisignature accounts can be combined to create more complex contracts.

Monax [8] supports the execution of Ethereum contracts, without having its own currency. Monax allows users to create private blockchains, and to define authorisation policies for accessing them. Its consensus protol[6] is organised in rounds, where a participant proposes a new block of transactions, and the others vote for it. When a block fails to be approved, the protocol moves to the next round, where another participant will be in charge of proposing blocks. A block is confirmed when it is approved by at least 2/3 of the total voting power.

Lisk [6] has its own currency, and a public blockchain with a *delegated proof-of-stake* consensus mechanism[7]. More specifically, 101 active delegates, each one elected by the stakeholders, have the authority to generate blocks. Stakeholders can take part to the electoral process, by placing votes for delegates in their favour, or by becoming candidates themselves. Lisk supports the execution of Turing-complete smart contracts, written either in JavaScript or in Node.js. Unlike Ethereum, determinism of executions is not ensured by the language: rather, programmers must take care of it, e.g. by not using functions like *Math.random*. Although Lisk has a main blockchain, each smart contract is executed on a separated one. Users can deposit or withdraw currency from a contract to the main chain, while avoiding double spending. Contract owners can customise

[6] Tendermint blockchain consensus: `tendermint.com`

[7] Delegated Proof of Stake blockchain consensus: `lisk.io`

Platform	**Blockchain**			**Contract Language**	**Total Tx**	**Volume** (K USD)	**Marketcap** (M USD)
	Type	**Size**	**Block int.**				
Bitcoin	Public	96 GB	10 min.	Bitcoin scripts + signatures	184,045,240	83,178	15,482
Counterparty				EVM bytecode	12,170,386	33	4
Ethereum	Public	17-60 GB	12 sec.	EVM bytecode	14,754,984	10,354	723
Stellar	Public	?	3 sec.	Transaction chains + signatures	?	35	17
Monax	Private	?	Custom	EVM bytecode + permissions	?	n/a	n/a
Lisk	Private	?	Custom	JavaScript	?	45	15

Table 1: General statistics of platforms for smart contracts.

their blockchain before deploying their contracts, e.g. choosing which nodes can participate to the consensus mechanism.

Table 1 summarizes the main features of the analysed platforms. The question mark in some of the cells indicates that we were unable to retrieve the information (e.g., we have not been able to determine the size of Monax blockchains, since they are private). The first three columns next to the platform name describe features of the blockchain: whether it is public; its size; the average time between two consecutive blocks. Note that Bitcoin and Counterparty share the same cell, since the second platform uses the Bitcoin blockchain. Measuring the size of the Ethereum blockchain depends on which client and which pruning mode is used. For instance, using the Geth client, we obtain a measure of 17GB in "fast sync" mode, and of 60GB in "archive" mode.[8] In platforms with private blockchains, their block interval is custom. The fifth column describes the support for writing contracts. The sixth column shows the total number of transactions[9]. The last two columns show the daily volume of currency tranfers, and the market capitalisation of the currency (both in USD, rounded, respectively, to thousands and millions)[10]. All values reported on Table 1 are updated to January 1st, 2017.

3 Analysing the usage of smart contracts

In this section we analyse the usage of smart contracts, proposing a classification which reflects their application domain. Then, focussing on Bitcoin and Ethereum, we quantify the usage of smart contracts in relation to their application domain. We start by presenting the methodology we have followed to sample and classify Bitcoin and Ethereum smart contracts (Section 3.1). Then, we introduce our classification and our statistical analysis (Sections 3.2 and 3.3).

[8] `redd.it/5om2lw`

[9] Obtained from `blockchain.info` for Bitcoin, from `blockscan.com` for Counterparty, and from `etherscan.io` for Ethereum.

[10] Market capitalization estimated by `coinmarketcap.com`.

3.1 Methodology

We sample contracts from Bitcoin and Ethereum as follows:

– for Ethereum, we collect on January, 1st 2017 all the contracts that figure as "verified" on the blockchain explorer `etherscan.io`. This means that the contract bytecode stored on the blockchain matches the source code (generally written in a high level language, such as Solidity) submitted to the explorer. In this way, we obtain a sample of 811 contracts.
– for Bitcoin, we start by observing that many smart contracts save their metadata on the blockchain through the OP_RETURN instruction of the Bitcoin scripting language [1,2,7,21]. We then scan the Bitcoin blockchain on January 1st 2017, searching for transactions that embed in an OP_RETURN some metadata attributable to a Bitcoin smart contract. To this purpose we use an explorer[11] which recognises 23 smart contracts, and extracts all the transactions related to them.

3.2 A taxonomy of smart contracts

We propose a taxonomy of smart contracts into five *categories*, that describe their intended application domain. We then classify the contracts in our sample according to the taxonomy. To this purpose, for Ethereum contracts we manually inspect the Solidity source code, while for Bitcoin contracts we search their web pages and related discussion forums. After this manual investigation, we distribute all the contracts into the five categories that we present below.

Financial. Contracts in this category manage, gather, or distribute money as preeminent feature. Some contracts certify the ownership of a real-world asset, endorse its value, and keep track of trades (e.g., Colu currently tracks over 50,000 assets on Bitcoin). Other contracts implement crowdfunding services, gathering money from investors in order to fund projects (the Ethereum DAO project was the most representative one, until its collapse due to an attack in June 2016). High-yield investment programs are contracts that collect money from users under the promise that they will receive back their funds with interest *if* new investors will join the scheme. In most cases these are frauds, like Ponzi schemes (e.g., GovernMental) and chain-letter schemes (e.g., King of The Ether Throne). Some contracts provide an insurance on setbacks which are digitally provable (e.g., Etherisc sells insurance policies for flights; if a flight is delayed or cancelled, one obtains a refund). Other contracts publish advertisement messages (e.g., PixelMap is inspired to the Million Dollar Homepage).

Notary. Contracts in this category exploit the immutability of the blockchain to store some data persistently, and in some cases to certify their ownership and provenance. Some contracts allow users to write the hash of a document on the blockchain, so that they can prove document existence and integrity (e.g.,

[11] `github.com/BitcoinOpReturn/OpReturnTool`

Category	Platform	Contracts	Transactions
Financial	Bitcoin	6	470,391
	Ethereum	373	624,046
Notary	Bitcoin	17	443,269
	Ethereum	79	35,253
Game	Bitcoin	0	0
	Ethereum	158	58,257
Wallet	Bitcoin	0	0
	Ethereum	17	1,342
Library	Bitcoin	0	0
	Ethereum	29	37,034
Unclassified	Bitcoin	0	0
	Ethereum	155	3,679
Total	Bitcoin	23	913,660
	Ethereum	811	759,611
	Overall	834	1,673,271

Table 2: Transactions by category.

Proof of Existence). Others allow to declare copyrights on digital arts files, like photos or music (e.g., Monegraph). Some contracts (e.g., Eternity Wall) just allow users to write down on the blockchain messages that everyone can read. Other contracts associate users to addresses (often represented as public keys), in order to certify their identity (e.g., Physical Address).

Game. This category gathers contracts which implement *games of chance* (e.g., LooneyLottery, Dice, Roulette, RockPaperScissors) and *games of skill* (e.g., Etherization), as well as some games which mix chance and skill, paying for the solution of some puzzle (e.g., PRNG challenge).

Wallet. The contracts in this category handle keys, send transactions, manage money, deploy and watch contracts, in order to simplify the interaction with the blockchain. Wallets can be managed by one or many owners, in the latter case requiring multiple authorizations (like, e.g. in Multi-owned).

Library. These contracts implement general-purpose operations (like e.g., math and string transformations), to be used by other contracts.

3.3 Quantifying the usage of smart contracts by category

We analyse all the transactions related to the 834 smart contracts in our sample. Table 2 displays how the transactions are distibuted in the categories of Section 3.2. For both Bitcoin and Ethereum, we show the number of detected contracts (third column), and the total number of transactions (fourth column).

Overall, we have 1,673,271 transactions. Notably, although Bitcoin contracts are fewer than those running on Ethereum, they have a larger amount of transactions each. A clear example of this is witnessed by the financial category, where 6 Bitcoin contracts[12] totalize two thirds of the transactions published by the 373 Ethereum contracts in the same category.

[12] Bitcoin financial contracts: Colu, CoinSpark, OpenAssets, Omni, SmartBit, BitPos.

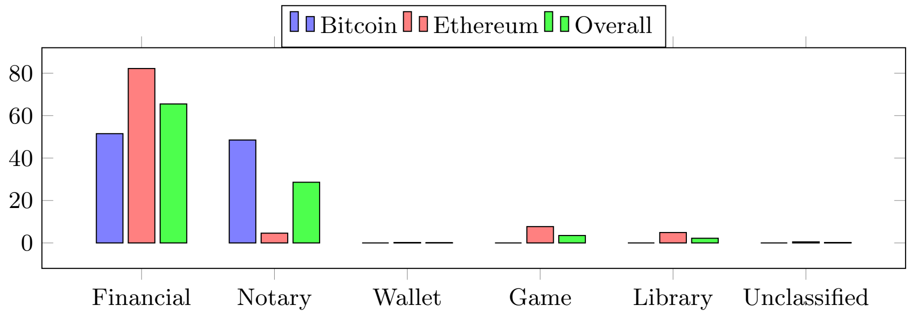


Fig. 2: Distribution of transactions by category.

While both Bitcoin and Ethereum are mainly focussed on financial contracts, we observe major differences about the other categories. For instance, the Bitcoin contracts in the Notary category[13] have an amount of transactions similar to that of the Financial category, unlike in Ethereum. The second most used category in Ethereum is Game. Although some games (e.g., lotteries [15, 16, 18, 22] and poker [34]) which run on Bitcoin have been proposed in the last few years, the interest on them is still mainly academic, and there is no experimental evidence that these contracts are used in practice. Instead, the greater flexibility of the Ethereum programming language simplifies the development of this kind of contracts (although with some quirks [29] and limitations[14]).

Note that in some cases there are not enough elements to categorise a contract. This happens e.g., when the contract does not link to the project webpage, and there are neither comments in the forum nor in the contract sources.

4 Design patterns for Ethereum smart contracts

In this section we study design patterns for Ethereum smart contracts. To this purpose, we consider the sample of 811 contracts collected through the methodology described in Section 3. By manually inspecting the Solidity source code of each of these contracts, we identify some common design patterns. We start in Section 4.1 by describing these patterns. Then, in Section 4.2 we measure the usage of the patterns in the various categories of contracts identified in Section 3.

[13] Bitcoin notary contracts: Factom, Stampery, Proof of Existence, Blocksign, Crypto-Copyright, Stampd, BitProof, ProveBit, Remembr, OriginalMy, LaPreuve, Nicosia, Chainpoint, Diploma, Monegraph, Blockai, Ascribe, Eternity Wall, Blockstore.

[14] Although the Ethereum virtual machine is designed to be Turing-complete, in practice the limitations on the amount of gas which can be used to invoke contracts also limit the set of computable functions (e.g., verifying checkmate exceeds the current gas limits of a transaction [33]).

4.1 Design patterns

Token. This pattern is used to distribute some fungible goods (represented by tokens) among users. Tokens can represent a wide variety of goods, like e.g. coins, shares, outcomes or tickets, or everything else which is transferable and countable. The implications of owning a token depend on the protocol and the use case for which the token has been issued. Tokens can be used to track the ownership of physical properties (e.g., gold [3]), or digital one (e.g., cryptocurrency). Some crowdfunding systems issue tokens in exchange for donations (e.g., the Congress contract). Tokens are also used to regulate user authorizations and identities. For instance, the DVIP contract specifies rights and term of services for owners of its tokens. To vote on the poll ETCSurvey, users must possess a suitable token. Given the popularity of this pattern, its standardisation has been proposed [5]. Notably, the majority of the analysed Ethereum contracts which issue tokens already adhere to it.

Authorization. This pattern is used to restrict the execution of code according to the caller address. The majority of the analysed contracts check if the caller address is that of the contract owner, before performing critical operations (e.g., sending ether, invoking suicide or selfdestruct). For instance, the owner of Doubler is authorized to move all funds to a new address *at any time* (this may raise some concerns about the trustworthiness of the contract, as a dishonest owner can easily steal money). Corporation checks addresses to ensure that every user can vote only once per poll. CharlyLifeLog uses a white-list of addresses to decide who can withdraw funds.

Oracle. Some contracts may need to acquire data from outside the blockchain, e.g. from a website, to determine the winner of a bet. The Ethereum language does not allow contracts to query external sites: otherwise, the determinism of computations would be broken, as different nodes could receive different results for the same query. Oracles are the interface between contracts and the outside. Technically, they are just contracts, and as such their state can be updated by sending them transactions. In practice, instead of querying an external service, a contract queries an oracle; and when the external service needs to update its data, it sends a suitable transaction to the oracle. Since the oracle is a contract, it can be queried from other contracts without consistency issues. One of the most common oracles is Oraclize [15]: in our sample, it is used by almost all the contracts which resort to oracles.

Randomness. Dealing with randomness is not a trivial task in Ethereum. Since contract execution must be deterministic, all the nodes must obtain the same value when asking for a random number: this struggles with the randomness requirements wished. To address this issue, several contracts (e.g., Slot) query oracles that generate these values off-chain. Others (e.g., Lottery) try to generate the numbers locally, by using values not predictable *a priori*, as the hash of a block not yet created. However, these techniques are not generally considered secure [17].

[15] `oraclize.it`

	Token	**Auth.**	**Oracle**	**Random.**	**Poll**	**Time**	**Termin.**	**Fork**	**Math**	**None**
Financial	24-51	51-39	2-15	1-2	5-29	23-31	14-30	8-69	4-47	29-66
Notary	13-6	52-9	1-2	0-0	8-9	20-6	29-13	0-0	1-3	30-15
Game	3-3	84-27	25-74	72-93	25-57	73-43	21-19	1-3	2-9	1-1
Wallet	18-2	100-3	0-0	0-0	0-0	94-6	100-10	0-0	12-6	0-0
Library	0-0	31-2	0-0	14-3	0-0	24-3	24-4	34-24	21-19	17-3
Unclassified	43-39	66-21	3-9	1-1	3-6	18-10	28-25	28-25	1-5	15-15
Total	*21-100*	*61-100*	*7-100*	*15-100*	*9-100*	*33-100*	*22-100*	*5-100*	*4-100*	*20-100*

Table 3: Relations between design patterns and contract categories. A pair (p, q) at row i and column j means that $p\%$ of the contracts in category i use the pattern of column j, and $q\%$ of contracts with pattern j belong to category i.

Poll. Polls allows users to vote on some question. Often this is a side feature in a more complex scenario. For instance, in the Dice game, when a certain state is reached, the owner issues a poll to decide whether an emergency withdrawal is needed. To determine who can vote and to keep track of the votes, polls can use tokens, or they can check the voters' addresses.

Time constraint. Many contracts implement time constraints, e.g. to specify when an action is permitted. For instance, BirthdayGift allows users to collect funds, which will be redeemable only after their birthday. In notary contracts, time constraints are used to prove that a document is owned from a certain date. In game contracts, e.g. Lottery, time constraints mark the stages of the game.

Termination. Since the blockchain is immutable, a contract cannot be deleted when its use has come to an end. Hence, developers must forethink a way to disable it, so that it is still present but unresponsive. This can be done manually, by inserting ad-hoc code in the contract, or automatically, calling `selfdestruct` or `suicide`. Usually, only the contract owner is authorized to terminate a contract (e.g., as in SimpleCoinFlipGame).

Math. Contracts using this pattern encode the logic which guards the execution of some critical operations. For instance, Badge implements a method named `subtractSafely` to avoid subtracting a value from a balance when there are not enough funds in an account.

Fork check. The Ethereum blockchain has been forked four times, starting from July 20th, 2016, when a fork was performed to contrast the effect of the DAO attack [4]. To know whether or not the fork took place, some contracts inspect the final balance of the DAO. Other contracts use this check to detect whether they are running on the main chain or on the fork, performing different actions in the two cases. AmIOnTheFork is a library contract that can be used to distinguish the main chain from the forked one.

4.2 Quantifying the usage of design patterns by category

We now study how the design patterns identified in Section 4.1 are used in smart contracts. Out of the 811 analysed contracts, 648 use at least one of the 9 patterns presented, for a grand total of 1427 occurrences of usage.

Table 3 shows the correlation between the usage of design patterns and contract categories, as defined in Section 3. A cell at row i and column j shows a pair of values: the first value is the percentage of contracts of category i that use the pattern of column j; the second one is the percentage of contracts with pattern j which belongs to category i. So, for instance, 24% of the contracts in the financial category presents the token pattern, and 51% of all the contracts with the token pattern are financial ones.

We observe that *token*, *authorization*, *time constraint*, and *termination* are generally the most used patterns. Some patterns are spread across several categories (e.g., *termination* and *time constraint*), while others are mainly adopted only in one. For instance, *oracle* and *randomness* patterns are peculiar of game contracts, while the *token* pattern is mostly used in financial contracts. Although *math* is the less used, it appears in each category. Some contracts do not use any pattern (29% of financial and 30% of notary); almost all the contracts in game and wallet categories uses at least one. Further, only 15% of all the unclassified contracts do no use any pattern at all.

The most frequent patterns in financial contracts are *token* (24%), *authorization* (51%), and *time constraint* (23%). Due to the presence of contracts which implement assets and crowdfunding services, we have that half of contracts using *token* and *math* patterns belong to the financial category. For instance, these services use *token* for representing goods or developing polls. Moreover, a great 69% of contracts that use the *fork check* pattern is financial. This is caused by the necessity of knowing the branch of the fork before deciding to move funds. Finally, several financial applications (29%) perform simple operations (e.g. sending a payment) without using any of our described patterns.

The *authorization* pattern is used in many notary contracts to ensure that only the owner of a document can add or modify its data, in order to avoid tampering. Most gambling games involve players who pay fees to join the game, and rewards that can be collected by the winner of the game. The *authorization* pattern is used to let the owner to be the only one able to redeem participants' fees or to perform administrative operations, and to let the winner withdraw his reward. The *time constraint* pattern is used to distinguish the different phases of the game. For instance, within a specific time interval players can apply for the game and/or bet; then, bets are over, and the game determines a winner. To obtain the winner, usually gambling games resort to random numbers, which are often generated through an oracle. Indeed, 25% of games use the *oracle* pattern, and the pattern itself is used 74% of cases by a game contract. Since *all* game contracts invoking an *oracle* (25%) ask for random values, and since 72% of contracts use the *random* pattern, we can deduce that 47% of them generate random numbers without resorting to oracles.

Notably, 100% of wallet contracts adopt both *authorization* and *termination* design patterns. A high 94% also uses *time constraint*. On the contrary, *oracle*, *poll*, and *randomness* patterns are of little use when developing a wallet, while *math* is sometimes used for securing operations on the balance.

5 Conclusions

We have analysed the usage of smart contracts from various perspectives. In Section 2 we have examined a sample of 6 platforms for smart contracts, pinpointing some crucial technical differences between them. For the two most prominent platforms — Bitcoin and Ethereum — we have studied a sample of 834 contracts, categorizing each of them by its application domain, and measuring the relevance of each of these categories (Section 3). The availability of source code for Ethereum contract has allowed us to analyse the most common design patterns adopted when writing smart contracts (Section 4).

We believe that this survey may provide valuable information to developers of new, domain-specific languages for smart contracts. In particular, measuring what are the most common use cases allows to understand which domains deserve more investments. Furthermore, our study of the correlation between design patterns and application domains can be exploited to drive the correct choice of programming primitives of domain-specific languages for smart contracts.

Due to the mixed flavour of our analysis, which compares differents platforms and studies how smart contracts are interpreted on each them, our work relates to various topics. The work [36] proposes design patterns for altering and undoing of smart contracts; so far, our analysis in Section 4.2 has not still found instances of these patterns in Ethereum. Among the works which study blockchain technologies, [14] compares four blockchains, with a special focus on the Ethereum one; [43] examines a larger set of blockchains, including also some which does not fit the criteria we have used in our methodology (e.g., RootStock and Tezos). Many works on Bitcoin perform empirical analyses of its blockchain. For instance, [41, 42] study users deanonymization, [37] measures transactions fees, and [20] analyses Denial-of-Service attacks on Bitcoin. Also, [32] investigates whether Bitcoin users are interested more on digital currencies as asset or as currency, with the aim of detecting the most popular use cases of Bitcoin contracts, similarly to what we have done in Section 3.3. Our classification of Bitcoin protocols based on OP_RETURN transactions is inspired from [21], which also measures the space consumption and temporal trend of OP_RETURN transactions.

Recently, some authors have started to analyse the security of Ethereum smart contracts: among these, [17] surveys vulnerabilities and attacks, while [35] and [23] propose analysis techniques to detect them. Our study on design patterns for Ethereum smart contracts could help to improve these techniques, by targeting contracts with specific programming patterns.

Acknowledgments. This work is partially supported by Aut. Reg. of Sardinia project P.I.A. 2013 "NOMAD".

References

1. Bitcoin contract, https://en.bitcoin.it/wiki/Contract. Last accessed 2017/01/14
2. Bitcoin OP_RETURN wiki page, https://en.bitcoin.it/wiki/OP_RETURN. Last accessed 2017/01/14
3. Dgx website, https://www.dgx.io/. Last accessed 2017/01/14
4. Ethereum hard fork 20 july 2016, https://blog.ethereum.org/2016/07/20/hard-fork-completed/. Last accessed 2017/01/14
5. Ethereum request for comment 20, https://github.com/ethereum/wiki/wiki/Standardized_Contract_APIs. Last accessed 2017/01/14
6. Lisk, https://lisk.io/. Last accessed 2017/01/14
7. Making sense of blockchain smart contracts, http://www.coindesk.com/making-sense-smart-contracts/. Last accessed 2017/01/14
8. Monax, https://monax.io/. Last accessed 2017/01/14
9. Smart contracts: The good, the bad and the lazy, http://www.multichain.com/blog/2015/11/smart-contracts-good-bad-lazy/. Last accessed 2017/01/14
10. Stellar, https://www.stellar.org/. Last accessed 2017/01/14
11. The Stellar consensus protocol, https://www.stellar.org/papers/stellar-consensus-protocol.pdf. Last accessed 2017/01/14
12. Thinking about smart contract security, https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/. Last accessed 2017/01/14
13. Understanding the DAO attack, http://www.coindesk.com/understanding-dao-hack-journalists/. Last accessed 2017/01/14
14. Anderson, L., Holz, R., Ponomarev, A., Rimba, P., Weber, I.: New kids on the block: an analysis of modern blockchains. CoRR abs/1606.06530 (2016)
15. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. In: IEEE S & P. pp. 443–458 (2014)
16. Andrychowicz, M., Dziembowski, S., Malinowski, D., Mazurek, L.: Secure multiparty computations on Bitcoin. Commun. ACM 59(4), 76–84 (2016), http://doi.acm.org/10.1145/2896386
17. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts. Cryptology ePrint Archive, Report 2016/1007 (2016), http://eprint.iacr.org/2016/1007
18. Back, A., Bentov, I.: Note on fair coin toss via Bitcoin. http://www.cs.technion.ac.il/~idddo/cointossBitcoin.pdf (2013)
19. Banasik, W., Dziembowski, S., Malinowski, D.: Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In: ESORICS. pp. 261–280 (2016)
20. Baqer, K., Huang, D.Y., McCoy, D., Weaver, N.: Stressing out: Bitcoin "stress testing". In: Bitcoin Workshop. pp. 3–18 (2016)
21. Bartoletti, M., Pompianu, L.: An analysis of Bitcoin OP_RETURN metadata. CoRR abs/1702.01024 (2016), https://arxiv.org/abs/1702.01024
22. Bentov, I., Kumaresan, R.: How to use Bitcoin to design fair protocols. In: CRYPTO. pp. 421–439 (2014)
23. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Beguelin, S.: Formal verification of smart contracts. In: PLAS (2016)

24. Bonneau, J., Miller, A., Clark, J., Narayanan, A., Kroll, J.A., Felten, E.W.: SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In: IEEE S & P. pp. 104–121 (2015)
25. Brown, R.G., Carlyle, J., Grigg, I., Hearn, M.: Corda: An introduction. http://r3cev.com/s/corda-introductory-whitepaper-final.pdf (2016)
26. Buterin, V.: Ethereum: a next generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper (2013)
27. Churyumov, A.: Byteball: a decentralized system for transfer of value. https://byteball.org/Byteball.pdf (2016)
28. Clack, C.D., Bakshi, V.A., Braine, L.: Smart contract templates: foundations, design landscape and research directions. CoRR abs/1608.00771 (2016)
29. Delmolino, K., Arnett, M., Miller, A., Kosba, A., Shi, E.: Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab (2016)
30. Dermody, R., Krellenstein, A., Slama, O., Wagner, E.: Counterparty: Protocol specification (2014), http://counterparty.io/docs/protocol_specification/. Last accessed 2017/01/14
31. Frantz, C.K., Nowostawski, M.: From institutions to code: towards automated generation of smart contracts. In: Workshop on Engineering Collective Adaptive Systems (eCAS) (2016)
32. Glaser, F., Zimmermann, K., Haferkorn, M., Weber, M.C., Siering, M.: Bitcoin-asset or currency? revealing users' hidden intentions (2014)
33. Grau, P.: Lessons learned from making a chess game for Ethereum (2016), https://medium.com/@graycoding/lessons-learned-from-making-a-chess-game-for-ethereum-6917c01178b6#.fwtdwly6e. Last accessed 2017/01/14
34. Kumaresan, R., Moran, T., Bentov, I.: How to use Bitcoin to play decentralized poker. In: ACM CCS. pp. 195–206 (2015)
35. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: ACM CCS (2016), http://eprint.iacr.org/2016/633
36. Marino, B., Juels, A.: Setting standards for altering and undoing smart contracts. In: RuleML. pp. 151–166 (2016)
37. Möser, M., Böhme, R.: Trends, tips, tolls: A longitudinal study of bitcoin transaction fees. In: Financial Cryptography and Data Security. pp. 19–33 (2015)
38. Nakamoto, S.: Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf (2008)
39. Nomura Research Institute: Survey on blockchain technologies and related services, http://www.meti.go.jp/english/press/2016/pdf/0531_01f.pdf
40. Popejoy, S.: The Pact smart contract language. http://kadena.io/pact (2016)
41. Reid, F., Harrigan, M.: An analysis of anonymity in the Bitcoin system. In: Security and privacy in social networks, pp. 197–223. Springer (2013)
42. Ron, D., Shamir, A.: Quantitative analysis of the full Bitcoin transaction graph. In: Financial Cryptography and Data Security. pp. 6–24. Springer (2013)
43. Seijas, P.L., Thompson, S., McAdams, D.: Scripting smart contracts for distributed ledger technology. Cryptology ePrint Archive, Report 2016/1156 (2016), http://eprint.iacr.org/2016/1156
44. Sompolinsky, Y., Zohar, A.: Secure high-rate transaction processing in bitcoin. In: Financial Cryptography and Data Security. pp. 507–527 (2015)
45. Szabo, N.: Formalizing and securing relationships on public networks. First Monday 2(9) (1997), http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/548

46. UK Government Chief Scientific Adviser: Distributed ledger technology: beyond block chain, https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/492972/gs-16-1-distributed-ledger-technology.pdf
47. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. gavwood.com/paper.pdf (2014)

Trust in Smart Contracts is a Process, As Well

Firas Al Khalil, Tom Butler, Leona O'Brien, and Marcello Ceci

Governance, Risk, and Compliance Technology Center
University College Cork
{firas.alkhalil,tbutler,leona.obrien,marcello.ceci}@ucc.ie

Abstract. Distributed ledger technologies are rising in popularity, mainly for the host of financial applications they potentially enable, through smart contracts. Several implementations of distributed ledgers have been proposed, and different languages for the development of smart contracts have been suggested. A great deal of attention is given to the practice of development, i.e. programming, of smart contracts. In this position paper, we argue that more attention should be given to the *"traditional developers"* of contracts, namely the lawyers, and we propose a list of requirements for a human and machine-readable contract authoring language, friendly to lawyers, serving as a common (and a specification) language, for programmers, and the parties to a contract.

