IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Tunnelling Trust into the Blockchain: a Merkle Based Proof System for Structured Documents

FRANCESCO BRUSCHI[1], VINCENZO RANA[1], ALESSIO PAGANI[2], and DONATELLA SCIUTO.[1]

[1]Politecnico di Milano, Milano, 20133, Italy.
[1]The Alan Turing Institute, London, NW1 2DB, UK.

Corresponding author: Francesco Bruschi (e-mail: francesco.bruschi@polimi.it).

ABSTRACT The idea of Smart contracts foresees the possibility of automating contractual clauses using hardware and software tools and devices. One of the main perspectives of their implementation is the automation of interactions such as bets, collaterals, prediction markets, insurances. As blockchain platforms, such as Ethereum, offer very strong guarantees of untampered, deterministic execution, that can be exploited as smart contracts substrate, the problem of how to provide reliable information from the "outside world" into the contracts becomes central. In this paper, we propose a system based on a Merkle tree representation of structured documents (such as all XML), with which it is possible to generate compact proofs on the content of web documents. The proofs can then be efficiently checked on-chain by a smart contract, to trigger contract action. We provide an end-to-end proof of concept, applying it to real use case scenarios, which allows us to give an estimate of the costs.

INDEX TERMS Blockchain, Smart Contracts, Oracles, Merkle Trees, Ethereum

I. INTRODUCTION

SMART CONTRACTS were conceptually defined in 1997 by Nick Szabo [1]: "The basic idea of smart contracts is that many kinds of contractual clauses (such as liens, bonding, delineation of property rights, etc.) can be embedded in the hardware and software we deal with, in such a way as to make breach of contract expensive (if desired, sometimes prohibitively so) for the breacher". Szabo then went on using automatic vending machines as an actual example of smart contracts. The concept is of particular economical and social relevance, since it envisages the possibility of making agreements that do not rely on trusted third parties as monitoring and enforcing institutions, opening a huge realm of possibilities. In the vending machine example, fulfillment of the contractual clauses is guaranteed by the mechanics of the system, which can be described as a deterministic state machine (if a coin is inserted, wait for the code of the requested drink, etc).

Szabo's inception remained a theoretical idea until the advent of Bitcoin [2]. Bitcoin is a decentralized system that implements a digital form of cash: an open, permissionless platform for the exchange of digital assets. In Bitcoin, a ledger (blockchain) tracking the ownership of the digital assets (tokens) at any time is maintained and validated by an arbitrary number of potentially unidentified and egoistic actors. This distributed character makes the ledger, among other things, resistant to many forms of censorship. Bitcoin, in addition to this, introduces tools that makes its assets "programmable" through a scripting language. Examples of programmability include multisignature wallets, escrows, some kind of lotteries.

Even though Bitcoin introduces money programmability, its scripting language is purposefully limited below Turing completeness. For example, there are no loops, data structures are extremely limited, and the scripts are stateless. In response to the limitations of Bitcoin script, in 2013 Ethereum [3], an alternative system, inheriting much of Bitcoin technical characteristics, was defined, implemented, and deployed. The biggest difference of Ethereum is that it features a Turing-complete scripting language, with which it is possible to describe software processes of arbitrary complexity. These processes, remarkably, can handle the digital assets tracked by the Ethereum blockchain, and can send and receive tokens (that is, arrange change of token ownership), according to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3028498, IEEE Access

their programmed logic, and to the occurrence of certain conditions. These pieces of code are called smart contracts, since they can be a medium to implement Szabo's idea, with even stronger and more peculiar features. In fact, going back to the vending machine example, a question remains open: how can an observer be sure, just by watching or examining a vending machine, that it is properly programmed, and that it will not steal the coins? In the physical world of the vending machine, the observer should be able to assess, just by inspecting the machine, that 1) the software running on the microcontroller implements a fair behavior, 2) that the sensors will properly represent the world events (i.e., they will trigger the software with the correct signal when the user will insert the coins), and 3) that the actuators will properly execute the commands coming from the software (i.e., the robot arm will pick and deliver the drink when commanded by the software). In this context, Ethereum can guarantee that the agents will execute according to their code, which is publicly open for inspection. In this sense, they can be thought as a means for implementing Szabo smart contracts. What remains problematic are points 2) and 3). In particular, problem 2) can be generalised this way: how can information from the "outside world" be reliably conveyed to a smart contract? This issue is referred to as the "oracle problem".