1 Introduction

The emergence of distributed ledger technology, due to the development of Bitcoin [22], sparked a lot of interest in different communities: from academia to industry, and from technological and financial circles to philosophical ones [24,27].

The amount of enthusiasm generated around distributed ledgers is indicative of the potentialities that are waiting to be tapped into. What is undeniable, today, is that the financial industry is paying very close attention to cryptocurrencies, especially Bitcoin, but also to other financial applications enabled by distributed ledgers.

Which brings us to *smart contracts*, a concept first envisioned by Nick Szabo [29], as far as 1995 so is claimed, and now believed to be enabled by the advent of distributed ledgers. Several definitions for smart contracts exist, varying in their faithfulness to the original concept, and some of them only adding to the existing confusion surrounding them. We will stand by the original definition of Szabo: *"[s]mart contracts [. . .] facilitate all steps of the contracting process"*; search, negotiation, commitment, performance, and adjudication are all parts of the contracting process he mentioned [28].

Bitcoin, as a platform, is able to model and execute smart contracts, but with a lot of restrictions due to its limited scripting language. This limitation, along with the observation that cryptocurrencies can be viewed as "just another kind of smart contracts", led eventually to the development of Ethereum [31]: a decentralised platform where smart contracts are first-class citizens; the distributed ledger is equipped with a Turing complete programming language that enables developers to write *"arbitrary"* contracts/code. More recently, platforms built on top of Bitcoin and supporting a Turing complete smart contracts language were developed (e.g. Rootstock [11]), and maybe more interestingly, platforms for smart contracts with non Turing complete languages were also developed, i.e. τ-Chain [6].

It is not a surprise that traditional programmers, if one may call them so, are unable to carry out *"economical thinking"* [10]; indeed, they are also, in our experience, ill-equipped to capture legal or regulatory thinking. The inverse can be said of *subject-matter experts*, i.e. business analysts and lawyers; they are most certainly unable to carry out *"computational thinking"*.

How to carry out the development of smart contracts in large financial institutions, where, traditionally, contracts are drafted by subject-matter experts? More importantly, how can we reason on the legality of developed contracts? Either manually by a lawyer, or automatically using a tool for compliance checking? A failure to answer these questions inevitably contributes to the scepticism of the financial industry –which has been put under the microscope by regulators since 2008– about the future of smart contracts, and the industry's reluctance in adopting it.

In this position paper, we argue that trust in smart contracts, is *also* a process; a bridge is needed to connect both sides of the abyss.

The rest of this paper is organised as follows: Section 2 shows how diverse is the scene of distributed ledger technologies; Section 3 shows how irreconcilable are the languages of programmers and subject-matter experts; Section 4 develops our views on how can we build a bridge that enables trust, from an institutional perspective, in smart contracts; we finally conclude in Section 5.

2 On Distributed Ledger Technologies

The introduction of Bitcoin by Satoshi Nakamoto [22] polarised the actors in the financial industry since the beginning: some were extremely enthusiastic about it, to the point where they claimed that Bitcoin is the *"next big thing"*, and others were extremely sceptical about it.

The innovation of Bitcoin is not limited to the currency; the idea of a shared ledger itself proved to be very powerful and sprung many platforms rivalling or even complementing Bitcoin. The interested reader can refer to Tschorsch and Scheuermann [30] for an excellent technical survey on distributed ledger technologies. Moreover, a quick look at the currently available platforms inspired by Bitcoin, gives a good idea on the rising popularity of the technology: for instance, `coinmarketcap.com`, a site that tracks market capitalisation of different cryptocurrencies, lists 719 platforms.

Since its inception, Bitcoin provided a stack-based scripting language that allowed developers to define the conditions to spend Bitcoins (e.g. requiring multiple signatures), which revived the vision of smart contracts. However, this scripting language is purposefully not Turing complete, which ultimately means that it is limited in expressivity. In the following, we will take a look at four different platforms that are meant to overcome Bitcoin's scripting limitations, illustrating the different technical choices one can make, regarding the development of smart contracts.

The first platform we are going to look at, which is currently almost synonymous to "smart contracts", is Ethereum [31]. Ethereum was proposed as a distributed platform independent of –yet very similar to– Bitcoin. To create distributed trust-less consensus and solve the double-spending problem, Ethereum uses proof-of-work, just like Bitcoin, however, it provides the Ethereum Virtual Machine (`EVM`) that runs a Turing complete stack based language, which opens the doors to a hypothetically unlimited number of applications. Developers are not forced to use the `EVM`'s opcode to write smart contracts. Indeed, they can use `Solidity` or `Serpent`, which are high-level programming languages, similar to `javascript` or `python`, respectively, that can compile to `EVM` byte code.

The second platform we are going to look at is Nxt [1], one of the earliest smart contract platforms. Unlike Bitcoin and Ethereum, Nxt uses proof-of-stake to achieve consensus and solve the double-spending problem. Moreover, Nxt does not provide a scripting language to smart contract developers; instead, it provides a RESTful API exposing a set of primitive operations (like spending, storing strings, sending messages, etc.) that developers can invoke.

The third platform we will consider is Rootstock [11]. Unlike Ethereum, and Nxt, Rootstock was developed to complement Bitcoin (as a *sidechain* [12]) and provides its own Turing complete virtual machine (the `RVM`) to enable smart contracts.

The fourth and final platform we will examine is $\tau-$Chain [6]. The authors of this platform argue that Turing completeness is not necessary for distributed ledgers, because with Turing completeness comes undecidability, i.e. smart contracts can go in an infinite loop and the network will never be able to predict this behaviour. Indeed, Ethereum overcomes the problem of undecidability by forcing the caller of the smart contract to provide *gas* with the transaction (bought with *ether*, Ethereum's own cryptocurrency); every instruction on the `EVM` consumes a predefined amount of gas, and they are non-refundable, i.e. if the gas is totally consumed and the smart contract didn't finish execution, the gas is never returned to the caller.

However, Asor [6] propose the use of an ontology [2] of rules, along with a reasoner, to enable computations on the network. Authors of smart contracts would write them in a totally functional programming language, like `Idris` [7], that will be ultimately translated into the ontology. This approach will not only make computations decidable, but it also allows the assertion of properties of smart contracts that were impossible with Turing complete languages, e.g. if the contract connects to the Internet or not, or if the contract fulfills some interfaces/requirements/etc.

The interested reader can refer to the survey written by Seijas et al. [25] for more information on scripting languages for distributed ledgers. The aforementioned platforms illustrate some of the variations that exist in distributed

[1] `https://nxt.org/`

ledger technology's ecosystem. These platforms can differ not only in the tooling and the language they expose for smart contract development, but also in the paradigms that govern them. The development of smart contracts thus requires a deep and serious understanding of the target platform. In the following section, we will examine what hinders a fast adoption of such an enabling technology by the financial industry.

3 Staring Into the Abyss

A close inspection of the literature shows that effort is being put in helping developers author smart contracts, by either developing tools, or creating abstractions.

Recently, Delmolino et al. [10] reported on their experience in teaching smart contract programming, using Ethereum, to undergraduate students at the University of Maryland. The authors concluded that smart contract programming *requires an "economic thinking" perspective that traditional programmers may not have acquired.* Indeed, students repeatedly made logical errors that ultimately lead to money leaks, failed to use cryptographic primitives to secure the contracts from attackers, failed to account for the incentives of contract callers, and even made mistakes directly related to Ethereum.

This observation lead to the development of a Masters thesis by Pettersson and Edström [23], and their objective was to help said programmers to develop safer smart contracts. Their aim is to prevent 3 kinds of mistakes contract developers fall in: unexpected states, failure to use cryptography, and overflowing the `EVM`'s stack. They propose to use of a functional programming language, namely `Idris`. They developed a code generator that transforms code produced by an `Idris` compiler to `Serpent` code, which can be subsequently compiled into `EVM` bytecode.

In a different, yet related work, Luu et al. [21] noted that a class of security-related bugs in smart contracts are due to the gaps in the understanding of the distributed semantics of the underlying platform.

Another interesting work is that of Florian et al. [20], who propose the use of logic-based smart contracts. They showed that this approach can complement smart contracts written in procedural code, in terms of contract negotiation, formation, storage/notarizing, enforcement, monitoring and activities related to dispute resolution.

In a different take, García-Bañuelos et al. [16] showed how the business process language `BPMN` can be mapped into executable smart contracts on the Ethereum. This development lead Hull et al. [19] to propose a *Business Collaboration Language* (`BCL`) for shared ledgers. Indeed, this `BCL` can be thought of as the equivalent of `SQL` for relational databases, targeting shared ledgers, regardless of implementation-specific details.

As far as we know, the only works that consider the issue of authoring smart contracts from the subject-matter expert's perspective are those proposed by Frantz and Nowostawski [14] and Clack et al. [9].

Frantz and Nowostawski [14] propose a semi-automated method for the translation of human readable contracts to smart contracts on Ethereum. The authors develop a domain specific language for contract modelling, where statements are rules expressed in English, and that translates into `Solidity`. However, this solution is very tied to Ethereum, and it is not clear how extensible or adaptable it is. Additionally, it doesn't cover the legal language that a lawyer would be accustomed to.

Clack et al. [9] rightly identify two semantics of contracts:

Operational semantics: concerned with the execution of the contract on a specific platform

Denotational semantics: that captures the *"legal meaning"* of the contract, as understood by a lawyer.

The authors envision the use of smart contract templates, based on the idea of Ricardian Contracts [17,18]. A Ricardian Contract is a digitlly signed triple $\langle P, C, M\rangle$, where P is the legal prose, capturing denotational semantics, C is the platform specific code expressing operational semantics, and M is a map (key-value pairs) of parameters used in P and C.

While the use of smart contract templates, based on Ricardian Contracts, looks like a move towards the right direction, we argue that prose should not be tied to code:

- While the semantics of legal language can be expressed as a set of deontic defeasible rules, the code is rather procedural. The order of the instructions in the procedure does not reflect the natural order of the contract clauses expressed in natural language [20].
- The life-cycle of legal prose is independent from the life-cycle of the code. For example, a lawyer might describe the terms of a contract in prose and never come back to it, while a developer will –most likely– iterate through different implementations (e.g. bug fixes).
- There is not a single smart contract platform, which ultimately means that different parameters (key-value pairs of M) will be needed for different platforms. For example, several works (e.g. [32,3,1]) describe data feed systems that enable smart contracts to consume data feeds from outside the distributed ledger (e.g. a stock market index); while the notion of an external feed might be familiar to a lawyer, its technical details, thus the choices related to the adoption of one method over another, and eventually the parametrisation is definitely out of her/his reach and/or interest.

In the following section, we will identify the key issues, as we see them, regarding the adoption of smart contracts, and how we envision to solve them.

4 Trusting Smart Contracts

In Section 2 we tried to show, through a non-exhaustive list of examples, how distributed ledgers can differ on a deep technical level, which requires a very intimate technological knowledge by the *implementer* of the smart contract. Afterwards, we showed, in Section 3, how current effort is mostly focused on developing technical tools and infrastructure aimed at facilitating the technical implementation of smart contracts. However, there is a major lacuna in all this: that is the translation, or mapping, of the contract's denotational semantics to its operational semantics.

We share the view of Clack et al. [9] on the separation between operational and denotational semantics of contracts. In fact, we argue that trust in smart contracts can only stem from the ability of lawyers in financial institutions to understand, express, and ultimately validate the denotational semantics of a contract. However, we disagree on the assumption they make on the languages expressing these semantics, i.e. any assumption on the correspondence between a *"legal language"* and the *"technical language"* cannot be achieved, as the lawyer cannot predict the behaviour of the code.

What is missing from all of the described work, is the realisation that the involvement of a lawyer, especially in the heavily regulated financial industry, in the authoring of contracts, not only smart contracts, is paramount, for her/his knowledge on the regulation governing said contracts dictates the denotational semantics. A lawyers' knowledge of the explicit and implicit rights and obligations, counterparties, stakeholders, schedules and penalties, and regulations governing a financial contract needs to be represented.

Indeed, the financial crisis of 2008 was in part caused by the sub-prime lending practice that encouraged high credit risk borrowers to take on mortgages at high interest rates that they had little ability to repay. These debts were pooled together and engineered to be offered as low risk asset-backed securities. These were heavily traded because of the *perceived* low risk while providing high returns. The housing market in the US slumped setting off a chain reaction that ultimately meant the mortgage-backed securities became worthless having direct effect on the capital of the major global banks. Funding dried up and more importantly, the trust that keeps the financial system performing dissolved. As a result, regulation in the financial industry has grown exponentially.

There are two scenarios where the lawyer's involvement is unavoidable:

- When the contract is partly fulfilled through code, because the lawyer can only validate its textual version [20], i.e. the prose.
- When assessing the compliance of the contract with regulations, from the point of view of both the legal requirements introduced by the regulation (e.g. on financial activities, anti-money laundering, or consumer protection), and of the effects that these regulations automatically bind to the contract (*naturalia negotii* [15]).

Therefore, we think that proper authoring of smart contracts should involve two main agents: the lawyer and the developer. The interaction between both agents should be governed by a common language. The lawyer authors and consumes contracts written in that language, while the developer uses it as a specification guiding her/his implementation. This common language should have the following properties:

– It should not alienate the lawyer, i.e. it should be as close as possible to the language of contracts s/he is used to.
– It should be expressive enough to allow the authoring of smart and *"not-so-smart"* contracts.
– It should be a Controlled Natural Language (CNL) with an unambiguous grammar. The CNL should be mappable to a logical formalism which will facilitate compliance checking with existing regulations.
– The concepts and actions described in the contract (i.e. the vocabulary) along the clauses of the contract (i.e. the rules) should be shareable across the network, which is important for both *discoverability* and *negotiation* –two defining aspects of smart contracts– by human and autonomous agents.
– It should be able to represent the actions coded in the smart contract [9], the duties and powers arising from the contract [14], and the meta-rules governing it (e.g. regulation on financial activities, Anti-Money Laundering or consumer protection).

In our previous work [8] we describe Mercury, a language to capture regulations for the purpose of compliance checking, alongside a methodology [4] to capture legal knowledge and translate it to `OWL` [5]. Mercury is based on the Semantics for Business Vocabulary and Business Rules [26] (`SBVR`), but the language of smart contracts should not forcibly be based on `SBVR`, as long as it can be mapped to a logical formalism, e.g. `OWL`, where reasoning on compliance is feasible.

In a recently published technical report, English et al. [13] investigated how distributed ledger technologies and the Semantic Web can affect one another. Indeed, the blockchain can provide secure resource identifiers (by ensuring authenticity, human-readability, and decentralisation), and ontologies can provide a unified way to understand blockchain concepts between humans, and exposing blockchain data according to an ontology enables the interlinking with other linked data and to perform reasoning.

Our proposal improves transparency, which is one of the major luring qualities of distributed ledgers, and a determining factor of the trust-less trust in the network. But doubt rises when it comes to the trust in the fact that the contract, as written by the lawyer, was correctly translated into code, i.e. the trust in the fact operational semantics faithfully represent denotational semantics. One may argue that this trust can be guaranteed if there is a mechanism $\mathcal{G}$ that automatically generates code from prose and/or a mechanism $\mathcal{C}$, potentially the inverse of $\mathcal{G}$, that proves the correspondence of the code to the prose, but a closer inspections shows that:

1. There is evidence from the literature that $\mathcal{G}$ and $\mathcal{C}$ can exist, especially from [20] and τ-Chain [6]. Indeed, if the vision of τ-Chain is possible, then there is an opportunity to go directly from denotational to operational semantics using our approach, but this *may* imply the restriction of said trust to one specific distributed ledger technology.

2. It is not really clear, at least for us, if $\mathcal{G}$ and $\mathcal{C}$ exist for shared ledgers that use stack-based languages. This is an open question that deserves closer attention, and can have one of two clear answers:
   (a) It is possible, or practically feasible, which is great news for everyone, or
   (b) It is impossible, or practically infeasible. Then it is only reasonable to ask: *is the existence of $\mathcal{G}$ and $\mathcal{C}$ a prerequisite for the establishment of said trust?* We conjecture that it is not, for two reasons:
      i. The implementation processes of existing financial contracts in the form of software is already opaque, especially to the consumer, and our proposed approach would only facilitate transparency.
      ii. Trust can be gained through the establishment of reputation: the better you are in effectively transforming your specification to code, the more reputable you are; the more reputable you are, the more trustworthy you are perceived to be.

5 Conclusion

In this position paper, we expressed our point of view on how trust in smart contrast, from a financial institution's point of view, can be enabled. It is true that cryptographic guarantees are enablers of, and integral to, trust in distributed ledger technology, but we argue that another kind of trust is needed; one that is established by a process involving lawyers.

We showed how distributed ledger technologies can vary on a deep technical level, which led to the development of tools and abstractions to help developers in programming smart contracts. These developments are essential for the technological ecosystem, but we show how most of the existing work do not take into account compliance with existing (and ever growing) regulations.

To that end, we set a list of criteria for a language necessary for the development of contracts, executed on the ledger, or not, that is close to the legal prose, transparent, and rooted in formal logic. We also identify a key research challenge, which is the ability to translate the aforementioned language to executable code.

References

1. Oraclize: "The provably honest oracle service". `http://www.oraclize.it/`, accessed: 2017-01-30
2. OWL 2 Web Ontology Language Document Overview (Second Edition). `https://www.w3.org/TR/2012/REC-owl2-overview-20121211/`, accessed: 2017-01-30
3. PriceFeed smart contract. `http://feed.ether.camp/`, accessed: 2017-01-30
4. Abi-Lahoud, E., O'Brien, L., Butler, T.: On the Road to Regulatory Ontologies, pp. 188–201. Springer Berlin Heidelberg, Berlin, Heidelberg (2014), `http://dx.doi.org/10.1007/978-3-662-45960-7_14`
5. Al Khalil, F., Ceci, M., Yapa, K., O'Brien, L.: SBVR to OWL 2 Mapping in the Domain of Legal Rules, pp. 258–266. Springer International Publishing (2016), `http://dx.doi.org/10.1007/978-3-319-42019-6_17`
6. Asor, O.: About Tau-Chain. ArXiv e-prints (Feb 2015)

7. BRADY, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. Journal of Functional Programming 23(5), 552–593 (Sep 2013)
8. Ceci, M., Al Khalil, F., O'Brien, L.: Making Sense of Regulations with SBVR. In: RuleML 2016 Challenge, Doctoral Consortium and Industry Track hosted by the 10th International Web Rule Symposium (RuleML 2016) (2016)
9. Clack, C.D., Bakshi, V.A., Braine, L.: Smart Contract Templates: essential requirements and design options. ArXiv e-prints (Dec 2016)
10. Delmolino, K., Arnett, M., Kosba, A.E., Miller, A., Shi, E.: Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. In: Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Christ Church, Barbados, February 26, 2016, Revised Selected Papers. pp. 79–94 (2016), http://dx.doi.org/10.1007/978-3-662-53357-4_6
11. Demian Lerner, S.: Rootstock. bitcoin powered smart contracts. white paper. (Nov 2015), https://uploads.strikinglycdn.com/files/90847694-70f0-4668-ba7f-dd0c6b0b00a1/RootstockWhitePaperv9-Overview.pdf
12. Demian Lerner, S.: Drivechains, sidechains, and 2-way hybrid peg designs (Jan 2016), https://uploads.strikinglycdn.com/files/27311e59-0832-49b5-ab0e-2b0a73899561/Drivechains_Sidechains_and_Hybrid_2-way_peg_Designs_R9.pdf
13. English, M., Auer, S., Domingue, J.: Block chain technologies & the semantic web: A framework for symbiotic development. Tech. rep., Technical report, University of Bonn, Germany (2016)
14. Frantz, C.K., Nowostawski, M.: From institutions to code: Towards automated generation of smart contracts. In: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W). pp. 210–215 (Sept 2016)
15. Frignani, A.: Some Basic Differences between the Common Law and the Civil Law Approach. http://www.jus.unitn.it/CARDOZO/Review/Comparative/Frignani-1997/Washingt.htm (1996), accessed: 2017-02-02
16. García-Bañuelos, L., Ponomarev, A., Dumas, M., Weber, I.: Optimized Execution of Business Processes on Blockchain. ArXiv e-prints (Dec 2016)
17. Grigg, I.: The ricardian contract. In: Proceedings. First IEEE International Workshop on Electronic Contracting, 2004. pp. 25–31 (July 2004)
18. Grigg, I.: On the intersection of Ricardian and Smart Contracts. http://iang.org/papers/intersection_ricardian_smart.html (Feb 2017), accessed: 2017-01-30
19. Hull, R., Batra, V.S., Chen, Y.M., Deutsch, A., Heath III, F.F.T., Vianu, V.: Towards a Shared Ledger Business Collaboration Language Based on Data-Aware Processes, pp. 18–36. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-46295-0_2
20. Idelberger, F., Governatori, G., Riveret, R., Sartor, G.: Evaluation of Logic-Based Smart Contracts for Blockchain Systems, pp. 167–183. Springer International Publishing, Cham (2016), http://dx.doi.org/10.1007/978-3-319-42019-6_11
21. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. CCS '16, ACM, New York, NY, USA (2016), http://doi.acm.org/10.1145/2976749.2978309
22. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)

23. Pettersson, J., Edström, R.: Safer smart contracts through type-driven development. Ph.D. thesis, Master's thesis, Dept. of CS&E, Chalmers University of Technology & University of Gothenburg, Sweden (2015)
24. Reijers, W., O'Brolcháin, F., Haynes, P.: Governance in blockchain technologies & social contract theories. Ledger 1(0), 134–151 (2016), `http://www.ledgerjournal.org/ojs/index.php/ledger/article/view/62`
25. Seijas, P.L., Thompson, S., McAdams, D.: Scripting smart contracts for distributed ledger technology. Cryptology ePrint Archive, Report 2016/1156 (2016), `http://eprint.iacr.org/2016/1156`
26. Semantics of Business Vocabulary and Business Rules (SBVR) Version 1.3. `http://www.omg.org/spec/SBVR/1.3/PDF` (May 2015), `http://www.omg.org/spec/SBVR/1.3/PDF`
27. Swan, M.: Blockchain Temporality: Smart Contract Time Specifiability with Blocktime, pp. 184–196. Springer International Publishing, Cham (2016), `http://dx.doi.org/10.1007/978-3-319-42019-6_12`
28. Szabo, N.: Formalizing and Securing Relationships on Public Networks. `https://web.archive.org/web/20050217172626/http://www.firstmonday.dk/ISSUES/issue2_9/szabo/index.html` (1997), accessed: 2017-01-25
29. Szabo, N.: The Idea of Smart Contracts. `https://web.archive.org/web/20160831070942/http://szabo.best.vwh.net/smart_contracts_idea.html` (1997), accessed: 2017-01-25
30. Tschorsch, F., Scheuermann, B.: Bitcoin and beyond: A technical survey on decentralized digital currencies. IEEE Communications Surveys and Tutorials 18(3), 2084–2123 (2016), `http://dx.doi.org/10.1109/COMST.2016.2535718`
31. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum Project Yellow Paper 151 (2014)
32. Zhang, F., Cecchetti, E., Croman, K., Juels, A., Shi, E.: Town crier: An authenticated data feed for smart contracts. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 270–282. CCS '16, ACM, New York, NY, USA (2016), `http://doi.acm.org/10.1145/2976749.2978326`

Defining the Ethereum Virtual Machine for Interactive Theorem Provers

Yoichi Hirai

Ethereum Foundation `yoichi@ethereum.org`

Abstract. Smart contracts in Ethereum are executed by the Ethereum Virtual Machine (EVM). We defined EVM in Lem, a language that can be compiled for a few interactive theorem provers. We tested our definition against a standard test suite for Ethereum implementations. Using our definition, we proved some safety properties of Ethereum smart contracts in an interactive theorem prover Isabelle/HOL. To our knowledge, ours is the first formal EVM definition for smart contract verification that implements all instructions. Our definition can serve as a basis for further analysis and generation of Ethereum smart contracts.

1 Introduction

Ethereum is a protocol for executing a virtual computer in an open and distributed manner. This virtual computer is called the *Ethereum Virtual Machine* (EVM). The programs on EVM are called Ethereum *smart contracts*. A deployed Ethereum smart contract is public under adversarial scrutiny, and the code is not updatable. Most applications (auctions, prediction markets, identity/reputation management etc.) involve smart contracts managing funds or authenticating external entities. In this environment, the code should be trustworthy.

The developers and the users of smart contracts should be able to check the properties of the smart contracts with widely available proof checkers. Our EVM definition is written in Lem, which can be translated into popular interactive theorem provers Coq [1], Isabelle/HOL [19] and HOL4 [23]. We used our EVM definition and proved safety properties of some smart contracts in Isabelle/HOL.

Our contributions are as follows:

- we gave a formal specification of the interface between a smart contract execution and the rest of the world (Sec. 4);
- we defined EVM in a way portable to different interactive theorem provers (Isabelle/HOL and HOL4) and a programming language OCaml, during which we found some subtle differences between the specification (the Yellow Paper [26]) and the implementations (Sections 5 and 6);
- we tested the executable part of our EVM definition against the VM test suite, which validates existing Ethereum node implementations (Subsection 5.3); we found unsearched corner cases in the test suite;
- we used our EVM definition to prove invariants and safety properties of Ethereum smart contracts (Sec. 7).

2 Choice of the Goal and the Tool

2.1 Goal: Which Programming Language to Formalize

Considerations around Solidity Although ultimately all Ethereum smart contracts are deployed as EVM bytecode, the bytecode is rarely directly written. The most popular programming language Solidity [3] has a rich syntax but no specification. The only definition of Solidity is the Solidity compiler implementation, which compiles Solidity programs into EVM bytecode.

The Solidity compiler is written in C++. Importing the C++ code in a theorem prover is nearly impossible because the definition of the whole C++11 language has not been formalized although some of the hardest aspects of the language have been addressed: concurrency [6], inheritance [21] etc.

It is feasible to verify a compiler with optimization (e.g. CompCert [14] and CakeML [13]). Something similar for Solidity would require formalization of both Solidity and EVM before correctness of the compiler can be stated.

Considerations on EVM There are drawbacks of verifying EVM bytecode:

- most developers and users do not read EVM bytecode;
- the EVM architecture might become obsolete after the protocol adopts one of the proposed new architectures (EVM 1.5 that introduces function calls or EVM 2.0 which is based on WebAssembly [4]).

The first point can be, in the future, mitigated by translating static assertions in Solidity into EVM bytecode. The second point is, in fact, milder compared with the fast changes of the Solidity compiler. When the virtual machine architecture changes, all Ethereum implementations need to implement the change. This makes EVM change slower than the Solidity compiler.

EVM is an attractive formalization target. It is a stack-machine with a simple instruction-encoding and fully sequential execution. The simplicity of the EVM architecture resulted in just over 2,000 lines of formal definition. EVM has an English specification called the Yellow Paper (Fig. 1) clear enough to allow multiple implementations to be developed independently[1]. Also, since any disagreements among implementations hurt the availability of the network, the community has implemented test suites to compare EVM implementations. We use one of these test suites to test our EVM definition.

2.2 Tool: Formalization in Which Language

We intend our EVM definition as a basis for smart contract verification. The verification should be done in a precise manner. Model checkers are not capable of doing this because they cannot treat the huge state space: a smart contract can store up to 2^{256} 256-bit machine words permanently (the resource usage is limited

[1] Several entities develop Ethereum clients in Python, C++, Rust, Java, Scala and Go, and each contains its own EVM implementation.

0s: Stop and Arithmetic Operations

All arithmetic is modulo 2^{256} unless otherwise noted.

Value	**Mnemonic**	δ	α	**Description**
0x01	ADD	2	1	Addition operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]$
$\vdots$				
0x08	ADDMOD	3	1	Modulo addition operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[2] = 0 \\ (\boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]) \mod \boldsymbol{\mu}_{\mathbf{s}}[2] & \text{otherwise} \end{cases}$ All intermediate calculations of this operation are not subject to the 2^{256} modulo.

Fig. 1. A short excerpt from the Yellow Paper [26]. The symbol δ (resp. α) stands for the number of deleted (resp. added) stack elements. $\boldsymbol{\mu}_{\mathbf{s}}[i]$ is the i-th stack element before the instruction execution. $\boldsymbol{\mu}'_{\mathbf{s}}[i]$ is the i-th stack element afterwards.

only economically). Such big state spaces can better be dealt with interactive theorem provers. Instead of specifying EVM in one particular theorem prover, we chose a framework called Lem [16] because definitions in Lem can be translated into some popular theorem provers: Coq [1], Isabelle/HOL [19] and HOL4 [23].

One potential alternative is the K-Framework [22]. The K-Framework is a tool specifically engineered for defining programming languages. We chose Lem and its translation targets for their larger user base[2] and their longer history.

3 A Brief Description of the Ethereum Virtual Machine

Some of our design choices and challenges can be described only after an overview of EVM. We just describe EVM as a state machine that executes programs. We omit the underlying techniques that support distributed execution.

3.1 States

In EVM, apart from several global parameters, most states are stored in accounts. EVM has a partial map from *addresses* (160-bit words) to account states. An *account state* contains code, storage, nonce and the balance. The *code* is a sequence of bytes. The *storage* is a mapping from a machine word (an EVM *machine word* has 256 bits) to a machine word. The *nonce* is an ever-increasing machine word. The *balance* is also a machine word, representing some transferable value that can be paid as fees to run EVM. When the code is not empty,

[2] The Coq users' mailing list has 1,404 subscribers while the K-Framework's has 127 at the time of writing.

the code controls the account; such an account is called a *contract*. Otherwise, the account is controlled by the holder of a private key corresponding to the address; such an account is called an *external account*. The code, when exists, encodes a sequence of instructions. Instructions are all encoded in a single byte except for the PUSH instructions, which contain immediate values.

3.2 State Transitions

An external account can initiate a *transaction* either by creating a contract or by calling an account. Once a transaction is initiated, the whole state transition of EVM is deterministic. We do not describe the contract creation by an external account because a contract's state after creation is publically checkable.

Both external accounts and contracts can *call* an account. When an account calls an account, the call is accompanied with transferred balance, gas, and data. The transferred balance is deposited to the called account. The gas regulates the resource consumption during the call. When the called account is an external account, a simple balance transfer happens. Otherwise, when the called account is a contract, after the balance transfer, the called contract's code is executed. The code execution can alter the storage of the executing account. The execution can read all accounts' balances and codes.

The resource consumption of the code execution is capped by the amount of gas that the initiating external account pays for. Executing an instruction consumes some amount of gas. When the gas is exhausted, the execution fails (*out-of-gas failure*). Such failures revert all state changes performed during the current call, except gas consumption.

A contract can call an account by executing the `CALL` instruction. The ensuing balance transfer and code execution belong to the same transaction as the calling code execution. The calling contract can limit the resource consumption in the called contract by choosing the amount of gas passed on. When the inner call fails, the side-effects of the inner call is reverted (except gas consumption) but the side-effects of the outer call remains intact. The outer call is informed of such a failure through a return value of the `CALL` instruction.

A transaction belongs to a block. A *block* is a unit of agreement among Ethereum nodes. EVM has special instructions that reads the *block number* and the cryptgraphic hash values of some previous blocks. Since a block specifies a previous block but not a unique successor, blocks in the network form a tree in general, but, as far as the states of EVM are concerned, only one branch in the tree matters. Because of this, we can think of EVM as a sequencially executed machine.

4 Interface of a Contract Invocation

4.1 Boundary between the System and the Environment

We are interested in propositions of the form: whatever the environment does, the system responds in a desired manner. Before we try to specify the desired behavior, we need to identify the system and the environment. The choice is not straightforward because multiple parties are involved in EVM.

One way is to say that the system is the contract. In that case, the environment contains anything out of EVM and all accounts on EVM except the contract under verification. In our development, the system is a single contract invocation, which is even narrower than a single account (Fig. 2 (b)). The difference can be seen in the following scenario. The environment can call into the contract. The contract can reply by calling an account. The environment can, depending on the states of accounts that we do not control, call our contract again. This is called *reentrancy*. During reentrancy, the storage and the balance of our contract might change. We chose to model the reentrancy as part of the environment. We explain this choice in Subsection 6.3.

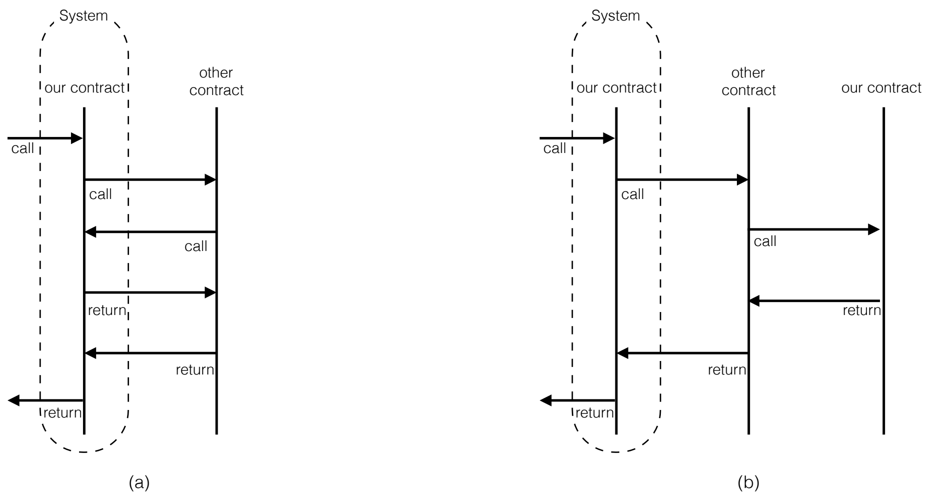


Fig. 2. Different choices of system-environment boundaries during reentrancy. Both pictures describe the same situation, but have different boundaries between the system and the environment. (a) When the system is our contract, the reentrant call is a part of the system. (b) When the system is a single invocation of our contract, the reentrant call is a part of the environment. Both are sound, but we chose (b) because it matches the program sytax where `CALL` instructions are followed by the next operations in the same message call, not the next operations in the reentrant call.

4.2 Input and Output of a Deployed Ethereum Smart Contract

In Subsection 4.1, we have set the boundary between the smart contract and the environment. Next, we identify their interaction. The specification of the interface is particularly important because it can be used to specify higher level languages for Ethereum smart contracts. Our most concrete contribution is our

EVM definition in Lem, so we show some snippets in this section and explain the syntax.

The interaction between the contract and the environment always starts with the envrionment's call into the contract. The environment can call into the contract with the following information:

```
type CALL_ENV = ⦇
 callenv_gaslimit : W_256;          (* the current invocation's gas limit *)
  callenv_value : W_256;                   (* the amount of Eth sent along*)
  callenv_data : LIST BYTE;                        (* the data sent along *)
  callenv_caller : ADDRESS;                      (* the caller's address *)
  callenv_timestamp : W_256;       (* the timestamp of the current block *)
  callenv_blocknum : W_256;     (* the block number of the current block *)
 callenv_balance : ADDRESS → W_256; (* the balances of all accounts. *) ⦈
```

The whole syntax defines a *record* type with seven *fields*. A value of CALL_ENV consists of seven values each accessible under a field name. The field names are italicized. Each field name is annotated with a type of the associated value. W_{256} denotes the type of 256-bit machine words and ADDRESS 160-bit machine words. LIST BYTE is the type of lists of bytes. The arrow type ADDRESS $\rightarrow$ W_{256} is the type of total functions that take an ADDRESS and return a W_{256}. This definition is useful not only for reasoning about EVM bytecodes but also for desining high level languages that would be compiled into EVM. Ethereum contracts written in any language needs to take the combination of data above.

The environment can also make a called account return or fail after our contract makes a call. Together, the environment's possible actions are described by the following *variant type* ENVIRONMENT_ACTION, whose value can be the label EnvironmentCall together with a value of CALL_ENV, the label EnvironmentRet together with a value of RETURN_RESULT, or the label EnvironmentFail. It is automatically understood that values with different labels are different. This definition describes everything that can happen to an Ethereum contract. If we have checked these cases, we have enumerated all possibilities.

```
 type ENVIRONMENT_ACTION =
| EnvironmentCall of CALL_ENV (* the environment calls the contract *)
| EnvironmentRet of RETURN_RESULT         (* the environment returns *)
| EnvironmentFail                           (* the environment fails *)
```

We omit the definition of RETURN_RESULT and many other symbols. The whole formalization is publicly available[3].

The contract can also make its move: calling another account, making a delegate call, creating a contract, failing, destroying itself, or returning. A *delegate call* runs a potentially different account's code on the caller's account.

```
 type CONTRACT_ACTION =
| ContractCall of CALL_ARGUMENTS                 (* calling an account *)
| ContractDelegateCall of CALL_ARGUMENTS             (* library call *)
| ContractCreate of CREATE_ARGUMENTS          (* deploying a contract *)
```

[3] https://github.com/pirapira/eth-isabelle/tree/wtsc01

| CONTRACTFAIL (* `failing back to the caller` *)
| CONTRACTSUICIDE (* `destroying itself and returning to the caller` *)
| CONTRACTRETURN of LIST BYTE (* `returning to the caller` *)
This definition describes everything that an Ethereum contract can do. When a high level language is designed for Ethereum, it's desirable that the language can cause all of these actions. Moreover, since the input-output interface is defined in an interactive theorem prover, the actions can be universally ($\forall$) or existentially ($\exists$) quantified in logical formulas that specify Ethereum smart contracts.

5 Formalizing the Deterministic Contract Execution

The Yellow Paper [26] specifies EVM's behavior uniquely for all possible inputs (either a contract creation or a message call) coming from external accounts. After no state transitions, the resulting state is left ambiguous. The original purpose of such determinism is to prevent the nodes from disagreeing, but the determinism also simplifies the formalization. We were able to formalize consecutive execution of instructions in our contract as a total function that produces a state. The deterministic definitions of the program semantics occupy 2,000 lines of Lem code. The determinism also made it straightforward to test this part of the EVM definition against a standard test suite (Subsection 5.3)[4].

We initially tried to implement EVM available in the latest Ethereum network. During the VM tests we found that EVM should price instructions differently depending on block numbers, so we modeled this as to pass the tests.

5.1 Defining Execution Contexts

During the formalization, we have identified the runtime state of EVM. While EVM is executing an account's code, EVM has access to the stack, the memory, the memory usage counter, the storage, the program counter, the balances of all accounts, the caller, the value sent along the current call, the data sent along the current call, the initial state kept for reverting into, the external account that originated the transaction, the codes on all addresses, the current block, the remaining gas, existence of accounts, and the list of touched storage indices. Everything except the last one is necessary for EVM execution. The last piece spares enumerating all storage indices while testing. These data are packed into a record type VARIABLE_CTX. Moreover, EVM can read the program and the address of the currently running contract. These data are packed into a record type CONSTANT_CTX.

An instruction can result in the following cases:

type INSTRUCTION_RESULT =
| INSTRUCTIONCONTINUE of VARIABLE_CTX (* `the execution continues.` *)
| INSTRUCTIONANNOTATIONFAILURE (* `annotation was false.` *)

[4] If nondeterminism existed in the EVM execution, at least, we would need to choose a representation of nondeterminism that works both in interactive theorem provers and in OCaml.

```
| InstructionToEnvironment of
   contract_action                            (* the contract's move *)
  * storage                                 (* the new storage content *)
  * (address → w256)              (* the new balance of all accounts *)
  * list w256          (* the list of possibly changed storage indices *)
  * maybe (variable_ctx * integer * integer)      (* continuation *)
```
The asterisk $*$ composes the type of *tuples*.

5.2 Defining Deterministic Contract Execution

Using the above definitions, we can define a function that operates an instruction on the execution environments:

```
val instruction_sem : variable_ctx → constant_ctx → inst →
instruction_result

let instruction_sem v c inst1 =
 subtract_gas (meterGas inst1 v c)
 (match inst1 with
 | Arith ADD → stack_2_1_op v c (fun a b → a + b)
 | Arith ADDMOD → stack_3_1_op v c
   (fun a b divisor →
    (if divisor = 0 then 0
     else word256FromInteger ((uint a + uint b) mod (uint divisor))))
 ⋮
 end)
```
where meterGas calculates the exact gas consumption of the executed instruction. We can repeat the semantics of single instructions to define the semantics of a whole program (`JUMP` instruction is not special because all instructions, including `JUMP`, change the program counter).

The type PROGRAM_RESULT is similar to INSTRUCTION_RESULT. The program semantics takes artificial step counters that disallow infinite execution because, in Isabelle/HOL, every function must be provably terminating[5]. This does not cause imprecision because any actual execution can be simulated with a sufficiently large step counter value.

```
val program_sem : variable_ctx → constant_ctx → int → nat →
program_result
```
During the modeling, we found that the Yellow Paper computes gas differently from the implementations. The subtlest case was the computation of gas for memory accesses: when a contract accesses the memory on addresses spanning from $2^{256} - 255$ to 1, the gas calculation differed in the Yellow Paper and in implementations. The Yellow Paper used 1 as the maximal touched address while all checked implementations used $2^{256} + 1$ instead. Since all implementations agreed, we filed a fix in the Yellow Paper.

[5] We can guarantee termination by the gas, but the proof is non-trivial (currently 980 lines of Isabelle code).

5.3 Testing the Deterministic Contract Interpreter

We tested our definition against a test suite called VM tests [2]. The test suite (together with other test suits) keep different Ethereum implementations conformant. We used VM tests to ensure conformance of our EVM definition. Lem automatically translated the definition into OCaml. The OCaml translation was then combined with a test case runner we wrote in OCaml (Fig. 3).

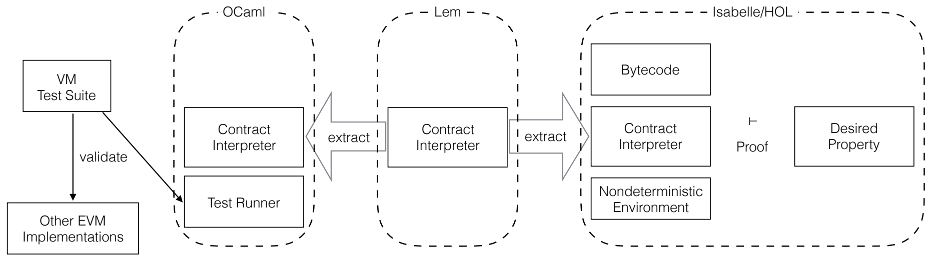


Fig. 3. Our Lem definition can be extracted into OCaml and Isabelle/HOL. We tested the OCaml extraction against the standard VM test suite. Using the Isabelle/HOL extraction, we proved safety properties about some bytecodes. In this figure, the VM test suite and other EVM implementations are not our contributions.

During the testing, we uncovered problems like:

- wrong word-to-integer conversion during `ADDMOD` in our EVM definition;
- different endianness between OCaml extraction and Isabelle/HOL extraction, due to our wrong direction; and
- small mistakes in the Yellow Paper, in most cases about modulo-2^{256}.

The number of successful test cases is 40,619 while no tests fail. We skipped 24 test cases because they involve running multiple contracts, and we chose to model only a single contract's execution deterministically. Running these 24 cases would involve major enhancements in our test runner: emulating multiple instances of our EVM model and communication among them.

In addition, we measured the code coverage of the VM test suite on the generated OCaml code. We found that `DELEGATECALL` instruction is never called, that `CALL` instruction is never called with insufficient balance to be transferred, that some instructions were never called with insufficient stack elements, and that the gas calculation after the latest changes is not tested. Although recent protocol changes are often tested in other test suites, the VM test suite can be complemented with these cases.

6 Formalizing the Nondeterministic Environment

We define the nondeterministic environment as a binary relation between a prestate of type (ACCOUNT_STATE * PROGRAM_RESULT) and a poststate of type

(ACCOUNT_STATE * VARIABLE_CTX). This binary relation encodes the environment's freedom. The binary relation is parametrized with an invariant (to be speculated by the verification practitioner) of the contract under verification, which limits the state changes on the contract during reentrancy. If this limitation makes the same speculated invariant provable, the invariant can be deemed established following an informal argument given in Subsection 6.3.

6.1 Implicit Balance Changes

We assume that the balances of accounts change freely while our contract is not executed. This assumption subsumes the payment for the gas. The storage of other accounts might change too. However, the balance of our contract is assumed not to decrease when there are no calls being executed on it[6].

On the other hand, the balance of our contract might increase when another account executes `SUICIDE` instruction, specifying our account as the recipient of the remaining balance. So the environment can freely increase the balance of our contract. We are assuming that the balance increase does not overflow (which seems to hold currently because the total balance of all accounts is below 2^{80} while the balances can be counted up to $2^{256} - 1$).

6.2 Gas Consumption During Calls

When our contract calls an account, the available amount of gas might decrease. We modeled this as a completely nondeterministic change. This treatment admits the actual gas decrease as one possibility, and it shortens the proof goals during brute-force proving. Without this treatment, during the symbolic execution described in Sec. 7, we saw the symbolic states grow rapidly because the remaining gas was represented as a long sequence of subtractions. With the nondeterministic choice, the remaining gas in the symbolic state is reduced into one variable after each call.

6.3 Modeling of Reentrancy as an Adversarial Environment's Step

We have freedom: the nested execution under reentrancy can either be a part of the system or the environment. The choice influences the proof structure. If the reentrancy is part of the system, proofs of safety properties need to explore all possibilities in the nested reentrant calls. If the reentrancy is part of the environment, the reentrancy is an adversarial step that changes the account state in some arbitrary ways. We chose the latter way because this matches better with the syntax of EVM bytecode, and it serves as the first approximation before building a bigger EVM definition involving call stacks.

We assume that the reentrancy can change the contract's account state (the balance and the storage) following a speculated invariant. Using this assumption,

[6] This property can be established only by checking all lines in the Yellow Paper that changes the balance.

we prove the same invariant on the outer call. If we finish proving this, we can perform mathematical induction over the number of nesting reentrancy to check that all message calls keep the invariant. This mathematical induction has not been formalized in any interactive theorem provers only because substantial development is required before stating the goal.

6.4 Cleanup of an Account after Self-Destruction

When a contract executes `SUICIDE` instruction, the storage and the code of the account are cleared not immediately but at the end of a transaction. The timing of this cleanup is determined by the adversarial environment. However, we know that the cleanup does not occur while a contract is still running.

7 Example Verification of Smart Contracts

To show the utility of our definitions, we have developed three example proofs in Isabelle/HOL.

Invariant of a Program that Always Fails As the shortest example, we prepared a smart contract that always fails. We proved that the code remains intact forever; in other words the contract does not execute `SUICIDE` operations.

Invariant of a Program that Fails on Reentrance The next example features reentrancy, which enabled an external account "to put ∼$60M under her control" [5] during "the DAO" incident, where a coding mistake in a contract allowed leakage of the fund. We implemented a contract (Fig. 4) that calls an account but fails on reentrance. We proved that its storage values always stay within the specified values (Fig. 5) even when reentrant calls are attempted.

```
abbreviation fail_on_reentrance_program :: "inst list"
where
"fail_on_reentrance_program ==
  Stack (PUSH_N [0]) # Storage SLOAD # Dup 1 # Stack (PUSH_N [2]) #
  Pc JUMPI # Stack (PUSH_N [1]) # Arith ADD # Stack (PUSH_N [0]) #
  Storage SSTORE # Stack (PUSH_N [0]) # Stack (PUSH_N [0]) #
  Stack (PUSH_N [0]) # Stack (PUSH_N [0]) # Stack (PUSH_N [0]) #
  Stack (PUSH_N [0xabcdef]) # Stack (PUSH_N [30000]) # Misc CALL #
  Arith ISZERO # Stack (PUSH_N [2]) # Pc JUMPI # Stack (PUSH_N [0]) #
  Stack (PUSH_N [0]) # Storage SSTORE # Misc STOP # []"
```

Fig. 4. An Ethereum smart contract that calls an account but fails on reentrancy. The expression in this figure defines a list of instructions in Isabelle/HOL. See the Yellow Paper [26] for intuitive descriptions of instructions.

```
inductive fail_on_reentrance_invariant :: "account_state ⇒ bool"
where
  depth_zero:
    "account_address st = fail_on_reentrance_address ⟹
     account_storage st 0 = 0 ⟹
     account_code st = program_of_lst
       fail_on_reentrance_program program_content_of_lst ⟹
     account_ongoing_calls st = [] ⟹ account_killed st = False ⟹
     fail_on_reentrance_invariant st"
| depth_one:
    "account_code st = program_of_lst
       fail_on_reentrance_program program_content_of_lst ⟹
     account_storage st 0 = 1 ⟹
     account_address st = fail_on_reentrance_address ⟹
     account_ongoing_calls st = [(ve, 0, 0)] ⟹
     account_killed st = False ⟹
     vctx_pc ve = 28 ⟹ vctx_storage ve 0 = 1 ⟹
     vctx_storage_at_call ve 0 = 0 ⟹
     fail_on_reentrance_invariant st"
```

Fig. 5. An invariant of the contract that fails on reentrancy, expressed in Isabelle/HOL. The whole invariant is a disjunction of two clauses: `depth_zero` holds when the contract is not running while `depth_one` holds when the contract has called an account.