A. THE ORACLE PROBLEM

The oracle problem, that is the need of representing external information to a smart contract, arises in different contexts, and with different requirements and constraints. This naturally induces different solutions to address it. Some examples are:

- A smart contract offers a bounty to anyone who will provide a solution to a mathematical problem (e.g., the solution to a given sudoku). How will the contract be able to check whether a given solution is correct?
- A smart contract defines and implements a kind of financial derivative, such as: A will pay B a premium if the value of an euro coin raises above 1.5 dollars within a month. How can the contract check the actual euro value?
- A smart contract implements a bet among users: will it rain, in Houston, within a week?
- A smart contract defines a car damage insurance: if the policy holder gets involved in an incident, the contract pays him the cost of repair. How can the contract check if the accident has actually happened? And how does it check repair costs?
- A smart contract will pay some subscribers if the there will be a Vulcanic eruption in Hawaii within one year, or..
- ..if Donald Trump will be impeached before the end of his term, or...
- ... if there will be be a diffusion of completely autonomous vehicles within year 2025.

The questions determining a smart contract behavior have different features and implications in terms of what it means to answer them. In a sense, they represent problems with different epistemological natures. For some it is easy to define an algorithmic checking procedure (e.g., ~the sudoku problem). For others, a procedure can be conceived if it is possible to rely on sources that are trusted and with a precise semantic structure. For instance, to check the price of the euro, it would be possible to access the Statistical Data and Metadata eXchange (SDMX) API endpoint offered by the European Central Bank at the URL [https://sdw-wsrest.ecb.europa.eu/service/data/EXR/M.USD.EUR.SP00.A] With respect to this example, the oracle problem takes the form of the question: how can a contract access external world APIs? Smart contracts execution must be deterministic and auditable in that it must be possible to verify, time after it happened, that a particular outcome is indeed correct and adhering to the behavior specified by the contract code. This implies that all the input to the contract must be recorded on-chain before its execution.

In the weather case, the algorithmic check definition can be more complex, since there can be no defined, ad-hoc API endpoint, even though the information is present from a trustworthy web source, such as https://www.accuweather.com/en/us/houston-tx/77002/weather-forecast/351197. In the damage insurance, the contract could rely on what some authorities affirm: an accident report signed by the police, a repair invoice by some accredited mechanic. These documents should be produced in a semantically parsable form to be automatically checked by a contract on chain. Another way could be to involve a trustworthy professional, such as a notary, to perform some sort of generalized "analog to digital" conversion, and produce machine readable representation of the "real world" information relevant for a contract. As for the volcano eruption, or for the impeachment, filtering news headlines with NLP techniques for keywords such as "eruption" could be envisioned. For the question on autonomous vehicles, on the other hand, even though its meaning could be sufficiently precise for a classic bet, an automatic check is more difficult to define. If autonomous vehicles will take hold, some precise market statistics will be available for everyone to see, but at the moment of the definition of the bet it is most probably unknown where such proof will be published and in which precise form.

Different approaches to the oracle problem are better suited for some of these situations than others. In the following, we review the current approaches, and discuss their fitness with respect to the situations described. Then, we will propose a novel solution to represent external information to on-chain smart contracts, and we will show some experiments to assess the method scope, expressivity, and costs.

In particular, we show how existing approaches in most cases introduce intermediaries, either centralized, or decentralized, with associated risks, costs, and complexities. To advance the state of the art, we propose a system that:

- is general, since it can be applied to most web contents,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3028498, IEEE Access

in the form of structured documents (any XML, thus including HTML);
- is flexible, since it can feed a smart contract with very specific information contained in a document or source of arbitrary size;
- requires no intermediary between the source and the smart contract;
- has a very low or no cost for data providers (since it can be transparently adopted as a server plugin);
- has a low cost for on chain verification.

To verify these properties, we check the process, and estimate its cost, with a full end-to-end proof of concept that spans from data proof generation to on-chain smart contract verification.

II. STATE OF THE ART

In this section, we describe and analyze the existing approaches to interface and connect smart contract execution with environments outside the blockchain to access external information.

A. PREVIOUS APPROACHES

1) Ad-hoc feeds

Ad-hoc feeds are implemented through smart contracts that are controlled by an entity that transfers information from the outside world, through transactions. For example, an oracle on the weather of Houston could be a smart contract that accepts updating transactions only by its owner, and forwards the information to the interested contracts on the chain. The oracle can implement a publish-subscribe pattern, and accepts subscriptions from the smart contracts that want to use its information. The controlling entity could be a single person, or a public or private institution. If a smart contract trusts the oracle, it can subscribe to its feed. Whenever the entity will enter new information, the oracle contract can transmit it, through a transaction, to the subscriber contracts. Or, the oracle could be passive, and just inform other contracts of its status when polled. This is the simplest way to inject information in the chain. Its main advantage is the low architectural cost. Its main drawbacks are:

- an oracle of this kind is very specific, and can provide only the information it was designed for (e.g., an oracle that feeds information on Houston weather will only do that). If new information is needed by a new smart contract, a new oracle has to be designed/instantiated.
- it requires trust in two distinct points of the chain: one is the source of the information (e.g., ~the weather web site), the other is the process/person that forwards the information into the blockchain.