Safety Property of a Compiled Program We proved a safety property of a realistic Ethereum contract with 501 instructions produced by the Solidity compiler. The safety property states that, if the storage has a flag set, only the owner recorded in the storage can decrease the balance or change the storage.

The proof is a brute-force symbolic execution in Isabelle/HOL. The proof contains repetitive 5,000 lines. It takes three hours for Isabelle to check the proof. There is huge room of improvements. Since the contract contains no loops, it should be possible to automate the whole proof. The proof checking time would be much shorter with more advanced techniques that appear in the next section.

8 Related Work

The idea and the techniques in this paper are not new, except that we apply these to EVM. Boyer and Yu [9] used a theorem prover Nqthm to model MC68020 processor, and checked correctness of a binary search implementation. Fox [10] modeled the ARM6 micro-architecture, which is far more complex than EVM, in HOL and validated it against the instruction set architecture. The deterministic part of our EVM definition happens to be in the form of functional big-step semantics [20] although our proof development is not advanced enough to enjoy its merits. The idea of combining theorem proving and testing is not new either even in the industry [7].

The literature suggests our future paths as well. Myreen, Fox and Gordon [18] defined Hoare logic for ARM machine code. Myreen, Gordon and Slint [17] further developed techniques for decompiling machine code with loops into recursive HOL functions. The approach of Kennedy et al. [11] is to formalize the machine code and then to build gradually structured programming method in Coq. Alternatively, we might try to build a higher level language that compiles into EVM. Jinja (Jinja is not Java) [12] demonstrates language specification and implementation in Isabelle/HOL. CakeML [13] is a programming language defined in Lem with a verified compiler into x86-64.

Some automatic analysis tools have been developed for Ethereum smart contracts. Oyente [15] implements abstract interpretation of EVM in Python with constraint solving using Z3. The tool can automatically detect several classes of vulnerabilities with false positives. Removing these classes of vulnerabilities does not guarantee lack of bugs. The tool does not implement all instructions. Bhargavan et al. [8] define translations from a fragment of Solidity and from EVM into F$*$, a functional programming language with a rich type system. They can detect diversion from certain programming disciplines in Solidity. They can also estimate an upper bound of gas consumption of an EVM program. They do not mention testing their translations against implementations[7].

9 Challenges and Future Work

Currently, verifying a realistic contract take around three hours on a Lenovo Ideapad 500S. Most of the time is spent in out-of-gas failures at various points in the program. One way to improve the situation is to set up a semantics that squashes all out-of-gas failures as a single case.

Another direction is to make the reasoning compositional. In other words, we should enable carrying over verification of small program snippets into verification of larger programs. This involves developing a syntax for properties (program logics) that is robustly concise during the compositional reasoning. Some program logics for machine code exist: e.g. Tan and Appel [24] and Myreen [18].

We have not tested the nondeterministic parts of our development. Also we have not validated our development against the blockchain history of the Ethereum network. The executable part of our model is considerably smaller than the whole EVM. If we model the whole EVM, we can try more standard test suites on our EVM definition. The modelling of the whole EVM would be the first step towards implementing a reference EVM out of our definitions.

The interactive theorem provers are designed for honest users. When a proof assistant admits a theorem that looks like falsehood, the proof assistant is called *Pollack-super-inconsistent*. Coq and Isabelle are known to be Pollack-super-inconsistent with auxiliary definitions and notations [25]. When falsehood seems provable, subtler errors can also creep in. For protecting users from malicious verification results, we need faithful presentation of the proven properties.

[7] One of the authors explained that the work had been done in a hackathon and the codebase had not been touched since.

For verifying smart contracts in more human-friendly languages, we can either formalize existing languages or build a compiler gradually in a theorem prover. The first approach poses the burden of developing and maintaining an up-to-date machine-readable definition of the language. The second approach poses the burden of integration with the ecosystem, where the contracts need to interface with JavaScript libraries and where developers need to be familiarized.

10 Conclusion

We defined EVM so that interactive theorem provers can reason about Ethereum smart contracts. Our EVM definition contains all instructions. We used our EVM definition in Isabelle/HOL and proved safety properties and invariants of Ethereum contracts in the presence of reentrancy. As a side effect, we discovered several problems in the specification; we requested eleven fixes to the Yellow Paper. We found thirteen code paths in our model that the VM test suite did not touch. We demonstrated formal executable specification is effective for verifying smart contracts, for testing the specification, and for measuring code coverage of virtual machine tests. We expect our development to be a basis for more sophisticated smart contract verification frameworks and for verified compilers from/to EVM bytecode.

Acknowledgments. We thank Sami Mäkelä for enabling Isabelle and Coq extractions and finding mistakes in our formalization. We thank Sidney Amani, Christian Reitwießner and the anonymous referees for their time and valuable comments on this paper.

References

1. The Coq proof assistant. https://coq.inria.fr/, accessed: 2016-12-19
2. Ethereum VM tests. https://github.com/ethereum/tests/tree/develop/VMTests, accessed: 2017-01-02
3. Solidity 0.4.8-develop documentation. https://solidity.readthedocs.io/, accessed: 2016-12-19
4. WebAssembly. http://webassembly.org/, accessed: 2016-12-16
5. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts. Cryptology ePrint Archive (2016), http://eprint.iacr.org/2016/1007
6. Batty, M., Owens, S., Sarkar, S., Sewell, P., Weber, T.: Mathematizing C++ concurrency. SIGPLAN Not. 46(1), 55–66 (2011)
7. Becker, H., Crespo, J.M., Galowicz, J., Hensel, U., Hirai, Y., Kunz, C., Nakata, K., Sacchini, J.L., Tews, H., Tuerk, T.: Combining mechanized proofs and model-based testing in the formal analysis of a hypervisor. In: FM 2016. pp. 69–84. Springer (2016)
8. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. pp. 91–96. PLAS '16, ACM (2016)

9. Boyer, R.S., Yu, Y.: Automated proofs of object code for a widely used microprocessor. J. ACM 43(1), 166–192 (1996)
10. Fox, A.: Formal specification and verification of ARM6, pp. 25–40. Springer (2003)
11. Kennedy, A., Benton, N., Jensen, J.B., Dagand, P.E.: Coq: The world's best macro assembler? pp. 13–24. PPDP '13, ACM (2013)
12. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Trans. Progr. Lang. Syst. 28(4), 619–695 (2006)
13. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: A verified implementation of ML. pp. 179–191. POPL '14, ACM, New York, NY, USA (2014)
14. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM 52(7), pp. 107–115 (2009)
15. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. pp. 254–269. CCS '16, ACM (2016)
16. Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.: Lem: Reusable engineering of real-world semantics. SIGPLAN Not. 49(9), 175–188 (2014)
17. Myreen, M.O., Gordon, M.J.C., Slind, K.: Decompilation into logic–improved. In: FMCAD 2012. pp. 78–81 (2012)
18. Myreen, M.O., Fox, A.C.J., Gordon, M.J.C.: Hoare logic for ARM machine code, pp. 272–286. Springer (2007)
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer (2002)
20. Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K.: Functional big-step semantics. In: ESOP'16. pp. 589–615. Springer (2016)
21. Ramananandro, T., Dos Reis, G., Leroy, X.: Formal verification of object layout for C++ multiple inheritance. SIGPLAN Not. 46(1), 67–80 (2011)
22. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming 79(6), 397–434 (2010)
23. Slind, K., Norrish, M.: A brief overview of HOL4, pp. 28–32. Springer (2008)
24. Tan, G., Appel, A.W.: A compositional logic for control flow. In: VMCAI 2006. pp. 80–94. Springer (2006)
25. Wiedijk, F.: Pollack-inconsistency. Electron. Notes Theor. Comput. Sci. 285, 85–100 (2012)
26. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger–EIP-150 revision. http://paper.gavwood.com/, accessed: 2016-12-19

A Proof-of-Stake protocol for consensus on Bitcoin subchains

Massimo Bartoletti, Stefano Lande, and Alessandro Sebastian Podda

Università degli Studi di Cagliari, Italy

Abstract. Although the main purpose of the Bitcoin blockchain is to record currency transfers, Bitcoin transactions can also carry a few bytes of metadata. Smart contracts built upon Bitcoin exploit this feature to store a tamper-proof historical record of their transactions. The sequence of these transactions forms a *subchain* of the Bitcoin blockchain, which usually does not interfere with the transfers of bitcoins recorded therein. A subchain is *consistent* when it represents a legit execution of the smart contract. A crucial issue is how to make it difficult for an adversary to subvert the execution of the smart contract by making its subchain inconsistent. The current approaches either postulate that subchains are always consistent, or give weak guarantees about their security (for instance, they are susceptible to Sybil attacks). We propose a consensus protocol, based on Proof-of-Stake, that incentivizes nodes to consistently extend the subchain. We empirically evaluate the security of our protocol, and we show how to exploit it as the basis for smart contracts on the Bitcoin blockchain.

1 Introduction

Recently, cryptocurrencies like Bitcoin [20] have pushed forward the concept of decentralization, by ensuring reliable interactions among mutually distrusting nodes in the presence of a large number of colluding adversaries. These cryptocurrencies leverage on a public data structure, called *blockchain*, where they permanently store and timestamp all the messages exchanged by nodes. Adding new blocks to the blockchain (called *mining*) requires to solve a moderately difficult cryptographic puzzle. The first miner who solves the puzzle earns some virtual currency (some fresh coins for the mined block, and a small fee for each transaction included therein). In Bitcoin, miners must invert a hash function whose complexity is adjusted dynamically in order to make the average time to solve the puzzle ∼10 minutes. Instead, removing or modifying existing blocks is computationally unfeasible: roughly, this would require an adversary with more *hashing power* than the rest of all the other nodes. If modifying or removing blocks were computationally easy, an attacker could perform a *double-spending* attack where he pays some amount of coins to a merchant (by publishing a suitable transaction in the blockchain) and then, after he has received the item he has paid for, removes the block containing the transaction. According to the folklore, Bitcoin would resist to attacks unless the adversaries control the majority of total computing power of the Bitcoin network. Even though some vulnerabilities have been reported in the literature (see Section 4), in practice Bitcoin has worked surprisingly well so far: indeed, the known successful attacks to Bitcoin are standard hacks or frauds [16], unrelated to the Bitcoin protocol.

The idea of using Bitcoin and its blockchain as the basis for *smart contracts* [22] — i.e., decentralized applications beyond digital currency — has been explored by several recent works. For instance, [3,7] propose protocols for secure multiparty computations and fair lotteries, *Blockstore* [8] is a key-value database with get/set operations; *CounterParty* [11] extends Bitcoin with advanced financial operations (like e.g., creation of virtual assets, payment of dividends, *etc.*), by embedding its own messages in Bitcoin transactions.

Although the Bitcoin blockchain is primarily intended to trade currency, its protocol allows clients to embed a few extra bytes as metadata in transactions. Besides the above-mentioned BlockStore and Counterparty, many other platforms for smart contracts exploit these metadata to store a persistent, timestamped and tamper-proof historical record of all their messages [1,5]. Usually, metadata are stored in `OP_RETURN` transactions [2], making them meaningless to the Bitcoin network and unspendable. With this approach, the sequence of platform-dependent messages forms a *subchain*, whose content can only be interpreted by the nodes that execute the platform (we refer to them as *meta-nodes*, to distinguish them from Bitcoin nodes). However, since the platform logic is separated from the Bitcoin logic, a meta-node can append to the subchain transactions with metadata which are meaningless for the platform — or even *inconsistent* with the intended execution of the smart contract. As far as we know, none of the existing platforms use a secure protocol to establish if their subchain is consistent. This is a serious issue, because it either limits the expressiveness of the smart contracts supported by these platforms (which must consider all messages as consistent, so basically lose the notion of state), or degrades the security of contracts (because adversaries can manage to publish inconsistent messages, so tampering with the execution of smart contracts).

Contributions. We propose a protocol that allows meta-nodes to maintain a consistent subchain over the Bitcoin blockchain. Our protocol is based on *Proof-of-Stake* [6,18], since extending the subchain must be endorsed with a money deposit. Intuitively, a meta-node which publishes a consistent message gets back its deposit once the message is confirmed by the rest of the network. In particular, our protocol provides an economic incentive to honest meta-nodes, while disincentivizing the dishonest ones. We empirically validate the security of our protocol by simulating it in various attack scenarios. Notably, our protocol can be implemented in Bitcoin by only using the so-called *standard* transactions[1].

[1] This is important, because non-standard transactions are discarded by peers running the official Bitcoin client. A restricted group of miners accept non-standard transactions (e.g., the *Eligius* community), but their mining power is quite limited.

2 Bitcoin and the blockchain

Bitcoin is a cryptocurrency and a digital open-source payment infrastructure that has recently reached a market capitalization of almost \$16 billions[2]. The Bitcoin network is peer-to-peer, not controlled by any central authority [20]. Each Bitcoin user owns one or more personal wallets, which consist of pairs of asymmetric cryptographic keys: the public key uniquely identifies the user *address*, while the private key is used to authorize payments. *Transactions* describe transfers of bitcoins (Ƀ), and the history of all transactions, which recorded on a public, immutable and decentralised data structure called *blockchain*, determines how many bitcoins are contained in each address.

To explain how Bitcoin works, we consider two transactions T_0 and T_1, which we graphically represent as follows:[3]

T_0
in: $\cdots$
in-script: $\cdots$
out-script(T, σ): $ver_k(\mathsf{T}, \sigma)$
value: v_0

T_1
in: T_0
in-script: $sig_k(\bullet)$
out-script($\cdots$): $\cdots$
value: v_1

The transaction T_0 contains v_0Ƀ, which can be *redeemed* by putting on the blockchain a transaction (e.g., T_1), whose in field is the cryptographic hash of the whole T_0 (for simplicity, just displayed as T_0 in the figure). To redeem T_0, the in-script of T_1 must contain values making the out-script of T_0 (a boolean programmable function) evaluate to true. When this happens, the value of T_0 is transferred to the new transaction T_1, and T_0 is no longer redeemable. Similarly, a new transaction can then redeem T_1 by satisfying its out-script.

In the example displayed above, the out-script of T_0 evaluates to true when receiving a digital signature σ on the redeeming transaction T, with a given key pair k. We denote with $ver_k(\mathsf{T}, \sigma)$ the signature verification, and with $sig_k(\bullet)$ the signature of the enclosing transaction (T_1 in our example), including *all* the parts of the transaction *except* its in-script.

Now, assume that the blockchain contains T_0, not yet redeemed, when someone tries to append T_1. To validate this operation, the nodes of the Bitcoin network check that $v_1 \leq v_0$, and then they evaluate the out-script of T_0, by instantiating its formal parameters T and σ, to T_1 and to the signature $sig_k(\bullet)$, respectively. The function ver_k verifies that the signature is correct: therefore, the out-script succeeds, and T_1 redeems T_0.

Bitcoin transactions may be more general than the ones illustrated by the previous example: their general form is displayed in Figure 1. First, there can be multiple inputs and outputs (denoted with array notation in the figure). Each output has an associated out-script and value, and can be redeemed independently from others. Consequently, in fields must specify which output they are

[2] Source: crypto-currency market capitalizations http://coinmarketcap.com

[3] in-script and out-script are respectively referred as scriptPubKey and scriptSig in the Bitcoin documentation.

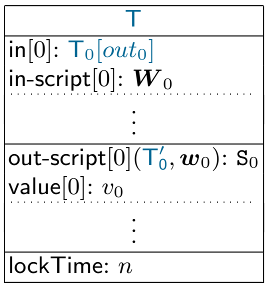

T
in[0]: $\mathsf{T}_0[out_0]$
in-script[0]: $\boldsymbol{W}_0$
⋮
out-script[0]($\mathsf{T}'_0, \boldsymbol{w}_0$): S_0
value[0]: v_0
⋮
lockTime: n

Fig. 1: General form of transactions.

redeeming ($\mathsf{T}_0[out_0]$ in the figure). Similarly, a transaction with multiple inputs associates an in-script to each of them. To be valid, the sum of the values of all the inputs must be greater or equal to the sum of the values of all outputs. In its general form, the out-script is a program in a (not Turing-complete) scripting language, featuring a limited set of logic, arithmetic, and cryptographic operators. Finally, the lockTime field specifies the earliest moment in time (block number or Unix timestamp) when the transaction can appear on the blockchain.

The Bitcoin network is populated by a large set nodes, called *miners*, which collect transactions from clients, and are in charge of appending the valid ones to the blockchain. To this purpose, each miner keeps a local copy of the blockchain, and a set of unconfirmed transactions received by clients, which it groups into *blocks*. The goal of miners is to add these blocks to the blockchain, in order to get a revenue. Appending a new block B_i to the blockchain requires miners to solve a cryptographic puzzle, which involves the hash $h(B_{i-1})$ of block B_{i-1}, a sequence of unconfirmed transactions $\langle T_i \rangle_i$, and some salt R. More precisely, miners have to find a value of R such $h(h(B_{i-1}) \| \langle T_i \rangle_i \| R) < \mu$, where the value μ is adjusted dynamically, depending on the current hashing power of the network, to ensure that the average mining rate is of 1 block every 10 minutes. The goal of miners is to win the "lottery" for publishing the next block, i.e. to solve the cryptopuzzle before the others; when this happens, the miner receives a reward in newly generated bitcoins, and a small fee for each transaction included in the mined block. If a miner claims the solution of the current cryptopuzzle, the others discard their attempts, update their local copies of the blockchain with the new block B_i, and start mining a new block on top of B_i. In addition, miners are asked to verify the validity of the transactions in B_i by executing the associated scripts. Although verifying transactions is not mandatory, miners are incentivized to do that, because if in any moment a transaction is found invalid, they lose the fee earned when the transaction was published in the blockchain.

If two or more miners solve a cryptopuzzle simultaneously, they create a *fork* in the blockchain (i.e., two or more parallel valid branches). In the presence of a fork, miners must choose a branch wherein carrying out the mining process; roughly, this divergence is resolved once one of the branches becomes longer than the others. When this happens, the other branches are discarded, and all the orphan transactions contained therein are nullified.

Overall, this protocol essentially implements a *"Proof-of-Work"* system [12].

3 A protocol for consensus on Bitcoin subchains

We define the notions of subchain and consistency in Section 3.1. In Section 3.2 we describe our protocol to embed consistent subchains on the Bitcoin blockchain, and we examine some of its properties in Section 3.3. Finally, in Section 3.4 we show how to actually implement our protocol in Bitcoin.

3.1 Subchains and consistency

We assume a set $\mathsf{A}, \mathsf{B}, \ldots$ of clients, who want to append messages $\mathtt{a}, \mathtt{b}, \ldots$ to the subchain. We denote with $\mathtt{pay}(v, \mathsf{B})$ a special message which represents a payment of vB to participant B. When this message is on the subchain, it also acts as a standard transaction on the Bitcoin blockchain, which makes vB in a transaction of A redeemable by B.

A *label* is a pair containing a participant A and a message $\mathtt{a}$, written $\mathsf{A} : \mathtt{a}$. A *subchain* is a finite sequence of labels, written $\mathsf{A}_1 : \mathtt{a}_1 \cdots \mathsf{A}_n : \mathtt{a}_n$, that are embedded in the Bitcoin blockchain. The intuition is that A_1 has embedded the message $\mathtt{a}_1$ in some transaction T_1 of the Bitcoin blockchain, then A_2 has appended some transaction T_2 embedding $\mathtt{a}_2$, and so on. Now, not all possible sequences of labels form valid subchains: to define the *consistent* ones, we interpret subchains as traces of a *Labelled Transition System* (LTS).

Formally, an LTS is a tuple $(Q, L, q_0, \rightarrow)$, where:

- Q is a set of states, ranged over by $q, q', \ldots$;
- L is a set of labels, which in our case have the form $\mathsf{A} : \mathtt{a}$;
- $q_0 \in Q$ is the initial state;
- $\rightarrow \subseteq Q \times L \times Q$ is a transition relation.

We require the transition relation $\rightarrow$ to be deterministic, i.e. if $q \xrightarrow{\mathsf{A}:\mathtt{a}} q'$ and $q \xrightarrow{\mathsf{A}:\mathtt{a}} q''$, then it must be $q' = q''$.

The intuition is that the subchain has a state (initially, q_0), and each message sent by participants updates the state of the subchain according to the transition relation. More precisely, if the subchain is in state q, then a message $\mathtt{a}$ sent by A makes the state evolve to q' whenever $q \xrightarrow{\mathsf{A}:\mathtt{a}} q'$ is a transition in the LTS.

Note that, since we are assuming the transition relation $\rightarrow$ to be deterministic, branches cannot happen. However, for some state q and label $\mathsf{A} : \mathtt{a}$, it may happen that there does *not* exist any state q' such that $q \xrightarrow{\mathsf{A}:\mathtt{a}} q'$. In this case, if q is the current state of the subchain, we want to make hard for a participant (possibly, an adversary trying to tamper with the subchain) to append such message. Informally, a subchain is *consistent* if it satisfies such condition, which we formalise as follows.

Definition 1 (Subchain consistency). *We say that a subchain* $\mathsf{A}_1 : \mathsf{a}_1 \cdots \mathsf{A}_n : \mathsf{a}_n$ *is* consistent *whenever there exist* $q_1, \ldots, q_n$ *such that:*

$$q_0 \xrightarrow{\mathsf{A}_1:\mathsf{a}_1} q_1 \xrightarrow{\mathsf{A}_2:\mathsf{a}_2} \cdots \xrightarrow{\mathsf{A}_n:\mathsf{a}_n} q_n$$

Note that, if a subchain is consistent, then by determinism we have that the state q_n exists and is unique. In other words, a consistent sequence of messages uniquely identifies the state of the subchain.

Similarly to Bitcoin, we do not aim at guaranteeing that a subchain is *always* consistent. Indeed, also in Bitcoin a miner could manage to append a block with invalid transactions: in this cases, as discussed in Section 2, the blockchain forks, and the other miners must choose which branch to follow. However, honest miners will neglect the branch with invalid transactions, so eventually (since honest miners detain the majority of computational power), that branch will be abandoned by all miners.

For subchain consistency we adopt a similar strategy: we assume that an attacker can append a label $\mathsf{A} : \mathsf{a}$ such that $q_n \not\xrightarrow{\mathsf{A}:\mathsf{a}}$, so making the subchain inconsistent. However, upon receiving such label, honest nodes will simply discard it. To formalise their behaviour, we define below a function Γ that, given a subchain η (not necessarily consistent), filters all the invalid updates. Hence, $\Gamma(\eta)$ is a consistent subchain.

Definition 2 (Branch pruning). *We inductively define the endofunction* Γ *on subchains as follows, where* ϵ *denotes the empty subchain:*

$$\Gamma(\epsilon) = \epsilon \qquad\qquad \Gamma(\eta\ \mathsf{A} : \mathsf{a}) = \begin{cases} \Gamma(\eta)\ \mathsf{A} : \mathsf{a} & \textit{if } \exists q, q' : q_0 \xrightarrow{\Gamma(\eta)} q \xrightarrow{\mathsf{A}:\mathsf{a}} q' \\ \Gamma(\eta) & \textit{otherwise} \end{cases}$$

In order to model which labels can be appended to subchain without breaking its consistency, we introduce below the auxiliary relation $\models$. Informally, given a consistent subchain η, the relation $\eta \models \mathsf{A} : \mathsf{a}$ holds whenever the subchain $\eta\ \mathsf{A} : \mathsf{a}$, obtained by appending $\mathsf{A} : \mathsf{a}$ to η, is still consistent.

Definition 3 (Consistent update). *We say that* $\mathsf{A} : \mathsf{a}$ *is a* consistent update *of a subchain* η*, denoted with* $\eta \models \mathsf{A} : \mathsf{a}$*, iff the subchain* $\Gamma(\eta)\ \mathsf{A} : \mathsf{a}$ *is consistent.*

3.2 Description of the protocol

Assume a network of mutually distrusted nodes $\mathsf{N}, \mathsf{N}', \ldots$, that we call *meta-nodes* to distinguish them from the nodes of the Bitcoin network. Meta-nodes receive messages from clients (also mutually distrusting) that want to extend the subchain. Our goal is to allow honest participants (i.e., those who follow the protocol) to perform consistent updates of the subchain, while disincentivizing adversaries who attempt to make the subchain inconsistent.

To this purpose, we propose a protocol based on *Proof-of-Stake* (PoS). Namely, we rely on the assumption that the overall stake retained by honest participants

1. Upon receiving an update request $\mathsf{UR}[\mathsf{A} : \mathsf{a}]$, a meta-node checks if it is consistent. If so, it votes the request, and adds it to the request pool;
2. when Δ expires, the arbiter signs all the well-formed UR in the request pool;
3. all requests signed by the arbiter are sent to the Bitcoin miners, to be published on the blockchain. The first to be mined, indicated with UR_i, is the i-th message of the subchain.

Fig. 2: Summary of a protocol stage i.

is greater than the stake of dishonest ones[4]. The stake is needed by meta-nodes, which have to vote for approving update requests. These requests are embedded into Bitcoin transactions, which we denote by $\mathsf{UR}[\mathsf{A} : \mathsf{a}]$, meaning that A wants to append the message a to the subchain. In order to vote an update request, a meta-node must invests κ ₿ on it. A request needs the vote of a single meta-node. The protocol requires meta-nodes to vote consistent updates only: namely, in the subchain η, a request $\mathsf{UR}[\mathsf{A} : \mathsf{a}]$ is voted only if $\eta \models \mathsf{A} : \mathsf{a}$[5]. To incentivize nodes to vote their requests, clients pay meta-nodes a *fee* (smaller than κ), that can be redeemed when the update request is accepted by the network.

We define our protocol in Figure 2. It is organised in *stages* of duration Δ. The actual value of Δ is discussed in Section 5. In each stage i, exactly one transaction UR_i is appended to the subchain (and consequently, to the Bitcoin blockchain). To cope with this issue, the protocol introduces the figure of the *arbiter* T, a node of the network which is considered to be honest (we discuss this hypothesis in Section 3.3). Through the arbiter, it is possible to implement in Bitcoin the mechanism of choosing exactly one transaction to append to the subchain, per stage, as well as ensuring this choice is random. We now describe the main steps of the protocol.

At step 1 of the stage i of the protocol, a meta-node votes a request transaction. Besides paying κ ₿ (as detailed in Section 3.4), it also N has to confirm a previous update in the subchain. Namely, N has to pay κ ₿ plus the client fee to the meta-node N' who appended to the subchain UR_j. The protocol limits the choice of j to the C most recent UR_j, with $C \geq 2$. The value C is called *checkpoint offset*, and let the protocol avoid the *self-compensation attack* shown in Section 3.3. Therefore, j must be chosen as the maximum value such that: (i) $j < i$; (ii) $|i - j| < C$; (iii) $\mathsf{UR}_\mathsf{j}[\mathsf{A} : \mathsf{a}]$ is consistent. This implements an *incentive* to vote consistent updates only, since inconsistent ones are not confirmed. If none of the last C updates in the subchain are consistent, then N chooses the

[4] Note that a similar hypothesis, but related to computational power rather than stake, holds in Bitcoin: there, honest nodes are supposed to control more computational power than dishonest ones.

[5] We are assuming that all meta-nodes agree on the Bitcoin blockchain; since η is a projection of the blockchain, they also agree on η.

last update. Then, N adds $\mathsf{UR}[\mathsf{A} : \mathsf{a}]$ to the *request pool*, i.e. the set of all voted requests of the stage (which is emptied at the beginning of each stage).

At step 2, that is enabled when Δ expires, the arbiter T signs all well-formed request transaction, (i.e., those respecting the format in Section 3.4).

At step 3, the meta-nodes send the request signed by the arbiter to the Bitcoin network. The mechanism described in Section 3.4 ensures that only one signed transaction in the request pool of stage i will appear on the blockchain. This transaction is denoted as $\mathsf{UR}_\mathsf{i}[\mathsf{A} : \mathsf{a}]$ as is considered appended on the subchain.

3.3 Basic properties of the protocol

We now establish some basic properties of our protocol. In particular, we show that the protocol imposes an upper-bound to the capabilities of the adversary. To do this, we assume the network to be saturated (i.e., there are enough requests to vote), and the honest nodes to control the majority of the total stake of the network[6], hereafter denoted by S.

Adversary power. An honest meta-node votes any request for which it has enough stake, hence if its stake is s it votes $\beta = s/\kappa$ requests per stage. Consequently, the rest of the network — which may include dishonest meta-nodes not following the protocol — can vote at most $(S - s)/\kappa$ requests. So, we have:

Proposition 1. *In a given protocol stage, the probability that an honest meta-node with stake s updates the subchain is at least s/S.*

Since we assume that honest meta-nodes control the majority of the stake, Proposition 1 also limits the capabilities of the adversary:

Proposition 2. *In a given protocol stage, if the honest meta-nodes have stake s, then the probability that a dishonest meta-node updates the subchain is at most $(S - s)/S$.*

Although inconsistent updates are ignored by honest meta-nodes, their side effects as plain Bitcoin transactions cannot be revoked once they are included in the Bitcoin blockchain. This is the case, e.g., of transfers of bitcoins due to $\mathtt{pay}(v, \mathsf{B})$ messages. We show in the following how the incentive system in our protocol reduces the feasibility of such inconsistent updates.

Assume that an adversary M manages to append 2 updates to the subchain, the first with index j, and the second with index $i > j$. Suppose that the update at index j is inconsistent, while the one at index i is consistent. Since M does not follow the protocol, she can exploit UR_i to refund its κ put on UR_j. Then, since UR_i is consistent, the adversary will be refunded for the second κ by another honest meta-node. We call the above *self-compensation attack*.

Now, according to Proposition 2, if M has stake m (and the other meta-nodes are honest), then she has probability at most m/S to extend the subchain in a

[6] Under this assumption, meta-nodes can ensure that the arbiter is honest.

given stage of the protocol. Since stages can be seen as independent events, and since M has to publish at least 2 updates over the most recent checkpoint to perform the attack, we obtain the following:

Proposition 3. *The probability that an adversary with stake m succeeds in a self-compensation attack is at most:*

$$\binom{C}{2} \cdot \mu^2 (1-\mu)^{C-2}$$

where C is the checkpoint offset, and $\mu = m/S$.

Since the probability to publish inconsistent updates without losing κ grows with C, it is crucial to keep this value small. For instance, if $\mu = 0.1$ an adversary could perform the attack with probability bounded by 0.01 if $C = 2$, 0.027 if $C = 3$, 0.0486 if $C = 4$.

Observe that if the attack succeeds once then the attack probability slightly increases, since the stake m is charged by the client fees of the published updates. This is not an issue if the fee is small compared to S.

Trustworthiness of the arbiter. In order to simplify the description of the protocol, we have assumed the arbiter T to behave honestly. However, our arbiter does not play the role of a trusted authority: indeed, the update requests to be voted are chosen by the meta-nodes and, once they are added to the request pool, the arbiter is expected to sign all of them, without taking part on the validation nor the voting. Since everyone can inspect the request pool, any misbehaviour of the arbiter can be detected by the meta-nodes, which can proceed to replace it.

3.4 Implementation in Bitcoin

In this section we show how our protocol can be implemented in Bitcoin. A transaction UR[A : a] at position i of the subchain is implemented as the Bitcoin transaction UR_i in Figure 3a, where the out-script[0] is an unspendable script of the form `OP_RETURN`, with attached as metadata (a suitable encoding of) A : a.

Each transaction in the chain redeems the output out[1] of the previous transaction in the subchain (pointed by in[2]) by providing the arbiter signature. In this way, all UR in a protocol stage redeem the same output, making them mutually exclusive; this ensures that only one update per stage will be mined.

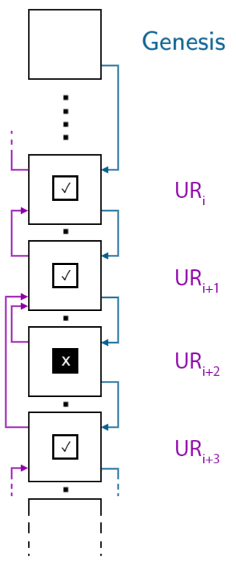
The incentive mechanism is implemented through out-script[2]. The script rewards the meta-node N′ that voted a preceding UR_j in the subchain. Therefore, this output can be redeemed only by providing the signature of N′. This output pays back to N′ a total amount of κ ₿ plus the client fee. The output out[3] is only used for messages on the form pay(v, B), i.e. those which pay v to a participant B (and whose input can be additionally provided by Fee_i).

All transactions must specify a `lockTime` $n+1$ (where n is the current Bitcoin block number), so that they can be mined only after the n-th block is mined.

Genesis
in: $\cdots$
in-script: $\cdots$
out-script(T, σ): $ver_{\mathtt{T}}$(T, σ)
value: 0.0001

$\mathsf{UR_i}$
in[0]: $\mathsf{Fee_i}[out]$
in-script[0]: $sig_{\mathtt{C}}(\bullet)$
in[1]: $\mathsf{Stake_i}[out]$
in-script[0]: $sig_{\mathtt{N}}(\bullet)$
in[2]: $\mathsf{Confirm_{i-1}}[out_1]$
in-script[0]: $sig_{\mathtt{T}}(\bullet)$
out-script[0](): `OP_RETURN` A : a
value[0]: 0
out-script[1](T, σ): $ver_{\mathtt{T}}$(T, σ)
value[1]: 0.0001
out-script[2](T, σ): $ver_{\mathtt{N'}}$(T, σ)
value[2]: κ + fee
out-script[3](T, σ): $ver_{\mathtt{B}}$(T, σ)
value[3]: v_{pay}
lockTime: $n + 1$

(a)



(b)

Fig. 3: In (a), format of Bitcoin transactions used to implement our protocol. In (b), a subchain mantained through our protocol. Since $\mathsf{UR_{i+2}}$ contains an inconsistent update, the meta-node that voted it is not rewarded.

The locktime avoids that a transaction signed by the arbiter before the others is sent to Bitcoin miners beforehand, thus having a higher probability to be mined.

To initialize the subchain, the arbiter puts the Genesis transaction on the Bitcoin blockchain. This transaction secures a small fraction of bitcoin, which can be redeemed only with the arbiter signature. Its out is initially redeemed by $\mathsf{UR_1}$, then transferred to each subsequent subchain update (see Figure 3b). At each protocol stage, clients send incomplete UR transactions to the network. These transactions contain only in[0] and out[0], specifying the client fee (plus v in the case of pay(v, B)) and the message for the subchain. To vote, meta-nodes add in[1], in[2] and out[2] to these transactions, to, respectively, put the required κ (from some $\mathsf{Stake_i}$), declare they want extend the last published update $\mathsf{Confirm_{i-1}}$, and specify the previous update to be rewarded. All the in[2] of transactions in a stage of the protocol must have a different value, to avoid that an attacker can vote more URs with the same funds. After that, meta-nodes ask clients to sign them. Before signing, clients perform some basic checks, e.g. that fields have been initialized correctly, (so to avoid attempts to steal the fee). Once voted by meta-nodes and signed by clients, $\mathsf{UR_j}$ are broadcast to the request pool. Finally, the arbiter signs all transactions respecting the protocol, so that they can be mined.

4 Evaluation of the protocol

In this section we evaluate the security of our protocol, providing some experimental results. We also investigate how possible attacks to Bitcoin may affect subchains built on top of its blockchain.

Attack scenario. We assume an adversary who can craft any update (consistent or not), and controls one meta-node M with stake μS, where $\mu \in [0;1]$ and S is the total stake of the network[7]. We suppose that each meta-node can vote as many update requests as possible, spending all its stake, and that the network is always saturated with pending updates, which globally amount to the entire stake of honest meta-nodes[8]. We also assume that M gets an additional extra revenue r for each inconsistent update, modelling the case where she manages to induce a victim to publish an inconsistent payment $\texttt{pay}(r, \mathsf{M})$. The goal of M is to append at least 2 updates to the blockchain (one of which inconsistent) every C published updates. She can use any possible strategy to achieve this goal.

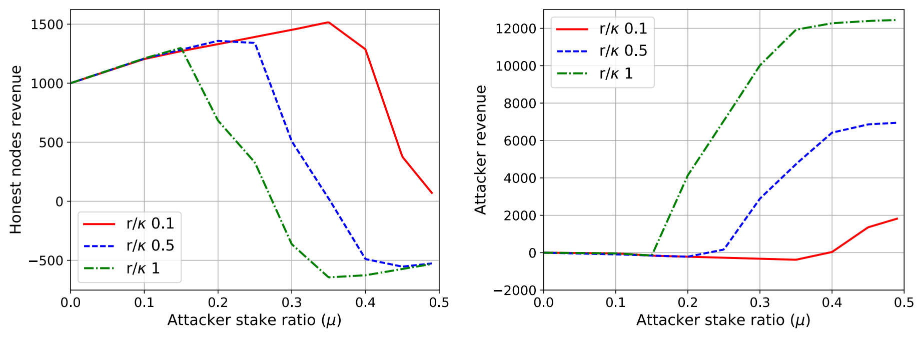
We simulate the protocol under the attack scenario described above. Each simulation runs the protocol to generate a subchain with $10,000$ messages, setting the client fee to 0.1κ and the checkpoint offset to 3. To this purpose we use DESMO-J [15], a discrete event simulator for Java.

Experimental results. Figure 4b measures the attacker revenue as μ increases. In particular, it shows that if the stake threshold κ is ten times greater than r, M gains only if she owns at least $\sim$40% of the global stake (i.e., $\mu \geq 0.4$). Therefore, under such assumption about the attacker stake, the security of our protocol is comparable with that of the Bitcoin *Proof-of-Work* protocol [14]. Instead, if $\kappa = r$, the attacker needs only $\sim$15% of the global stake to profit from the attack. Figure 4a shows that, in the absence of attackers ($\mu = 0$), the revenue of honest nodes is essentially the client fee times the number of updates published, as expected. Further, μ is below the threshold required to perform a profitable attack, the revenue of honest nodes increases: this happens because inconsistent updates voted by M reward honest ones, whereas the opposite cannot occur. Summing up, our protocol is secure only if, for updates on the form $\texttt{pay}(v, \mathsf{A})$, we have that $v \leq \kappa$. Hence, if v is close to 0, the behaving dishonestly is not economically advantageous.

Security of the underlying Bitcoin blockchain. So far we have only considered direct attacks to our protocol, assuming the underlying Bitcoin blockchain to be secure. However, although Bitcoin has been secure in practice till now, some works

[7] Assuming a single adversary is not less general than having many non-colluding meta-nodes which carry on individual attacks. Indeed, in this setting meta-nodes do not join their funds to increase the stake ratio μ.

[8] Note that saying the update queue is not always saturated is equivalent to model an adversary with a stronger μ: this because honest meta-nodes cannot spend all their stake in a single protocol stage, i.e. reducing their actual *power*. Thus, studying this particular case will not give any additional contribution to the analysis.



(a) (b)

Fig. 4: Revenue of honest nodes (a) and of the attacker M (b) for increasing values of the attacker stake ratio μ. The curves represent different values of r/κ (the ratio between the attack revenue r, given by inconsistent pay(r, M) updates, and the cost of the vote).

have spotted some potential vulnerabilities of its protocol. These vulnerabilities could be exploited to execute *Sybil attacks* [4] and *selfish-mining attacks* [13], which might also affect subchains built on top of the Bitcoin blockchain.

In Sybil attacks on Bitcoin, honest nodes are induced to believe that the network is populated by many distinct participants, which instead are controlled by a single malicious entity. This attack is usually exploited to quickly propagate malicious information on the network, and to disguise honest participants in a consensus/reputation protocol, e.g. by overwhelming the network with votes of the adversary. In the selfish-mining attack [13], small groups of colluding miners manage to obtain a revenue larger than the one of honest miners. More specifically, when a selfish-mining pool finds a new block, it keeps it hidden to the rest of the network. In this way, selfish miners gain an advantage over honest ones in mining the next block. This is equivalent to keep a private fork of the blockchain, which is only known to the selfish-mining pool. Note that honest miners still mine on the public branch of the blockchain, and their hash rate is greater than selfish miners' one. Since, in the presence of a fork, the Bitcoin protocol requires to keep mining on the longest chain, selfish miners reveal their private fork to the network just before being overcome by the honest miners. Eyal and Sirer in [13] show that, under certain realistic assumptions, this strategy gives better revenues than honest mining: in the worst scenario (for the adversary), the attack succeeds if the selfish-mining pool controls at least 1/3 of the total hashing power. Rational miners are thus incentivized to join the selfish-mining pool. Once the pool manages to control the majority of the hashing power, the system loses its decentralized nature. Garay, Kiayias and Leonardos in [14] essentially confirm these results: considering a core Bitcoin protocol, they prove that if the hashing power γ of honest miners exceeds the hashing power β of the adversary pool by a factor λ, then the ratio of adversary blocks in the blockchain is bounded by $1/\lambda$ (which is strictly greater than β). Thus, as β (the adversary pool size) approaches $1/2$, they control the blockchain.

Although these attacks are mainly related to Bitcoin revenues, they can affect the consistency of any subchain built on top of its blockchain. In particular, suitably adapted versions of these attacks allow adversaries to cheat meta-nodes about the current subchain state, forcing them to synchronize their local copy of the Bitcoin blockchain with invalid forks that will be discarded by the network in the future. To protect against such attacks, meta-nodes should consider only *l-confirmed* transactions. Namely, if the last published blockchain block is B_n, they consider only those transactions appearing in blocks B_j with $j \leq n - l$. This means that an attacker would have to mine at least l blocks to force the revocation of a *l-confirmed* transaction. Rosenfeld [21] shows that, if an attacker controls at most the 10% of the network hashing power, $l = 6$ is sufficient for reducing the risk of revoking a transaction to less than 0.1%.

5 Discussion

We have proposed a protocol to securely form consensus on subchains, i.e. chains of platform-dependent messages stored on top of the Bitcoin blockchain. Our protocol incentivizes nodes to validate the messages before appending them to the subchain, making economically disadvantageous for an adversary to append inconsistent updates. In order to confirm this intuition we have performed some simulations, which have measured the security of our protocol over different attack scenarios, showing that, under conservative assumptions, its security is comparable to that of Bitcoin (Section 4).

Performance of the protocol. As explained in Section 3.2, the protocol runs in periods of duration Δ. Due to the mechanism for choosing the message to append to the subchain from the request pool, the protocol can publish at most one transaction per Bitcoin block. This means that a lower bound for Δ is the Bitcoin block interval (~10mins). Note also that since we allow meta-nodes to monitor the arbiter behaviour (e.g. to detect if it is trying to marginalize some nodes), and since this requires the request pool to be shared and consistent between all the nodes of the network, meta-nodes may want to verify that the arbiter signs all voted requests in each protocol stage. Then, Δ needs to be large enough to let each node synchronize the request pool with the rest of the network. A possible approach to cope with this issue is to make meta-nodes broadcast their voted updates, and to keep a list of other ones (considering only those which satisfy the format of transactions, as in Section 3.4). More efficient approaches could exploit distributed shared memories [10,17].

Overcoming the metadata size limit. As noted in Section 3.4, we exploit `OP_RETURN` unspendable scripts to embed metadata in Bitcoin transactions. Since Bitcoin limits the size of such metadata to 80 bytes, this might not be enough to store the data needed by platforms. To overcome this issue, one can use distributed hash tables [19] maintained by meta-nodes. In this way, instead of storing full message data in the blockchain, `OP_RETURN` scripts will contain only the corresponding message digests. The unique identifier of the Bitcoin transaction can be used as the key to retrieve the full message data from the hash table.

Smart contracts over subchains. The model of subchains defined in Section 3.1, based on LTSs, can be easily extended to model the computations of smart contracts over the Bitcoin blockchains. A platform for smart contracts could exploit our model to represent the state of a contract as the state of the subchain, and model its possible state updates through the transition relation.

Implementing a platform for smart contracts would require a language for expressing them. To bridge this language with our abstract model, one can provide the language with an operational semantics, giving rise to an LTS describing the computations. Note that our assumption to model computations as a single LTS does not reduce the generality of the system, since a set of LTSs, each one modelling a contract, can be encoded in one LTS as their parallel composition. If the language is Turing-complete, an additional problem we would have to face is the potential non-termination. This issue has been dealt with in different ways by different platforms. E.g., the approach followed by Ethereum [9] is to impose a fee for each instruction executed by its virtual machine. If the fee does not cover the cost of the whole computation, the execution terminates.

A usable platform must also allow to create new contracts at run-time. Since in our model the LTS representing possible computations is fixed, we would need a mechanism to "extend" it. To handle the publication of new contracts, we could modify the protocol so that UR may contain its code, and the unique identifier of the transaction also identifies the contract. In this extended model, update request would also contain the identifier of the contract to be updated, so that meta-nodes can execute the corresponding code.

Acknowledgments. This work is partially supported by Aut. Reg. of Sardinia grant P.I.A. 2013 "NOMAD". Alessandro Sebastian Podda gratefully acknowledges Sardinia Regional Government for the financial support of her PhD scholarship (P.O.R. Sardegna F.S.E. Operational Programme of the Autonomous Region of Sardinia, European Social Fund 2007-2013 - Axis IV Human Resources, Objective l.3, Line of Activity l.3.1).

References

1. Making sense of blockchain smart contracts. `http://www.coindesk.com/making-sense-smart-contracts/`. Last accessed 2017/01/14.
2. opreturn.org. `http://opreturn.org/`. Last accessed 2016/12/15.
3. M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on Bitcoin. In *IEEE S & P*, pages 443–458, 2014.
4. M. Babaioff, S. Dobzinski, S. Oren, and A. Zohar. On Bitcoin and red balloons. In *ACM Conference on Electronic Commerce (EC)*, pages 56–73, 2012.

5. M. Bartoletti and L. Pompianu. An analysis of bitcoin OP_RETURN metadata. In *Financial Cryptography Workshops*, 2017. Also available as CoRR abs/1702.01024.
6. I. Bentov, A. Gabizon, and A. Mizrahi. Cryptocurrencies without proof of work. In *Financial Cryptography Workshops*, volume 9604 of *LNCS*, pages 142–157. Springer, 2016.
7. I. Bentov and R. Kumaresan. How to use Bitcoin to design fair protocols. In *CRYPTO*, volume 8617 of *LNCS*, pages 421–439. Springer, 2014.
8. Blockstore: Key-value store for name registration and data storage on the Bitcoin blockchain. https://github.com/blockstack/blockstore, 2014.
9. V. Buterin. Ethereum: a next generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2013.
10. M. Cai, A. Chervenak, and M. Frank. A peer-to-peer replica location service based on a distributed hash table. In *ACM/IEEE Conference on High Performance Networking and Computing*, page 56. IEEE Computer Society, 2004.
11. R. Dermody, A. Krellenstein, O. Slama, and E. Wagner. CounterParty: Protocol specification. http://counterparty.io/docs/protocol_specification/, 2014.
12. C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, volume 740 of *LNCS*, pages 139–147. Springer, 1993.
13. I. Eyal and E. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*, volume 8437 of *LNCS*, pages 436–454. Springer, 2014.
14. J. A. Garay, A. Kiayias, and N. Leonardos. The Bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, volume 9057 of *LNCS*, pages 281–310. Springer, 2015.
15. J. Göbel, P. Joschko, A. Koors, and B. Page. The discrete event simulation framework DESMO-J: review, comparison to other frameworks and latest development. In *European Conference on Modelling and Simulation (ECMS)*, pages 100–109. European Council for Modeling and Simulation, 2013.
16. A. Hern. A history of Bitcoin hacks. http://www.theguardian.com/technology/2014/mar/18/history-of-bitcoin-hacks-alternative-currency, march 2014.
17. S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *PODC*, pages 213–222. ACM, 2002.
18. A. Kiayias, I. Konstantinou, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure Proof-of-Stake blockchain protocol. *IACR Cryptology ePrint Archive*, 2016:889, 2016.
19. P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *LNCS*, pages 53–65. Springer, 2002.
20. S. Nakamoto. Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.
21. M. Rosenfeld. Analysis of hashrate-based double spending. *CoRR*, abs/1402.2009, 2014.
22. N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

On the feasibility of decentralized derivatives markets

Shayan Eskandari[1] Jeremy Clark[2] Vignesh Sundaresan[1] Moe Adham[1]

1 Bitaccess
2 Concordia University

Abstract. In this paper, we present Velocity, a decentralized market deployed on Ethereum for trading a custom type of derivative option. To enable the smart contract to work, we also implement a price fetching tool called PriceGeth. We present this as a case study, noting challenges in development of the system that might be of independent interest to whose working on smart contract implementations. We also apply recent academic results on the security of the Solidity smart contract language in validating our code's security. Finally, we discuss more generally the use of smart contracts in modelling financial derivatives.

1 Introductory Remarks

The introduction of Bitcoin [13] in 2009 led to a new frontier in decentralizing technologies, both in finance and elsewhere. Of the many implementations, we note a few: file systems like The InterPlanetary File System (IPFS) [2], dynamic name servers like DNSChain [14] and MaidSafe, a fully distributed platform [10]. For our purposes, the most interesting technology is Ethereum [4][17] — a decentralized general transaction ledger. Ethereum in simple words is a decentralized computer that can run code, called smart contracts, which enforce the performance of an agreed upon set of negotiated standards in an automated and immutable way. Smart contracts can be designed to disintermediate traditional trusted parties, replacing them with pre-defined logical parameters. The smart contract concept is not new and was introduced by Szabo in 1997 [16], however there has not been any real implementation of it until Bitcoin, and then in a much more flexible and verbose fashion: Ethereum.