As an example, consider MakerDAO's price feed. MakerDAO [4] is a platform that allows to generate DAI, tokens whose value is pegged to that of the US dollar, through some mechanisms encoded as smart contracts, and a decentralized governance defined by the possession of another kind of token, the MKR. In MakerDAO, users can ask for loans in DAI, and are required to secure providing a collateral denominated in another token such as ether, Ethereum's native token. For instance, a user can ask for 100$ worth of DAIs, providing 150$ worth of ethers. Whenever the user will pay his debt back (plus an interest rate), he will get the collateral back. Since ether value can vary (at the moment is quite volatile indeed), the system constantly checks if the collateral has enough value to guarantee the loan and, if it drops below a certain threshold, the system liquidates it, to prevent losses. It is clear how this requires the smart contract to know the price of ethers in USD at all times. In MakerDAO, the price is obtained through an ad-hoc oracle, composed by a set of authorised sources, that provide price feeds. The different feeds are then aggregated with certain policies (e.g., median computation). The system governance can update the authorized sources list.

2) TownCrier

TownCrier [5] proposes a system/architecture that allows on-chain smart contracts to require external web data, already accessible through TLS and HTTPS. TownCrier architecture is composed by:

- a smart contract that acts as front-end for the other on-chain client contracts;
- an enclave: a process running in a SGX (Software Guard Extensions) Intel environment;
- a relay: a process that handles communication of the enclave with the blockchain, the external resources, and provides attestations on the guarded execution of the enclave.

When a smart contract wants to obtain some information from a web resource, it contacts the TownCrier smart contract, using the APIs provided. The TownCrier smart contract encodes the request and emits it via some Ethereum events, that are monitored by the relay component, which in turn informs the enclave process, that generates HTTPS requests accordingly. In this way, the relay acts as a proxy between the enclave and the network. Upon reception of the response, the enclave checks the origin of the data, and provides a signed datagram in which it certifies the source of the requested information. The datagram is then forwarded from the relay to the TownCrier on-chain front-end, and from here to the original requesting smart contract. Some limits/problems of the approach are that there is a significant amount of centralization upon the TownCrier gateway (what if the relay or enclave are attacked, or shut down?). Moreover, even if it is possible to check the correct execution of the enclave off-chain, the attestation cannot be directly verified by the originating smart contract. An incorrect behavior of the enclave, due to an attack or other reasons, couldn't be noticed directly by the originating smart contract. If a smart contract wants to aggregate multiple sources, signature size and verification could significantly increase. Approaches like [6] allow to efficiently aggregate signatures coming from a variety of data sources.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3028498, IEEE Access

3) Oraclize

Oraclize employs an architecture similar to that of TownCrier, in which an external process is triggered by some smart contract events, accesses to some external data source, and communicates the result back to the requesting smart contract. In the case of Oraclize, the authenticity of the data source is proved by means of TLS-notary [7], a system with which, given a trusted auditor, it is possible to generate a proof that some data was indeed sent by a given source during a TSL session. In this case, Oraclize data retrieval process acts as auditee, and the auditor is an Amazon AWS instance, that guarantees and signs a proof that some traffic was indeed from the given source. This proof can be used by an external actor to verify the validity of the data forwarded by Oraclize to an on-chain smart contract. This Oraclize process offers interesting guarantees, but has some drawbacks: The TSL-notary proof cannot be checked directly by the smart contract, due to its complexity. If the data is tampered with, the fact will be evident to an external observer, but the smart contract will use the data as if it was valid. Moreover, the auditing process relies on the correct behavior of the AWS instance. Should the machine be compromised, it would be possible to forge correctness proofs. Moreover, Oraclize suffers the disadvantages of a centralized system: if it gets attacked, or controlled, it can hinder the functioning of the smart contracts that rely on it for external data access.

TownCrier and Oraclize can be categorized as request-response methods, since they allow smart contracts to specify which resources they want, and respond with the data requested.