Under the umbrella of "fintech", "blockchain", and "distributed ledger technology", many legacy entities in the financial world (investment banks, security exchanges, clearinghouses, etc.) have expressed interest (through whitepapers and commercial partnerships and consortiums) in decentralizing financial markets. Derivative markets are often cited as a potential target. From the other end, papers on Ethereum and tutorials on Solidity (a high level programming language for Ethereum) often use derivatives as an example application. So there is a degree of consensus that derivatives running on Ethereum is an interesting application to study, but we are not aware of any public projects to attempt to build a derivative market in a serious way. This paper is a first step in that direction.

1.1 Scope & Contributions.

A simplification of a derivative is as follows: two parties enter an agreement where the first stands to profit if a specified security (*e.g.,* stock) appreciates in value over a specified time-period and the second stands to profit if it falls. Since the profitability of the agreement is derived directly from the price of the security, it is called a derivative instrument. The exact operational details that realize this property differs between types of derivatives. The most common derivative is a put/call option which gives the second party (called the *buyer*) the opportunity (but not obligation) to buy/sell a security at a specified price (*strike price*) at (*American*) or within (*European*) a specified time (*expiration*). The buyer pays the first party (the *seller*) a flat fee (*option price*) when purchasing the option. Derivatives are generally held to hedge risks in price movements or for speculation.

In a decentralized derivative system, a buyer and seller can have fast and automatic clearing and settlement (straight through processing) of the derivative without trusting a third party. However the design of a market must consider the following challenges:

1. **Terms of the Contract.** The terms of derivative must be expressible in the smart contract language. In this paper, we write contracts in Solidity for the Ethereum blockchain which is sufficient for describing the core aspects of the contract. We present a full implementation stack (from the smart contracts to a UI) for buying/selling a special type of derivative instrument. We pay special attention to common security risks in developing Solidity-based contracts.
2. **Counterparty Risk.** In most derivatives, the seller is obliged to buy/sell securities upon request of the buyer subject to the terms of the derivative. A seller might choose to not follow through with her obligations. In a centralized setting, identity, reputation and legal recourse are used to combat this. In a decentralized environment, this problem must be addressed. In this paper (and the reason we position it as a first step), we start with derivatives that are fully collateralized — meaning the full settlement amount under all outcomes is capped and this amount is locked to the contract at initiation time and distributed under the conditions of the contract. This means we do not implement a traditional put/call option but rather a tweaked version we describe below. In future work, we will consider counterparty risk broadly and how mitigating it can be combined with our framework to offer more traditional derivatives.
3. **Price Feed.** In a derivative where settlement is fully automated, either the underlying security (or a token representing it) needs to be on the blockchain already or the blockchain needs to be able to assign a value to the security— or more precisely, be fed the price it should use in evaluating the code of the contract. In practice, an entity feeding prices (or any external information) into a smart contract is called an oracle. Some related work has examined

oracles, and we present our decentralized design in Section subsection 4.2 called PriceGeth, which we have made freely available.[1]

4. **Underlying Financial Model.** The buyer and seller of a derivative, whether implicitly or explicitly, must have some sense of what the probabilistic behaviour of the underlying security must be to determine the terms of the contract. This is the purpose of the infamous Nobel-awarded Black-Scholes model for stock prices — now obsolete but influential for decades. In our system, such a model is not baked into the functioning of the smart contract but would be used externally to decide favourable terms before buying/selling derivatives. For stocks, modern models (like jump-diffusion) might be used. For derivatives on cryptocurrencies or more esoteric securities, models simply do not exist yet and are an open area of research. Finally, we note that the derivative ultimately settles in Ether and so inflations/deflation of the currency might erode an otherwise profitable derivative.

In summary, we limit our contributions to (1) and (3) in this work, but also propose this fuller landscape as a useful research agenda for future researchers.

2 Related Work

Work on trusted oracles and price feeds, in the Ethereum eco-system, include TownCrier [18] which acts as an attested bridge (running within an SGX enclave) between trusted sources of information and the Ethereum blockchain. Oraclizeit[2] is another price feed which uses the similar workflow to fetch the requested information. Our approach differs from these as PriceGeth publishes the data to the Ethereum blockchain from the trusted source of information and the historical data is available to all smart contracts, however in comparison with the other approaches, is limited to only the published data (Price pairs).

Equibit [11] proposes a method to issue, create, disseminate and maintain equity across a broad base of investors without the need of intermediaries for record keeping. It is conceivable that derivative smart-contracts could utilize Equibit equity as payment or settlement method, as opposed to simply using Bitcoin or Ethereum's native digital currencies.

Bentov et al. [3] note than an extension to their work on decentralized prediction markets can be a derivative instruments they call a *capped contracts for difference*. It is similar to the one implemented in Velocity (their paper is not an implementation but a study of game theoretic properties).

Recent attacks on smart-contracts, such as TheDAO attack [9] attracted security researchers to analyze further on this era. Solidity security and survey of the attacks by Atzei et al. [1] lists some of the known security vulnerabilities and Luu et al. developed a tool for static analysis on smart contract codes [12] which we used.

[1] `https://github.com/VelocityMarket/pricegeth`

[2] `http://www.oraclize.it/`

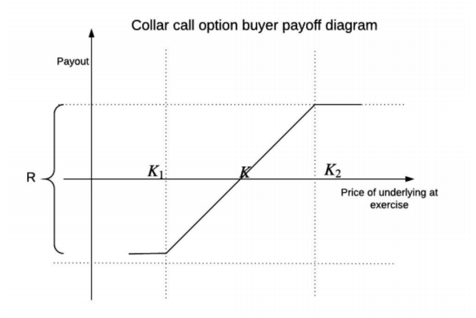


Fig. 1: Our collar-esque option with maximum long payout scenario. K1 is the initial price, K2 is the price at expiry time and R is the pre-defined collar for payouts

3 Materials and Methods

Smart Contracts. A *contract* is a written or spoken agreement between two or more parties that is intended to be enforceable by law. In a *smart contract*, terms are written in code and executed by machines, removing the human performance component (unless if such a component is specified). We can consider our main smart contract as a black box: the inputs are investors' deposited *ether* (Ethereum's cash) and their position on the future price of an asset, either short or long. The smart contract will retain the deposit in escrow and execute a payout calculation and the payout itself when the expiry date comes. The payout is in Ether only, no actual shares are exchanged (a *contract for difference*) and the maximum payout is capped (*limit up/down*). Due to the deposit, there is no counter-party risk however the contract requires a trustworthy price feed and the investors earn zero interest for the duration of the contract. For this reason, we consider this a first step toward more flexible arrangements. The contract disintermediates the trusted role of the exchange (or broker for over-the-counter) and settling/clearing entities.

Types of Options. We implement a non-standard option that is similar to a collar or hedge wrapper. It is non-standard due to our requirement of escrowing money, which we make to side-step counter-party risk and enable a fully autonomous and disintermediated contract. The contract collects funds from the hedgers/speculators who take opposing positions on the future prospects of an asset: one takes the short position when they believe the underlying asset's value will lose value from its current price, and other takes the opposite long position speculating a rise in the price. In its simplest form, the collar options pay out \$1 for every \$1 change in the underlying asset (the payout can be made dependent on a drift term or even made non-linear). The payout is limited by the amount of money held in escrow—if the price rises beyond the limit, it is said to be limit up (or limit down in the opposite case) and the payout will be fixed (see

Figure 1). This kind of payout capping helps the contract holders stay immune to systemic risks and extreme jumps.

Development and Deployment. There are a few blockchains that would let us code an autonomous smart contracts: Ethereum, RSK [8] and more. The decision to work on Ethereum blockchain rather than others solely came from the fact that there are more active developers in the community and maturity of the platform. Even though Ethereum is in early stages, it is more mature than other smart contract compatible platforms. The programming language used for smart contract development is Solidity in most of these platforms. All smart contracts developed and used in this paper has been deployed and tested by our beta testers on Ethereum testnet. In Ethereum blockchain, transactions and processing power costs some small amount of ether called *gas*[3]. For each transaction, the sender defines the gasLimit and also gasPrice for processing that transaction and miners decide to include those transactions in the blocks they mine or not. The concept of gas has many angles to discuss which falls outside of the scope of this paper. We will discuss some more in section 5.

4 Implementation

We call our platform Velocity. We tried to model the real-life scenario of buying an options derivatives. Consider the case where Alice goes to a broker and buys an options contract from Bob. The broker is the one that handles the money transfer and also execute the options contract at the contract expiry time. Now our goal is to replace the broker with a smart contract. For the purpose of a proof of concept, **the smart contract will also act as Bob**, meaning if Alice buys a short call option, the Velocity smart contract will put a long call against her short call. This can be generalized so that other entities can fund the contract but for the rest of this paper, Velocity acts as a market maker. This might lead to users gaming the system, however it's trivial to change the smart contract to wait for the other opponent to enter the contract. We discuss this more in section 5.

4.1 Velocity Main Smart Contract

A Velocity smart contract can be used for speculation on the price of any two assets[4], although the Ethereum price is always exposed as the deposits and the withdrawals are done in ETH[5]. As for this experiment, we use the price pair of Bitcoin (XBT/BTC) and Ethereum (ETH). If we used price pairs not involving ETH, for example the CAD/USD exchange rate, it would suffice to use two contracts for CAD/ETH and ETH/USD. Or the payout function could be changed to specify how it relates to numbers it is given. Note that in either case,

[3] What is gas? `http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html#what-is-gas`

[4] or any other events that an options contract can be based on

[5] Ethereum symbol

the payout will always be in ETH. In its full generality, any number that changes over time and has a suitable feed (we describe feeds below) can be used: price (stocks, bonds, commodities, etc.), rate (interest, inflation, population, etc.), or something else (average global temperature, number of days without rain, etc).

Smart contract. The way Velocity smart contract is implemented, one party purchases a contract by sending a nominal amount of ethereum (0.1 ETH) to the contract's ethereum address. Once confirmed by the network, the contract will fetch a starting price from the price feed, PriceGeth, and run for a period of time to reach the expiry time. The smart contract would put the same amount of ETH from its pool of funds into escrow for the payout. In the PoC demo, we use 5 ethereum blocks (approximately 1 minute) to settle a contract. When the expiry time reaches, the same party must send another transaction to the contract and call the settlement function to settle the contract which leads to sending the payouts by the smart contract. While this experiment was going under beta testings, we found out that if the user loses the contract, there is no incentive to call the settle function as it would use up some ETH in gas and would not pay the user. This would lead stale money held in the escrow of the smart contract. This made us redesign our settlement functions and write one centralized cron job script to go through the unsettled contracts once a day and call the settle function on the ones that have been expired.

```
 modifier checkMargin(uint amount) {
    if (amount == (applyLOT(Margin)))
    { _ ;} else {
        Error("Invalid Margin!");
        immediateRefund();}
    }
function goLong() public hasEnoughFunds(msg.value) checkMargin(msg.value)
 ↪  payable returns(uint) {
    lastOptionId = newOption(msg.sender, msg.value, true);
    LongOption(lastOptionId, msg.sender, msg.value, block.number);
    return lastOptionId;
    }
```

Code 1: Velocity Main Smart Contract - Long Option Call, The sender of a transaction to goLong() function has to send exactly the Margin value and with that he enters the option contract for Margin value with the smart Contract

Settle function. exercise() is responsible in settling the options contract and pay out both parties (see 2), in which here is the user and the Velocity smart contract. Most of the functions are responsible to find the appropriate option contract and calculate the pay outs. However there are some functions that were added later on for security measurements, such as isOpen modifier. Modifiers in Solidity are functions that can check some statements before executing the main function. The first deployed version of Velocity main contract was vulnerable to a similar (but not the same) attack as the DAO attack, see section 5. It was possible for an attacker to call an option contract and upon settling and winning, keep calling the exercise() function using his OptionId and get more of the same amount of payout over and over again. The code was patched and a new smart contract was deployed later in the experiment[6]. send() is a built-in function in Solidity which handles the sending of funds to other ethereum addresses or contracts. There are known vulnerabilities on how send() function works in solidity which should be appropriately handled. One can use a smart contract address as his option payout address which would execute some code upon receiving any funds and use that code flow to drain the sender's contract. payAndHandle() function tried to use the best security practices to prevent such attacks (see 5 for the source code).

```
modifier isOpen(uint optionId) {if (AllOptions[optionId].closed) throw; _ ;}
function exercise() public {
  exercise(findOptionId(msg.sender));
}
function exercise(uint optionId) public isOpen(optionId) returns(bool) {
  // REMOVED SOME CODE TO SAVE SPACE, FULL SOURCE CODE IS AVAILABLE ON VELOCITY GITHUB REPOSITORY
  AllOptions[optionId].closed = true; //Doing this before payouts to prevent replay attacks on same instance of the contract
  LockedBalance -= AllOptions[optionId].amount; //release locked amount from escrow
  // Payout calculation
  if (pricesToCheck.pricediff >= (int(Margin))) { // diff >= (margin) -> Pay Long
      //pay long
      return payAndHandle(optionId, AllOptions[optionId].Long, 2 * AllOptions[optionId].amount);
  }
  if ((0 < pricesToCheck.pricediff) && (pricesToCheck.pricediff < (int(Margin)))) { // 0 < diff <  margin
```

[6] Fix for the multiple payout bug: https://github.com/VelocityMarket/Options-Contract/commit/f3c8d0ef66b886c9ee8b432e92c83f3a4fb525ba

```
      return (payAndHandle(optionId, AllOptions[optionId].Long,
↪  (AllOptions[optionId].amount + pricesToCheck.priceDiffLOT)) &&
↪  payAndHandle(optionId, AllOptions[optionId].Short,
↪  (AllOptions[optionId].amount - pricesToCheck.priceDiffLOT)));
    }
  }
```

Code 2: Settle function of main options contract

Source Code API documentation for other smart contracts to use the functionality and also Python and NodeJS clients to communicate with the main smart contract are available on Github[7].

4.2 Price feed

A decentralized Price feed is an essential requirement for having a decentralized derivative market. There are a few proposals on how to fetch the price in a smart contract. One is using *Smart Contract* oracles[8], they offer daily updates for the price using a predefined data source. This was not an option to be used for our purpose as a daily update is not sufficient for short term derivative markets. Another option that could be used was Oraclizeit. They way Oraclizeit works is that the client smart contract, Velocity main contract in our case, sends a transaction to Oraclizeit smart contract with the required API url and the fields it needs, sometime after the confirmation by the network, Oraclizeit smart contract sends a callback transaction to Velocity smart contract with the requested data (Figure 2).

For the first implementation of Velocity smart contract we used Oraclizeit method to fetch the price.

As mentioned before, most of the decentralized application infrastructure on Ethereum blockchain are in Beta state and might not work as intended. This applies for Oraclizeit, specially as by design they have a central server which can stop working without any notice or visible signs. The red boxes in Figure 2 indicates the centralized parts of the system. As you can see in 3, Oraclizeit will send the price to the callback function at the time of the call and also execute the exercise() function which is responsible for saving the price and calculating the payout amounts. This makes the callback function one of the important functions which should be called at the specific time.

```
//initiating oraclize it
oraclize_setProof(proofType_TLSNotary | proofStorage_IPFS);
```

[7] Simple collared option smart contract: https://github.com/VelocityMarket/Options-Contract
[8] Data and Payments for your Smart Contracts https://smartcontract.com/

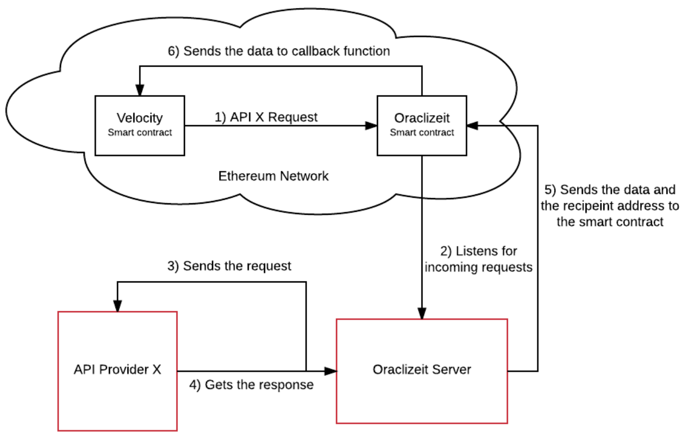


Fig. 2: Oraclizeit work flow

```
//oraclize_setNetwork(2); //
priceUrl = "json(https://www.bitstamp.net/api/v2/ticker/btcusd).last";
function updateBTCUSDFromFeed(uint delay){
  oraclize_query(delay, "URL",
    priceUrl, 400000);
  }
function __callback(bytes32 myid, string result, bytes proof) {
  if (msg.sender != oraclize_cbAddress()) throw;
    uint BTCUSDFeed;
    BTCUSDFeed = parseInt(result, 2);
  exercise() // this function exercises the contract to calculate the
↪  payouts
  }
```

Code 3: Implementation of Oraclizeit price feed in Velocity smart contract

In our testing period, we encountered multiple problems with this design:

1. The callback would not happen at all, which would result in an unsettled options contract. Oraclizeit support team were helpful and fixed this issue later on.
2. The callback would happen with some delays, which would result in inconsistency in the fetched price with the the options contract expiry date. decentralized networks have some latency by design, realtime does not really

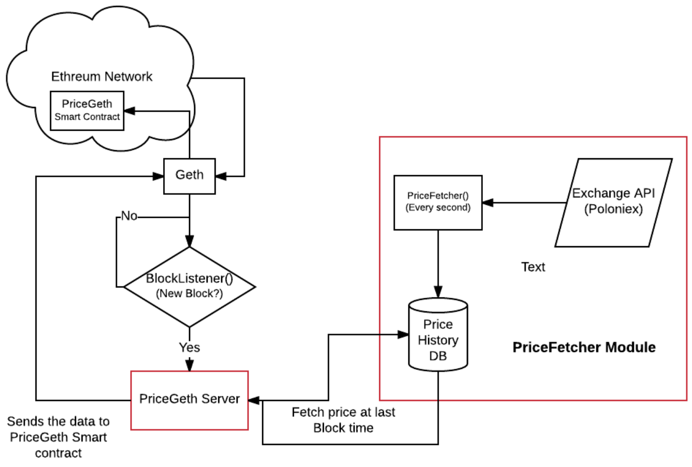

Ethereum Network
PriceGeth Smart Contract
Geth
No
BlockListener() (New Block?)
Yes
PriceGeth Server
Sends the data to PriceGeth Smart contract
Fetch price at last Block time
PriceFetcher() (Every second)
Exchange API (Poloniex)
Text
Price History DB
PriceFetcher Module

Fig. 3: PriceGeth Work Flow

mean anything in such networks, hence counting on a transaction to happen at a exact time is not the best solution.

3. The callback would happen with insufficient gas, which would result in the failure to properly run exercise() function and thus failiure to settle the options contract. Oraclizeit library offers a way to send more gas than needed in case the callback function needs more gas, however on the time of this experiment that functionality was not working properly.

PriceGeth We designed PriceGeth[9] to publish (almost) realtime price pairs to Ethereum blockchain. This is how PriceGeth works (also see Figure 3):

1. PriceFetcher server is saving an exchange Prices (USDBTC, BTCETH, BTCETC, BTCDOGE) every 1 second in a database
2. BlockListener is listening on using Geth[10] for new blocks
3. When BlockListener sees a new block it fetches the price at the Blocktime from PriceFetcher Module
4. PriceGeth server sends the data to PriceGeth smart contract(4) and updates the latest price

PriceGeth smart contract would keep all the historical prices and all would be available to all smart contracts on Ethereum blockchain for free (no gas needed to fetch the price). The reason this is almost realtime, goes back to the nature of blockchains. Time units as in seconds and minutes are not meaningful for most of the blockchain applications, but the block height can be used as the time

[9] Price API for Smart-Contracts on Ethereum Blockchain `https://github.com/VelocityMarket/pricegeth`
[10] Official Go implementation of the Ethereum protocol `https://geth.ethereum.org`

unit, meaning the time of each block is known to all users of the blockchain, but before a block is published no other time units can be used. This is why we designed PriceFetcher module to connect to an exchange API and saves the price pairs every second, to have the price for the previous block time anytime a new Ethereum block is generated.

```
struct Feed {
      uint    USDBTC;
      uint40    BTCETH;
      uint40    BTCETC;
      uint40    BTCDOGE;
      uint40  timestamp;
      uint    blockNumber;
    }
mapping (uint => Feed) priceHistory;
function setPrice(uint40 timestamp, uint40 blocknumber, uint USDBTC,
 ↪  uint40 BTCETH, uint40 BTCETC, uint40 BTCDOGE) ifOwner() {
  if (firstBlock == 0) firstBlock = blocknumber;
  priceHistory[lastBlock].timestamp = timestamp;
  priceHistory[lastBlock].blockNumber = blocknumber;
  priceHistory[lastBlock].USDBTC = USDBTC;
  priceHistory[lastBlock].BTCETH = BTCETH;
  priceHistory[lastBlock].BTCETC = BTCETC;
  priceHistory[lastBlock].BTCDOGE = BTCDOGE;
  PriceUpdated(timestamp, blocknumber, USDBTC, BTCETH, BTCETC, BTCDOGE);
}
```

Code 4: Pricegeth Main Smart contract

PriceGeth is a proof of concept implementation of having a trusted entity publishing price pairs to the blockchain and we are aware of the implications of trusting the PriceFetcher not to manipulate the prices. PriceFetcher is the central point of failure in PriceGeth design and should be addressed in future work. However after further research, it is almost impossible to have a truly trustless decentralized price feed unless we have a decentralized exchange infrastructure on the blockchain. This exchange can be used as the price oracle as the order books would be stored on the blockchain and hence there is no one single point of trust. The red boxes in Figure 3 are indicating the centralized parts of this implementation. PriceGeth is released as a stand alone smart contract and also a library to be used in other smart contracts to use the price feed free of charge[11]. Another challenge of PriceGeth design is that PricePublisher is paying the gas

[11] PriceGeth Library `https://github.com/VelocityMarket/pricegeth`

for publishing and storing all the price pairs, and as there is no incentive of doing so, it is not an inefficient way of offering price oracles. PriceGeth can be implemented in a way that clients should use a token issued to them beforehand to fetch the price, or require payments to release the price data.

By design PriceGeth operator should not be able to use Velocity options as he can manipulate the price to game the system.

There is a similar work on price feeds titled Town Crier [18], which uses TLS security to prove the fact that the data sent to the smart contract is exactly as the one provided by the API, conceptually similar to Oraclizeit TLSNotary-proof[12]. TownCrier uses Intel SGX in their central server which insures the integrity of hardware used and thus insures no manipulation is done on the server. Even though one can argue that the data provider is a trusted entity, one of the goals to have a decentralized application is to have no trusted entity in the infrastructure and to have a trustless system.

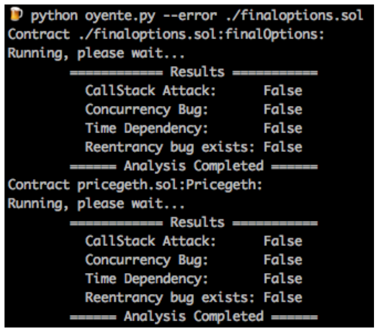
5 Discussion

Security Smart contracts have introduced some new security concerns to developers. Notions like gas usage and consensus and most importantly a function that pays out irreversible money are new to most of the developers hence the ability to develop a secure smart contract is hard to grasp. One of the visible examples of security issues is the attack on The DAO, Decentralized Autonomous Organization[13]. The goal of the DAO was to remove all the need for any venture capital intervention or any other third party for fundraising on a new idea or a company through crowdfunding and giving the investors tokens (shares) of the company. However due to an issue splitDAO function which was responsible to manage and fund new child DAOs or projects, an attacker was able to take one third of the money in the original DAO, worth approximately 86 million USD [7] at the time of the attack, this vulnerability is dubbed *Reentrancy Vulnerability*.

Luu et al. [12] developed a symbolic execution tool called "Oyente" to find potential security bugs, which they proved effective by running on Ethereum blockchain and successfully identifying The DAO vulnerability. We used this tool to analyze our code (see Figure 4).

Another family of vulnerabilities that have caused some of the known attacks are *Mishandled Exceptions*, which mostly has caused Denial of Service attacks on individual smart contracts. In Velocity main contract we used *modifier* functions to sanitize the inputs to narrow down the probability of such exceptions. Another set of attacks *Timestamp Dependence* and *Transaction-Ordering Dependence* are interesting to ponder, however due to the design of Velocity and PriceGeth, they are not applicable to these smart contracts. As an example, usage of timestamp was replaced by Ethereum blocknumber and smart contracts time is based on the block number rather than seconds and minutes. There has been more security bugs in solidity compiler, a few related bugs were explained in 4.1.

[12] `https://docs.oraclize.it/#security-tlsnotary-proof`
[13] `https://github.com/slockit/DAO`



```
python oyente.py --error ./finaloptions.sol
Contract ./finaloptions.sol:finalOptions:
Running, please wait...
        ============ Results ===========
          CallStack Attack:      False
          Concurrency Bug:       False
          Time Dependency:       False
          Reentrancy bug exists: False
        ====== Analysis Completed ======
Contract pricegeth.sol:Pricegeth:
Running, please wait...
        ============ Results ===========
          CallStack Attack:      False
          Concurrency Bug:       False
          Time Dependency:       False
          Reentrancy bug exists: False
        ====== Analysis Completed ======
```

Fig. 4: Results of Smart Contract analysis tool called Oyente [12] to find security bugs

```
    function payAndHandle(uint optionId, address addr, uint amount)
↪   private returns (bool success) {
      if (addr.send(amount)) {
            optionPaid(optionId, addr, amount); //event for successful
↪   payment
      } else { throw;}
      return true;
    }
```

Code 5: Secure payouts in smart contracts

Gas Sustainability The concept of gas usage for processing power is not easy to grasp even for long term developers. People might be familiar with limited computational or storage resources, but the concept of passing gasLimit to a function to use to process inputs is a new concept. Each step has its own estimated gas usage, as an example to store a value in a variable, you have to pay *100 Wei*[14] for each *sstore* call[15]. This should be considered that there's a cap for gas usage for each transaction and block, thus complex computation should be split into multiple transactions which makes smart contract design more complicated than they are. Also we should mention that function calls can fail due to the fact that they run out of gas and they don't have enough gas to finish their required computation or storage. This can cause unpredicted behaviour from the smart contract as there would be broken flows in the code which should have been handled by the developer. The gas usage could change as there are updates and security patches to Ethereum protocol, e.g transaction spam attack[16]. It

[14] Wei: Smallest unit of Ethereum, equevalent to 0.000000000000000001 ETH
[15] put into permanent storage
[16] Long-term gas cost changes for IO-heavy operations to mitigate transaction spam attacks `https://github.com/ethereum/EIPs/issues/150`

might take multiple implementation of the same function to find an equilibrium between readability and gas efficiency.

Misuse of the contract In the current implementation of Velocity smart contract, one can call the Long option when he is sure of the price increase between the start time and expiry time and keep on doing this until there is no money left in the smart contract's pool of funds. This is because the smart contract calls the opposite of the incoming option call blindly. However in future work, there should be market scoring rule which depends on how many short option calls are placed comparing to the long calls and make it more expensive to call short when there are more short option calls than long calls.

Collar Option library Velocity smart contract can be used as a module in any other smart contract to handle option calls and execute some functions on the expiry time. This smart contract was written as a proof of concept and was released under *GPL* license[17].

6 Future work

As discussed in subsection 4.2, fully decentralized Price feeds and oracles are needed in order to have a trustless decentralized financial market. This can be done by having a decentralized exchange to extract prices from using smart contracts. Even though there has been many price feed methods discussed, none of them seem to have trustless infrastructure. Smart contracts security is not well practiced and there are many unknown attack vectors in the eco system, from solidity compiler security bugs [15] to best practice security implementations [6], there is work to be done and tests to have a more mature secure eco-system to work with, Specially if the end goal is to have a decentralized financial application in place where money is at stake.

As for the options contracts, there should be more research and work on the payouts to make them smarter. One proposed solution is to have market scoring rules in place, which means if there are more open short option calls than long calls, it should get more expensive to call short options and vice-versa. Smart contracts are unchangeable piece of code that run autonomously, meaning if there's a market crash or systematic error, there cannot be anything to do to suspend the payouts and shut down the application, unless with pre-defined functions in the smart contract which only the owner can trigger, which would be a double standard in the trustless eco-system.

7 Conclusion

Even though the idea of having a fully autonomous and decentralized derivative market is intriguing, the infrastructure to reach this goal is still missing from the

[17] `https://github.com/VelocityMarket/Options-Contract`

underlying network. As for example, price feed is one of the essentials of such a market and it should be done in a fully decentralized trustless way to prevent fraud and market manipulation by the feed provider. All the existing solutions today, have a central point that can manipulate data, it is either the exchange API or the component responsible to publish the price. As discussed before, one of the only solutions to this problem is to have a fully decentralized exchange on the network to provide realtime price feed for other smart contracts. There are some work done on decentralized exchanges [5], although there is no real world deployment of such a system at the time of writing. Smart contracts are fascinating idea that can revolutionize the technology by removing the middlemen, however the underlying technology is more on the proof of concept level than mature enough to be used on the real world scenarios. We should also mention that the barrier for people to have the relevant crypto-currency to work with such systems still exists.

References

1. N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts. In *POST*, 2017.
2. J. Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv:1407.3561*, 2014.
3. I. Bentov, A. Mizrahi, and M. Rosenfeld. Decentralized prediction market without arbiters. *arXiv:1701.08421*, 2017.
4. V. Buterin et al. A next-generation smart contract and decentralized application platform, 2014.
5. J. Clark, J. Bonneau, E. W. Felten, J. A. Kroll, A. Miller, and A. Narayanan. On decentralizing prediction markets and order books. In *WEIS*, 2014.
6. ConsenSys. Ethereum contract security techniques and tips. *ConsenSys*, 2016.
7. P. Daian. Analysis of the dao exploit. *Hacking, Distributed*, 2016.
8. S. Demian Lerner. Rootstock: Bitcoin powered smart contracts. *Whitepaper*, 2015.
9. K. Finley. A 50 million dollar hack just showed that the dao was all too human. *wired*, 2016.
10. D. Irvine. Maidsafe distributed file system, 2010.
11. B. Kievit-Kylar, C. Horlacher, M. Godard, and C. Saucier. Equibit: A peer-to-peer electronic equity system. *arXiv:1612.06953*, 2016.
12. L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *CCS*, 2016.
13. S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
14. okturtles. A blockchain-based dns, http server that fixes https security, 2014.
15. C. Reitwiessner. Security alert: Solidity variables can be overwritten in storage. *Ethereum Blog*, 2016.
16. N. Szabo. The idea of smart contracts, 1997.
17. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 2014.
18. F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. Town crier: An authenticated data feed for smart contracts. In *CCS*, 2016.

A Demo Website (UI) for the Velocity smart contract

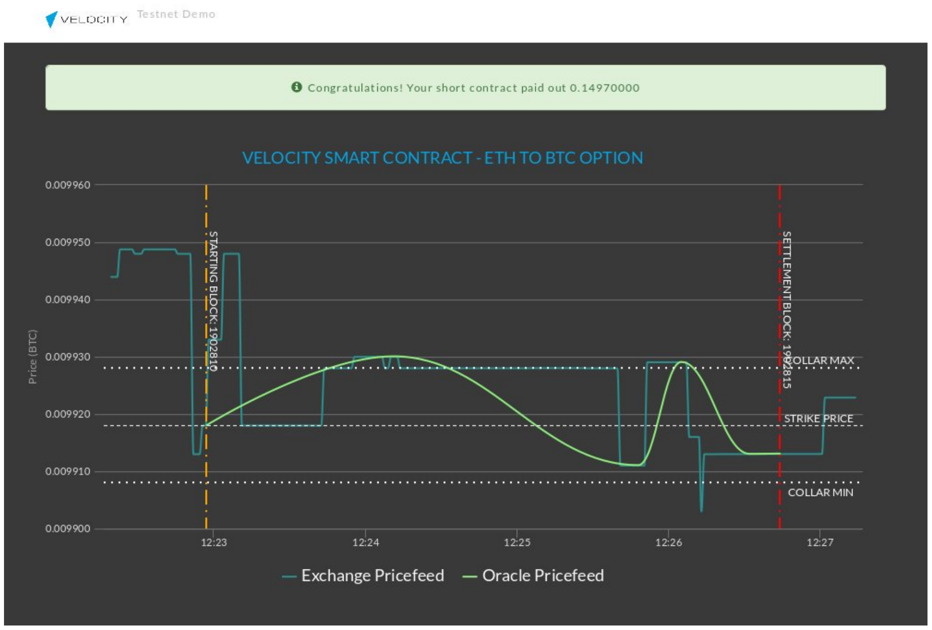

Congratulations! Your short contract paid out 0.14970000

Fig. 5: Velocity Options Smart Contract Demo

SmartCast: An Incentive Compatible Consensus Protocol Using Smart Contracts

Abhiram Kothapalli kothapa2@illinois.edu
Andrew Miller soc1024@illinois.edu
Nikita Borisov nikita@illinois.edu

Abstract

Motivated by the desire for high-throughput public databases (i.e., "blockchains"), we design incentive compatible protocols that run "off-chain", but rely on an existing cryptocurrency to implement a reward and/or punishment mechanism. Our protocols are incentive compatible in the sense that behaving honestly is a weak Nash equilibrium, even in spite of potentially malicious behavior from a small fraction of the participants (i.e., the BAR model from Clement et al. [8]). To show the feasibility of our approach, we build a prototype implementation, called SmartCast, comprising an Ethereum smart contract, and an off-chain consensus protocol based on Dolev-Strong [11]. SmartCast also includes a "marketplace" smart contract that randomly assigns workers to protocol instances. We evaluate the communication costs of our system, as well as the "gas" transaction costs that are involved in running the Ethereum smart contract.

Keywords— atomic broadcast, TRB, game theory, Ethereum, smart contracts

1 Introduction

Bitcoin and related cryptocurrencies have sparked renewed interest in decentralized consensus protocols, as exemplified by the so-called blockchain technologies. It turns out that many applications (including complementary currencies, certificate revocation [15, 7], directory authorities for p2p networks like Tor [10]), benefit from a globally agreed-upon sequence of transactions. Currencies such as Bitcoin and Ethereum use the proof-of-work mining to distribute the responsibility for maintaining the blockchain integrity to a large collection of parties; the integration of mining with a financial reward makes this collection difficult to subvert. However, the global nature of this ledger creates some inherent costs, both in terms of transaction costs and the agreement latency. An alternative approach is what has been termed a *permissioned ledger*, where the role of miners is taken by a trusted coalition of parties, whose motivation to properly follow the protocol is assumed to come externally.

Several applications of blockchains, however, would benefit from a middle ground between these two extremes. Loosely defined coalitions, such as food banks, cooperatives, or student organizations, are some times in need of a blockchain-like ledger for tracking membership benefits or exchanges between sister organizations; however, they would not have the resources to directly operate a reliable collection of "miners," nor, necessarily agree on a set of trusted parties. At the same time, directly using cryptocurrency for account deposits might limit their accounting flexibility and incur non-trivial transaction costs.

Our approach creates a system where workers who act to enforce integrity are financially rewarded for their correct participation in the process, as monitored by other workers and enforced through an Ethereum smart contract. Our protocol draws inspiration from a consensus protocol designed by Clement et al. [8], where honest participation is shown to be a *rational* strategy for participants trying to maximize their utility. Their protocol, however, assumes that workers derive *intrinsic* utility from the correct functioning of the protocol and requires an infinite time horizon; in our scenario, which we believe to be more realistic, we expect consensus to be enforced by inherently disinterested parties whose only motivation is financial. This *extrinsic* reward dramatically simplifies the protocol design and improves its efficiency. Our protocol requires only occasional communication with the Ethereum blockchain through the smart contract, thus minimizing transaction costs.

To design our protocol, we create a *generic* transform that renders any existing protocol where communication is the dominant cost *incentive-compatible*. We show that, under this transform, honest participation is a weak Nash equilibrium in a worst-case utility model, which was previously used by Clement et al. [8].

To show the feasiblity of our approach, we build a prototype implementation, called SmartCast, comprising an Ethereum smart contract and an "off-chain" consensus protocol (based on the Dolev-Strong [11] broadcast protocol). . We evaluate the communication costs of our system, as well as the "gas" transaction costs that are involved in running the Ethereum smart contract. We additionally describe how these protocols can be deployed in practice with random consensus nodes.

1.1 Related Works

Several previous works have proposed using cryptocurrencies to enforce properties in off-chain protocols. Bentov and Kumaresan's protocol [1] guarantees either a fair output or else financial compensation to each honest party, but requires significant collateral deposits. In contrast, our weak Nash equilbirium notion guarantees that parties cannot benefit by deviating. In a separate line of work, Garay et al. design a general framework to build protocols that are resilient against rational adversaries [12]. We instead design a protocol transformer that can achieve resilience for a certain class of protocols. To the best of our knowledge, we are the first to harness smart contracts for the purpose of Byzantine fault tolerance.

2 Background and Preliminaries

2.1 Network Model

Our basic computation model is the standard point-to-point network setting with synchronous authenticated channels. We consider a fixed set of parties $\mathcal{N}$, where an individual party is denoted $p \in \mathcal{N}$. The protocol proceeds in rounds of communication, with the exact order of messages in each round may be arbitrary (i.e., chosen by the adversary). Messages not delivered within the round are invalid. Each party is associated with a common-knowledge public signing key to send authenticated messages. Our model accounts for Byzantine failures. The adversary can choose to corrupt a subset of nodes $\mathcal{B} \subset \mathcal{N}$, giving them complete control over these nodes. $|\mathcal{B}|$ is bounded by a parameter b.

2.2 Smart Contract Protocols

Public cryptocurrencies [5] (or "blockchain") systems, such as Bitcoin [18] and Ethereum [21], provide a decentralized platform for programmable money. These can be used as general-purpose trusted third parties, but with caveats. For instance, they can be trusted for correctness, but do not provide any inherent privacy. For some applications, privacy can be provided by a layer of multi-party computations and zero-knowledge proofs [13, 2]. A second caveat is that blockchain transactions are expensive (because they are fully replicated throughout the entire cryptocurrency network), so it typically is not cost-effective to carry out protocols directly on top of the blockchain.

A protocol in the smart contract model is therefore most effective with two components: 1) A smart contract program, which receives reports from nodes about each other, and dispenses rewards at the end; and 2) Local code for each of the parties to execute, including interactions with the smart contract and participation in "off-chain" subprotocols. We also assume a rushing adversary, who can observe the smart contract transactions sent by non-Byzantine parties before submitting transactions on behalf of the Byzantine parties.

2.3 Utilities in the BAR Model.

We adapt the The Byzantine-Altruistic-Rational (BAR) fault tolerance model from Clement et al. [8] to the smart contract setting. The BAR model is a game-theoretic layer on top of the standard distributed protocol execution model. That is, we view the choice of what code to run (i.e., either following the protocol or deviating in some arbitrary way) as a strategic decision.

We associate each "off-chain" protocol message with a *cost* to the sender of that message, determined by the total size of the messages sent. However, we ignore any other costs of computation, storage, and other resources. We thus assume that the total utility of each party therefore depends on the monetary payments disbursed by the smart contract, minus the cost of the messages they send. Since the protocol execution is probabilistic, unless indicated otherwise we are concerned with the expected utility.

As Clement identifies, in an ordinary protocol (i.e., without the smart contract incentive mechanism in place), parties may be able to profit by deviating from the protocol — in particular by withholding messages to reduce their costs (i.e., by acting "lazy"). Thus the high level approach is to punish lazy nodes.

A strategy profile $\vec{\sigma}$ defines a program for each party p in $\mathcal{N}$ to run. Given a protocol ρ, we use the symbol $\vec{\rho}$ to denote the *prescribed strategy*, in which every party follows the protocol correctly.

While standard distributed systems models feature a worst-case adversary, and standard game models feature a set of strategic (i.e., "rational") players, the intersection of these has yet to be studied widely. Clement proposes the following notion of "worst-case utility," which we also adopt.

Definition 1. *Worst-case Utility. The worst case utility $\bar{u}_p(\sigma)$ for a rational player $p \in \mathcal{N}$ is where p follows strategy σ, every non-Byzantine player in $N - \mathcal{B} - \{p\}$ follows the prescribed strategy, $\vec{\rho}_{N-\mathcal{B}}$, and the choice of Byzantine players $\mathcal{B}$ and their strategies $\vec{\tau}_{\mathcal{B}} \in \mathcal{S}_{\mathcal{B}}$ are chosen to minimize the resulting utility u_p. This is defined more precisely as:*

$$\bar{u}_p(\sigma) \triangleq \min_{\mathcal{B} \subset \mathcal{N} : |\mathcal{B}| \leq b} \circ \min_{\vec{\tau} \in S_{\mathcal{B}}} \circ \, u_p(\vec{\rho}_{\mathcal{N}-\mathcal{B}-\{p\}} + \sigma + \vec{\tau}_{\mathcal{B}}) \tag{1}$$

Our goal is then to show that the prescribed strategy is a worst-case *weak Nash equilibrium*, i.e., $\bar{u}_p(\rho_p) \geq \bar{u}_p(\sigma)$ for any σ. That is, a rational party cannot *improve* their expected utility by following any other deviant protocol σ. This solution concept could be thought of as modeling paranoid players who think that other parties (up to b of them colluding) are "out to get them."

2.4 Synchronous Byzantine Agreement

Alternative definitions of consensus primitives abound in the distributed systems literature. Perhaps the strongest of these — and the one most naturally suited to our application scenario — is "atomic broadcast." This primitive allows any of the N protocols parties to submit input values, and the parties all reach agreement on an ordered sequence of values that at least includes the inputs from each honest party. Atomic broadcast could thus be described as the "blockchain" primitive in today's post-Bitcoin parlance.

Below we provide a more formal definition of this primitive, adapted for the synchronous setting. We assume that each input value is bounded by a maximum message size C, and that the protocol finally terminates after a maximum number of rounds $r^\dagger$.

Definition 2. *Bounded Atomic Broadcast: Given a set of players $\mathcal{N}$, each process p in $\mathcal{N}$ receives inputs $m_{p,r} \in \{0,1\}^C$ in round r.*

- *(Termination):* after a bounded number of rounds $r^\dagger$, each correct process terminates.
- *(Agreement):* The sequence of outputs $V_{p,r}$ in round r by each correct process p are all identical, i.e. $\forall r, \forall p, q \in \mathcal{N} - \mathcal{B}. V_{p,r} = V_{q,r}$.
- *(Validity):* every input from a correct node (received before $r \leq r^\dagger$) is included in $V_{r^\dagger}$.

Looking ahead, in Section 3.4 we implement an atomic broadcast protocol by composing a simpler primitive, called Terminating Reliable Broadcast (TRB). In TRB, one of the parties is designated as the leader, and only the leader may input messages. Thus in TRB there is no need to apply an ordering to messages from different sources, and if the leader is faulty then the parties may need to output a default value $\perp$.

Definition 3. *Terminating Reliable Broadcast Given a set of players $\mathcal{N}$, among which one, D, is designated the leader and receives an input $m \in \{0,1\}^C$ (i.e., within some bounded message size of C bits), a Terminating Reliable Broadcast protocol must satisfy the following properties:*

- *(Termination):* Every correct process p delivers some value $m \in \{0,1\}^C \cup \{\perp\}$ after a bounded time r^*.
- *(Agreement):* If any correct process delivers m, then all correct processes deliver m.
- *(Validity):* If the leader D is correct, then every correct process delivers D's input m.

Alternative network models. Although our SmartCast protocol relies on a synchronous network model. This is generally considered a strong assumption. Other protocols such as PBFT [6] provide security in the more challenging weakly synchronous setting — they meet the lower bound in this model, which is $b \leq N/3$. However, synchrony is an assumption we must take anyway if we rely on a smart contract system in the style of Bitcoin and Ethereum. It is not clear how to adapt our protocol to the asynchronous setting, since we would not be able to detect whether a message was omitted by a party or just delayed in the network.

3 Smart Contracts for Incentive Compatible Protocols

In this section we present our main contribution, a protocol transformer, $\mathsf{SmartBAR}(\cdot)$, which transforms an arbitrary synchronous protocol with costly communication, π into an incentive compatible protocol $\mathsf{SmartBAR}(\pi)$. As an application, in Section 3.4 we apply this transformation to yield an incentive-compatible consensus protocol, called SmartCast.

At a high level, $\mathsf{SmartBAR}(\cdot)$ adds a smart contract layer to π that implements a reward/punishment mechanism. In an ordinary protocol (i.e., without this incentive mechanism in place), parties may be able to profit by deviating from the protocol — in particular by withholding messages to reduce their costs (i.e., by acting "lazy"). To ensure that laziness is not profitable, our protocol enlists the honest parties to detect lazy nodes and the smart contract to punish them.

The transformation works for an arbitrary synchronous protocol π that satisfy the following assumptions. First, each correct party in π terminates after a bounded number of rounds r^*, for some parameter r^*. Second, the total number of bits between any pair of parties is bounded by a value M. We call a protocol that satisfies these an $(r^*, M)-$*bounded* synchronous protocol.

Since the transformation runs π in place, any fault tolerance properties of π still carry over to $\mathsf{SmartBAR}(\pi)$. In particular, if π tolerates b faults, and we prove that running is a b-worst-case equilibrium, then the security of the overall protocol security reduces to the assumption of strategic behavior among the rational remaining parties.

3.1 The Protocol Transformer $\mathsf{SmartBAR}(\cdot)$

The transformed protocol $\mathsf{SmartBAR}(\pi)$ runs π with the following minimal modifications:

- Modification 1: We impose a predictable communication pattern so that nodes can detect if another is cutting costs by not forwarding messages. Our predictable communication pattern requires that in each protocol instance, node p must send every node q the maximum possible total message size M. If fewer than M message bits are sent by the end of the protocol, then dummy messages are sent to make up the difference.
- Modification 2: We impose a penalty on nodes that fail to forward messages, by implementing the following rules:
  - Each node keeps track of the total message bits received from each other node.
  - At the end of the protocol, if fewer than M bits have been received by p from q, then p sends a report $R_{p,q} = \texttt{enemy}$ to the smart contract. Otherwise, if at least M bits have been received, then p sends a report $R_{p,q} = \texttt{friend}$.
  - The smart contract waits until the final round of the protocol r^* to collect status reports from all nodes $p \in \mathcal{N}$. Finally, the smart contract determines the payout to each party by deducting a penalty of θ (a parameter discussed shortly) for each `enemy` report about that party.

Alternative definitions of `enemy` Note that we propose a relatively lenient definition for `enemy` as a node that does not send at least M bits. This protects honest nodes with harmless or negligible deviations from being marked as dishonest by other honest nodes. On the other hand, we can follow a much stricter definition of enemy by marking nodes that do not send at least M bits, send incorrect bits, send more than M bits, and so on. This leads to additional protocol security by barring more forms of misbehavior, but may

Protocol $\mathsf{Smart}(\pi)$ for a bounded synchronous protocol π, a set of parties $\mathcal{N}$, and a maximum number of Byzantine nodes $b < |\mathcal{N}| - 2$.

Let r^* be a bound on the final round before π terminates.

Let M bound the total size of messages sent between any pair of parties in π.

Local program (for node p).

- Run the given protocol π_p.
- For each received message m, parse m as either an ordinary message $\texttt{PASS}(m')$ (in which case pass m' through to π_p) or else a padding message $\texttt{DUMMY}(0^*)$, in which case discard this message.
- For each outgoing message m generated by π_p, intended for player q, send $\texttt{PASS}(m)$ to q.
- At the final round r^*, let $M_{p,q}$ be the total size of messages sent so far to q (including any messages sent during this round). If $M_{p,q} < M$, then send a padding message $\texttt{DUMMY}(0^{M-M_{p,q}})$.
- After r^*, for each player $q \neq p$, let $M_{q,p}$ be the total size of messages received from q. If $M_{q,p} < M$, then set $R_{p,q} := 0$ (an enemy report). Otherwise, set $R_{p,q} := \theta$ (a friend report). Finally, send a transaction containing $\texttt{report}(\vec{R_p})$ to the smart contract, where $\vec{R_p} = \{R_{p,q}\}_{q \neq p}$ is the vector of all of the reports from p about each other player q.

Smart contract program.

- The contract is parameterized by a set of players $\mathcal{N}$, identified by their addresses (i.e., public keys).
- The contract must be initialized with an endowment (a quantity of digital currency) of at least $E \geq (|\mathcal{N}|)(|\mathcal{N}| - 1) \cdot \theta$, where $\theta = \frac{|\mathcal{N}|-1}{|\mathcal{N}|-1-b} M$.
- The contract contains an entry point $\texttt{report}(\vec{R_p})$, which when invoked by party p, stores the argument vector $\vec{R_p}$.
- By a fixed deadline time T, the contract receives a report $R_{p,q} \in [0, \theta]$ from each party $p \in \mathcal{N}$ about each other party $q \in \mathcal{N}$. Any reports that are not received in time are treated as a default value of 0.
- After time T, for each $p \in N$,
  - determine the reward as the sum of reports about p, so $\mathsf{reward}_p := \sum_{q \in \mathcal{N} | q \neq p} R_{q,p}$, and send reward_p to player p

Figure 1: Our protocol transformer, $\mathsf{Smart}(\cdot)$, which provides incentive-compatibility for an arbitrary synchronous protocol. Each party pads outgoing messages to the maximum size, and reports to the smart contract about any "lazy" peers.

unnecessarily penalize honest nodes that perform harmless or negligible deviations.

The entire SmartBAR$(\cdot)$ protocol is defined in Figure 1. For simplicity, we assume the smart contract is initialized with an endowment $E \geq N(N-1)\cdot\theta$. In practice, this endowment might be provided by collecting collateral deposits from the participants or collecting usage fees from users of the system, as described shortly in Section 3.5. We next describe how the parameter θ is determined in order to satisfy the worst-case equilibrium notion.

3.2 Rationality Analysis

We now prove that following the SmartBAR protocol is a worst-case weak Nash equilibrium. The utility for party $p \in N$ as a function of a strategy vector $\vec{\sigma}$ is $u_p(\vec{\sigma}) = \mathit{benefits}_p(\vec{\sigma}) - \mathit{costs}_p(\vec{\sigma})$. The overall benefits will be decided by reward_p and the overall cost is $\mathit{cost}_{msg} + \mathit{cost}_{report}$. In the following, we use the notation $\vec{\sigma}_{N-\{p\}} + \vec{\sigma}_p$ to denote the union of the strategy vectors $\vec{\sigma}_{N-\{p\}} + \vec{\sigma}_p$.

In order to prove that rational parties gain the highest utility by following the recommended protocol, we take the following steps: First we show a lower bound that following the protocol earns p a minimum utility u^*, regardless of the adversary's choice of strategies. Next, we partition the space of alternative strategies into classes based on how they behave towards honest nodes. We define a simple family of strategies, called the "indiscriminate" strategies, which act as representatives of these classes. We can prove that the indiscriminate strategies perform just as well (in the worst-case) as any other strategy. Finally, we show how to choose the protocol parameter θ so that u^* is an upper bound for the utility of any indiscriminate strategy. The setting of θ directly determines the overall collateral cost (i.e., the required endowment) for the protocol.

Lemma 1. *Regardless of the strategy $\vec{\tau}_{\mathcal{B}}$ followed by Byzantine parties, if p follows $\vec{\rho}_p$, then p obtains at least* $\bar{u}_p(\rho) \geq u^*$ *where* $u^* \triangleq (N-1)\theta - (N-1)M - b\theta$.

Proof. The ideal reward of the protocol is initially set to be $(N-1)\theta$. The prescribed strategy sends all possible messages, incurring the maximum message cost $(N-1)M$. Since all the non-Byzantine nodes report p as a `friend`, the maximum report cost can be at most $b\theta$, which occurs when all b Byzantine nodes report `enemy`. □

This bound holds regardless of how the protocol parameter θ is chosen. This worst-case utility is incurred when the Byzantine parties follow the spiteful strategy.

The indiscriminate strategies, α_γ. We next turn towards proving an upper bound on the utility of deviating from the prescribed strategy ρ. We first define a family of simple strategies, α, which we call the indiscriminate strategies. Looking ahead, these strategies will serve as representatives for a partioning of the overal strategy space. The indiscriminate strategies α by a fraction $0 \leq \gamma \leq 1$, such that α_γ misbehaves towards each other node with probability γ. More precisely, α_γ is defined as follows: At the beginning of the game, for each other party q a coin is flipped with probability γ (for some arbitrary percentage γ). If the coin flip succeeds, then p refuses to send any messages to q; otherwise p sends messages to q according to the ordinary protocol.

The strategy α_γ clearly causes p to incur a message cost of $(1-\gamma)(N-1)M$. Since this strategy witholds messages from exactly $\gamma(N-1-b)$ honest uncorrupted parties in expectation, the worst-case expected report cost is $(b + \gamma(N-1-b))\theta$. We therefore have the following claim:

Claim 1. *The worst-case expected utility for the strategy α_γ is*

$$\bar{u}_p(\alpha_\gamma) = (N-1)\theta - (N-1)(1-\gamma)M - (b + \gamma(N-1-b))\theta \tag{2}$$

The Spiteful Strategy, δ. Following Clement et al. [8], we define a particular adversarial strategy, called the spiteful strategy, which serves as a worst-case adversary (as we will see shortly). The spiteful strategy initially behaves according to the prescribed strategy, but in the final round it always reports `enemy` for player p, causing p to be punished.

If rational party p could determine which nodes were corrupted, then p would be able to cut his losses by withholding messages from just the nodes in $\mathcal{B}$. The spiteful strategy, however, blends in with the honest parties. As shown by the following lemma, this means p can do no better than to withhold messages from other nodes chosen uniformly at random, as with the indiscriminate strategy α_γ. In the following, we say that player p follows an *acceptable message sequence* towards player q if p sends q a total of at least M bits.

Lemma 2. *Consider a strategy σ_γ, such that in an execution with all honest parties (i.e., with the strategy vector $\{\sigma_\gamma\} + \rho_{\mathcal{N}-\{p\}}$), party p sends an unacceptable message sequence to exactly $\gamma(N-1)$ nodes in expectation. Then the worst-case utility $\bar{u}(\sigma_\gamma)$ is at most $\bar{u}(\alpha_\gamma)$.*

Proof. Let γ_q be the probability that σ_γ sends an unacceptable message sequence to party $q \in \mathcal{N} - \{p\}$ when all parties besides p follow the protocol. By assumption, we know that

$$\sum_{q \in \mathcal{N}-\{p\}} \gamma_q = \gamma.$$

First, note that against the spiteful adversary, p incurs an expected message cost of at least $(1-\gamma)(N-1)M$. Next, to bound the report cost, we will choose $\mathcal{B} \subseteq \mathcal{N} - \{p\}$, with $|\mathcal{B}| = b$, such that we minimize $\sum_{q \in \mathcal{B}} \gamma_q$. This minimization guarantees that p sends an unacceptable message sequence to at least $(N-1-b)\gamma$ honest nodes in expectation, resulting in an expected report cost of at least $(b + (N-1-b)(\gamma))\theta$. □

Note the above proof above holds regardless of whether probabilities γ_q are independent.

Choosing the parameter θ. We want to choose θ so that deviating from ρ cannot improve the worst-case expected utility. Starting from Lemma 2, it will suffice if we can guarantee that $\bar{u}_p(\rho) \geq \bar{u}_p(\alpha_\gamma)$ for all γ. We therefore solve the following:

$$\bar{u}_p(\rho) \geq \bar{u}_p(\alpha_\gamma) \tag{3}$$

$$(N-1)M + b\theta \leq (N-1)(1-\gamma)M + (b + (N-1-b)(\gamma))\theta \tag{4}$$

$$\frac{N-1}{N-1-b}M \leq \theta \tag{5}$$

Theorem 1. *If π is a synchronous protocol that terminates after r^* rounds and each party sends a maximum of M message bits to each other party, then the transformed protocol $\mathsf{Smart}(\pi)$ is a worst-case weak Nash equilibrium.*

Proof. When the SmartTRB protocol is instantiated with θ defined as in Equation 5, from Lemma 1 we have that the worst-case expected utility when p follows the protocol $\bar{u}_p(\rho)$ is at least as good as any indiscriminate strategy $\bar{u}_p(\alpha_\gamma)$. And from Lemma 2, we know that the indiscriminate strategies perform as well in the worst-case as any other strategies. □

3.3 Comparison with the BAR Primer [8].

Our protocol and analytical framework is adapted from the bar model of Clement et al. [8], but with several significant differences.

While Clement's model requires an infinitely repeated game, our model considers the bounded case. In the infinite settings, rational parties simply play tit-for-tat, immediately and irrevocably "retaliating" against nodes that misbehave, preventing them from all future rewards. In a finite setting, a node could misbehave in the final round without fear of retaliation.

Additionally Clement's model assumes that nodes gain a positive utility from the correction execution of the protocol itself. Alternatively, in our model, nodes gain a positive utility monetary payments disbursed by the smart contract. We believe our utility model is more realistic, especially in a marketplace setting (like that discussed in Section 3.5) where the participants in a protocol instance are randomly assigned from some population of available workers.

Together, these two modelling differences require a significant change to the protocol and proof. First, while "retaliation" in Clement's model involves requiring nodes to send expensive "penance" messages (since that is the only way to inflict a punishment in their model), the smart contract provides a direct alternative. Second, in the finite setting we must rule out deviant strategies that withhold messages in a possibly randomized way, even in the last round, as though "guessing" at which parties might be corrupted. We overcome this by introducing a new family of "indiscriminate strategies" that serve as simple representatives of the full strategy space. Finally, like Clement, our proof involves a "spiteful strategy," that acts as a worst case adversary. However, the "spiteful strategy" is different in our model: it misbehaves only in the final round, after it is too late for the victim p to retaliate.

3.4 SmartCast: An Incentive Compatible Consensus Protocol

As an application of our general protocol transformation, we now describe how to apply our $\mathsf{SmartBAR}(\cdot)$ transformation to a classic synchronous protocol, $\mathsf{DolevStrong}$ [11], in order to obtain an incentive-compatible off-chain consensus protocol.

The Dolev-Strong protocol for Terminating Reliable Broadcast. The Dolev-Strong protocol is a classic algorithm for synchronous byzantine agreement using signatures, that achieves optimal resilience by tolerating $N - 1$. However, it provides no explicit incentives for participants to follow. As seen in Clements et al., individual participants in the protocol can reduce their computational cost by omitting messages.

The protocol runs for $b+1$ rounds, where the leader D sends a signed message containing its input to each of the other nodes in the first round. Each node that receives the leader's message in the first round "accepts" the message, and then appends their own signature and relays the message to every other node. If the leader fails to send a message to some node p, some other node q will relay the message to p in any round r, as long as the relay contains at least r signatures. p will then continue to relay the message. If the leader equivocates, it is possible that a node accepts two or more distinct values. In this case, a node outputs $\perp$, and only relays the first two such values received. In total, each node must therefore send a maximum of $2N$ total messages, each containing an input value and up to $b + 1$ signatures.

We provide a listing of the Dolev-Strong algorithm in Figure 2, adapted from Kumaresan's thesis [14]. For a proof of security we refer the reader to [11, 14].

We let $D \in \mathcal{N}$ denote a designated leader. We let $m \in \{0,1\}^C$ denote the sender D's input, and sk_D its secret key.

- (Stage 1): The leader D sends $(m, \mathsf{sign}_{\mathsf{sk}_D}(m))$ to every party. It then outputs m and terminates the protocol. Each other party p initializes $\mathsf{ACC}_p := \emptyset$, and $\mathsf{SET}_p := (v \mapsto \emptyset)$, a mapping from values to (initially empty) sets of signatures.
- (Stage 2): In rounds $r = 1, ..., b + 1$, perform the following:
  - If a pair (v, SET) is received from some q, with $v \in \{0,1\}^C$, and if SET contains valid signatures on v from at least r distinct parties including the leader D, and if ACC_p contains only 0 or 1 values, then p updates $\mathsf{ACC}_p := \mathsf{ACC}_p \cup \{v\}$, and $\mathsf{SET}_p[v] := \mathsf{SET}_p[v] \cup \mathsf{SET}$.
  - Each party p checks whether any value $v \in \{0,1\}^C$ was newly added to the set of accepted values ACC_p during round $r - 1$. In this case, p computes $\mathsf{sign}_{\mathsf{sk}_p}(v)$, and sends $(v, \mathsf{SET}_p[v] \cup \{\mathsf{sign}_{\mathsf{sk}_p}(v)\}$ to every other party.
- (Stage 3): If $\mathsf{ACC}_p = \{v\}$ for some v, then p outputs v. Otherwise p outputs $\bot$.

Figure 2: Definition of the DolevStrong protocol [11] for Terminating Reliable Broadcast (adapted from Kumaresan [14])

From Reliable Broadcast to Atomic Broadcast. Atomic broadcast further guarantees that messages can be committed by any node, not just a leader. In a synchronous network, atomic broadcast can be easily built from terminating reliable broadcast, simply by having nodes take turns becoming leaders. In brief, each node maintains a buffer of input values that have not been committed yet. When it is node p's turn as leader, p broadcasts the set of elements in its buffer. When each turn ends, the nodes remove any newly committed elements from their buffers.

3.5 Deploying Consensus Protocols with Smart Contracts

So far, we have discussed protocols assuming a fixed set of parties, with collateral provided abstractly by a benefactor. We now describe an alternative deployment scenario where many independent SmartCast instances are run concurrently, and where the participants in each are randomly drawn from a large population of potential workers. Our idea is to build a smart contract-based marketplace, SmartCast-Market, that matches up workers to protocol instances.

A naïve approach might be to allow participants to join a SmartCast instance a first-come-first-serve basis. This naïve solution would be vulnerable to Sybil attacks, where malicious nodes join as fast as they can with numerous pseudonyms, hoping to fill up all of the slots in a protocol and therefore crowd out honest nodes. Instead, our solution is to allow nodes to join a pool of workers, and to allow task creators to deposit collateral and add to a pool of pending tasks. Every epoch, workers are assigned to tasks in a randomized batch. This prevents nodes from gaining too much influence within any particular protocol instance.

If all participants in an instance follow the protocol, then the total payment for a task must be $P = N(N-1)\theta$. In principle, this could be collected from a combination of up-front payment from the task creator, along with collateral deposits from the participants themselves. Since participation is voluntary, we should ensure as a guideline that workers never lose money by participating in the protocol. Thus if they deposit collateral, they must

```
contract SmartCast {
    mapping(address => uint) playermap;
    bool[] reported;
    address[] players;
    uint[] rewards;
    uint theta;
    uint deadline; // Deadline to receive reports

    function assert(bool b) internal { if (!b) throw; }
    modifier after_ (uint T) { if (block.number >= T) _; else throw; }
    modifier before(uint T) { if (block.number <  T) _; else throw; }
    modifier onlyplayer() { if (playermap[msg.sender] != 0) _; else throw; }

    function SmartCast(address[] _players, uint _theta, uint _deadline) {
        var N = players.length;
        // Each player earns up to N * theta if they receive all good reports
        assert(msg.value == N * N * _theta);
        theta = _theta;
        deadline = _deadline;
        for (var p = 0; p < _players.length; p++) {
            players.push(_players[p]);
            rewards.push(0);
            playermap[_players[p]] = (p+1);
        }
    }
    function report(uint[] reports) onlyplayer before(deadline) {
        var p = playermap[msg.sender] - 1;
        assert(!reported[p]); reported[p] = true; // only report once
        assert(penalties.length == players.length);
        for (var q = 0; q < reports.length; q++) {
            var report = reports[q];
            assert(report >= 0);
            assert(report <= theta);
            rewards[q] += report;
        }
    }
    function withdraw() onlyplayer after_(deadline) {
        var i = playermap[msg.sender] - 1;
        if (!msg.sender.send(balance[i])) throw;
        balance[i] = 0;
    }
}
```

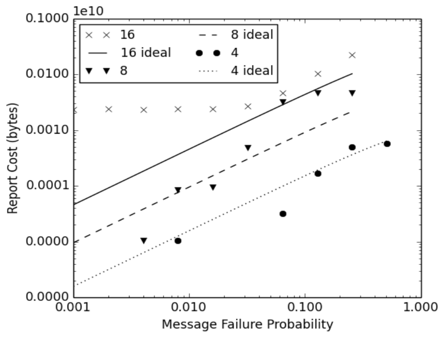
Figure 3: Implementation of the Smart contract in Solidity.

get at least that collateral back (in expectation) despite a worst-case adversary. However, since the parameter $\theta = \frac{N-1}{N-1-b}M$ is chosen minimally, in the worst-case each honest party just breaks even, receiving only $(N-1-b)\theta$ in payment but incurring an equal message cost of $(N-1)M$. Therefore there is no opportunity for collateral deposits to contribute to the necessary endowment. Thus the task creator must pay up-front the maximum payment $N(N-1)\theta$, although the maximum total message cost is only $N(N-1)M$. Hence, the task creator potentially pays an overhead of $\frac{N-1}{N-1-b}$ above the raw cost of the resources used.

4 Implementation and Evaluation

To evaluate the practical limitations of the SmartCast protocol, we develop a prototype implementation of both the Dolev-Strong consensus algorithm and an Ethereum smart contract capable of assigning various nodes to arbitrary consensus tasks.

Our Dolev-Strong implementation is written in Python, using ordinary threads and TCP sockets, with messages signed using the `ed25519` signature scheme. We evaluated our protocol by running on a network of up to 16 Amazon EC2 instances. To simulate realistic



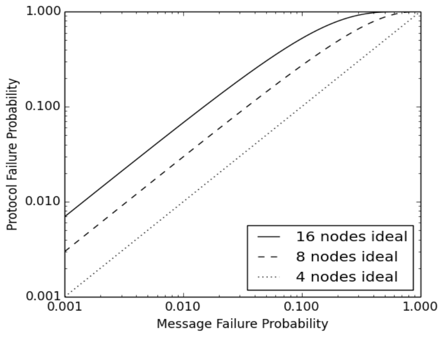


Figure 4: Penalties imposed on nodes vs. message failure probability.

Figure 5: Consistency failure vs. message failure probability (analytic only).

network delays, we used the Linux traffic control tool to limit bandwidth to 5mbps and impose a 100ms latency per message.

In the synchronous network model, messages between honest parties are simply guaranteed to be delivered within a given time bound. However, in reality, it is necessary to choose this timeout parameter concretely, based on estimates of network performance and on a tolerance for failure. Too short a timeout, and messages between otherwise-honest parties may fail to be delivered in time. In our experiments the payload for each broadcast is a constant size of 1 megabyte (i.e., the size of a Bitcoin block today). We benchmarked the network and computation load by performing several trial computations and measuring the resulting message delivery time, and then fitting a normal distribution to the results.

We first analyze the effects of message failure on the individual participants bottom line. If a node p fails to deliver a message to q in time, then q will inflict a punishment on p. Since each node is required to send 2 messages, if each message fails with probability ζ, we expect the expected cost of punishments to be $(N-1)(1-(1-\zeta)^2)$. In Figure 4 we compare the actual punishment incurred in our experiment with this expected line.

Message delivery failures can also lead to inconsistent outputs. In the worst case, if the maximum number of b nodes are actively attacking the network, then even a single failed message among the remaining nodes can lead to inconsistency. This occurs in the following scenario: suppose b nodes (including the leader) are corrupted, and send no messages at all until round b (the next-to-last round). At the beginning of round b, one of the corrupted nodes sends a message to a single honest node p containing a value v and b signatures. The node p will accept (and output) the value v, and relay it to the remaining $N-1-b$ honest nodes. If even one of these nodes fails to receive this final-round message, then it will output an inconsistent value $\perp$. Thus given b malicious nodes, and assuming messages fail independently with probability ζ, the uncorrupted nodes could output inconsistent values with probability $1-(1-N^2)$ probability (these are plotted in Figure 5).

4.1 Ethereum Smart Contract

We implemented the smart contract component of SmartCast in Ethereum's Solidity programming language. Our implementation includes:

- A smart contract for collecting reports, and handling payments. The entire program listing is shown in Figure 3.

- A smart contract implementing the "Marketplace" described in Section 3.5.
- A test framework using `pyethereum`, allowing us to measure the "gas costs" (i.e., transaction fees) for varying numbers of parties.

The Solidity language syntax resembles Javascript, and the intended effect of each line should be clear in context (though we imagine readers may be skeptical of the details, given several recent high-profile failures caused by subtle Solidity quirks [17, 9, 16]). Fortunately, the Smart Contract program listing in 3 fairly closely matches the pseudocode in Figure 1. We explain a few Solidity idioms that readers are likely to be unfamiliar with. Solidity supports "modifier" macros, which are convenient for specifying preconditions which must hold before a function is called (or else they `throw` an error). Furthermore, although the pseudocode disburses all rewards immediately upon the deadline, Ethereum does not directly support time-triggered events, thus the indirect `withdraw` function is necessary.

The Marketplace Contract. We also implemented a Solidity version of the "marketplace" smart contract described in Section 3.5. Below we describe its high level functions. For space, we omit the full Solidity code listing; the full code will be made available online.

- `registerTask`: creates a new task, configured with any application-specific parameters (e.g., description of a validation condition or a list of approved clients). The task creator must include payment sufficient to pay the workers for the task.
- `registerWorker`: allows a worker to sign up, depositing any necessary collateral.
- `finalize`: shuffles the list of workers and list of tasks, and then assigns workers to tasks until either a) no tasks are remaining, or b) not enough workers are available to fill the remaining task. For each fully-assigned task, spawn a new instance of the SmartCast contract. Return any deposited collateral to workers who were not assigned to a task, and refund payment to task creators whose tasks were not fulfilled.

Our protocol relies on a random beacon; our prototype simply uses `block.blockhash(0)` as a source of randomness, although this is known to be manipulable by miners [4, 19].

Ethereum Benchmarks. We tested our smart contract implementation using the `pyethereum.tester` framework. Table 1 shows the required gas costs for varying configurations of our application. We show results for only a few possible configurations: we increase the number of parties P, but always fill two tasks with two workers left over. The `finalize` method is the most expensive, since it grows with $O(N)$ when shuffling the list of workers. However, the `registerWorker` and `registerTask` methods are each invoked N times, and thus contribute about equally to the total.

Ethereum imposes a per-block (and hence, per-transaction) gas limit, which miners can vote to change gradually over time. Although the simulator easily supports these large transactions, today's Ethereum blockchain enforces a limit of approximately 2 million gas units, which the `finalize` operation busts when $P \geq 20$ (as underlined in Table 1). To avoid this limit, an alternative approach would be to spread the `finalize` operation over several contract invocations. This would require more complicated code, since each invocation would need to explicitly load and save its internal state. Our application provides a motivation for higher-level programming abstractions for transactions spanning multiple blocks.

Table 1: Smart contract gas costs (normalized to dollars, based on current Ethereum parameters and price (as of Nov 14 2016)). Underlined costs are infeasible, exceeding Ethereum's current per-block gas limit.

(N,P,T)	`registerWorker`		`registerTask`		`finalize`		Tot	
	Gas	(USD)	Gas	(USD)	Gas	(USD)	Gas	(USD)
$(4, 10, 2)$	110743	2.7¢	153347	3.8¢	1215702	30.4¢	2614826	65.4¢
$(8, 18, 2)$	110743	2.7¢	153347	3.8¢	1863111	46.6¢	4234665	\$1.05
$(16, 34, 2)$	110743	2.7¢	153347	3.8¢	<u>2966784</u>	74.0¢	6678740	\$1.70
$(32, 66, 2)$	110743	2.7¢	153347	3.8¢	<u>5271727</u>	\$1.32	12047459	\$3.01

Alternative implementation in Bitcoin. Our SmartBar protocol could still function using only Bitcoin's multi-signature transactions. The parties and the benefactor would generate N^2 transactions, where each transaction $T_{p,q}$ rewards θ to party q conditionally on a signature from p.

5 Conclusion and Future Work

We have adapted the work of Clement et al. [8] to the "smart contract" world, using cryptocurrencies to provide incentive compatibility for off-chain consensus protocols. Though we give a specific instantiation based on the Dolev-Strong protocol for reliable broadcast, our protocol is expressed as a generic transformation for arbitrary synchronous protocols.

Although the incentive compatibility notion we have adapted from Clement et al. [8] is described as "worst-case," modeling arbitrary Byzantine failures, many plausible attacks yet lie outside this model. In particular, our definition counterintuitively rules out "bribery" attacks, which are well-known though have not been observed in practice [3, 20]. Notice that the "worst-case" notion is from the point of view of an individual participant; since accepting a bribe makes an individual party richer, this is excluded by definition. Additionally, our utility model assumes unilateral deviation, which rules out collusion attacks. Incorporating both bribery and collusion into our model remains an important open problem.

References

[1] Iddo Bentov and Ranjit Kumaresan. How to use bitcoin to design fair protocols. In *International Cryptology Conference*, pages 421–439. Springer, 2014.

[2] Iddo Bentov and Ranjit Kumaresan. How to Use Bitcoin to Design Fair Protocols. In *CRYPTO*, 2014.

[3] Joseph Bonneau. Why buy when you can rent? bribery attacks on bitcoin consensus. *Bitcoin Research Workshop*, 2016.

[4] Joseph Bonneau, Jeremy Clark, and Steven Goldfeder. On bitcoin as a public randomness source. Cryptology ePrint Archive, Report 2015/1015, 2015. `http://eprint.iacr.org/2015/1015`.

[5] Joseph Bonneau, Andrew Miller, Jeremy Clark, Arvind Narayanan, Joshua A. Kroll, and Edward W. Felten. Research Perspectives and Challenges for Bitcoin and Cryptocurrencies. IEEE Symposium on Security and Privacy, 2015.

[6] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[7] Melissa Chase and Sarah Meiklejohn. Transparency overlays and applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 168–179. ACM, 2016.

[8] Allen Clement, Harry Li, Jeff Napper, Jean-Philippe Martin, Lorenzo Alvisi, and Mike Dahlin. Bar primer. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 287–296. IEEE, 2008.

[9] Kevin Delmolino, Mitchell Arnett, Ahmed E Kosba, Andrew Miller, and Elaine Shi. Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab. *Bitcoin Research Workshop*, 2016.

[10] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, DTIC Document, 2004.

[11] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[12] Juan Garay, Jonathan Katz, Ueli Maurer, Bjoern Tackmann, and Vassilis Zikas. Rational protocol design: Cryptography against incentive-driven adversaries. Cryptology ePrint Archive, Report 2013/496, 2013. `http://eprint.iacr.org/2013/496`.

[13] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 839–858, May 2016.

[14] Ranjit Kumaresan. Broadcast and verifiable secret sharing: New security models and round optimal constructions. 2012.

[15] Ben Laurie, Adam Langley, and E Kasper. Certificate transparency. *Network Working Group Internet-Draft, v12, work in progress. http://tools. ietf. org/html/draft-laurie-pki-sunlight-12*, 2013.

[16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.

[17] David Z. Morris. Blockchain-based venture capital fund hacked for $60 million. `http://fortune.com/2016/06/18/blockchain-vc-fund-hacked/`, June 2016.

[18] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `http://bitcoin.org/bitcoin.pdf`, 2008.

[19] Cecile Pierrot and Benjamin Wesolowski. Malleability of the blockchain's entropy. Cryptology ePrint Archive, Report 2016/370, 2016. `http://eprint.iacr.org/2016/370`.

[20] Jason Teutsch, Sanjay Jain, and Prateek Saxena. When cryptocurrencies mine their own business. *Bitcoin Research Workshop*, 2016.

[21] Gavin Wood. Ethereum: A secure decentralized transaction ledger. `http://gavwood.com/paper.pdf`, 2014.