4) Augur

Augur [8] is a decentralized platform for the creation and resolution of prediction markets. Prediction markets are environments in which participants can trade shares on the outcome of some prediction (e.g., will it rain tomorrow in New York?). They are qualified with "prediction" because the price of a share bound to an outcome is connected in a remarkable way to the probability of the outcome as estimated that the market actors. In particular, according to the Efficient Market Hypothesis [9], [10], the share price aggregates all the available information about the event outcome, and thus represents a crowdsourced estimation of its probability. Someone interested in evaluating the likelihood of an event could then create a prediction market selling shares of that event, let the actors trade them, and observe their price. Working prediction markets predate blockchains and smart contracts, and the implementation of the software platform has of course been an essential tool to set up and effectively run a prediction market ever since. Traditional platforms are centralized, and suffer various limitations: they require a trusted entity that handles a ledger of users shares, and that determines the actual outcome of the event. Prediction markets are a particularly well fitting application for smart contracts, and in fact different proposals, such as Augur, Gnosis and Stox seek to decentralize and automate their creation, share negotiation, and resolution. The trading part is a natural fit for smart contracts: keeping track of assets trading is what blockchains were born for. On the other hand, how do these systems deal with the problem of automatically establish the truth of a given outcome? Augur uses a system of incentives: after a market is due, that is when the time of the event it is predicting on is passed, it enters a so called "reporting" phase, in which a "designated reporter" can claim that a certain outcome is the true one, staking some value on it, in the form of reputation tokens (REP). After the initial claim, for a given period of time other users can "dispute" the initial claim, staking some value on alternative outcomes. If the disputing stake reaches a certain threshold, the outcome is considered disputed, the tentative outcome updated and another possibility for disputing opens for some time. This goes on until the tentative isn't successfully disputed within a time frame. If the disputed stake is greater than a given share of the existing REPs, a fork is triggered. A fork is an extreme procedure in which the whole platform in split into two so called universes, one in which the reported outcome is valid, and the other in which it is not. All the users are requested to choose in which universe to bring their tokens. After the fork is over, the two universes will be completely disjoint. Every new market will be either in one or in the other universe. The fundamental logic is that no one will want to open markets and trade in the universe in which the false report is considered valid, and thus the tokens there will have no value. These mechanisms strongly align reporters incentives towards claiming the truth.

This ingenuous clockwork has remarkable benefits, the most notable being that it can be applied to very general situations, wherever actors can agree on the truth of one sentence. This means that the question that defines the market can be simply formulated in human understandable language (in such a way that people can uniquely and easily answer). On the other hand, truth settlement can be very slow, from one to several weeks, to months in the case of a fork. Moreover, the incentives underlying the truth establishing mechanism are tightly coupled to the market structure, and is questionable whether they could be transposed as-is outside that application.

5) Kleros

Kleros is, like Augur, a decentralized oracle, structured as a decentralized autonomous organization. The system provides economic incentives to users that want to act as jurors in order to gain fees. Jurors are structured in courts, that have competence over different topics. The epistemic mechanism exploited is that of focal points, also known as Schelling points [11]: these are elements in a choice set that actors tend to choose when they have to coordinate in absence of reliable communication, because for some reason they perceive them as special, and think their relevance is common knowledge (i.e., they think that also the other actors will find the same element special, and all other actors will find it special etc.). The idea of Kleros is to construct a mechanism in which

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3028498, IEEE Access

jurors have to coordinate to provide a coherent answer to a question about the state of the world, and in which the actual state is the focal point. When a question is posed, candidate jurors have to stake some value through pinakions, tokens defined within the system. The higher the value staked, the higher will be the probability of being selected for the court. If they are selected, jurors inquire about the question considered, and then commit to their solution, publishing a hash of their answer, of a secret salt, and of their address (hash(salt+answer+address)). After having committed, they cannot change vote. If a juror A wants to convince B that he voted for the option x, he could show him the answer, the salt, and his address, and B could verify that A indeed voted x. This could enable coordination among actors, and then disturb convergence on the Schelling point. To prevent jurors from convincingly communicate their vote, the system allows this mechanism: if a juror B gets to know the salt and vote of the voter A, he can get hold of A's stake. After the committing phase, jurors reveal their votes. The tokens put at stake are redistributed to reward the jurors that voted the winning options, which is the one that received most votes. Kleros system could allow to make questions general and human understandable as is possible in Augur, but without the necessity to create a prediction market. It shares, with Augur, the high latency and low throughput due to the coordination of human actors.

6) Other approaches

Location based services could be conveniently implemented with smart contracts. In this case, the smart contract needs to gather information about position of users (for instance, in a ride sharing distributed application) or devices (e.g., in a supply chain control system). In [12], to prevent users from forging their positions, authors imagine to use an external provider, such as a mobile network operator, to transmit a position estimation based on network information to a smart contract. Authors then analyse different methods to conveniently encode information such as geofences for smart contracts. In [13], the authors propose a decentralized oracle system similar to that of Kleros, that additionally addresses the verifiers's dilemma [14], according to which, in some circumstances, actors of a distributed system could implicitly agree on a common value, such as constant 0, despite the actual answer to the question asked. The mechanism provides, with respect to Kleros, the introduction of certifiers, another class of actors. Certifiers stake high amounts of value to certify that propositions of their choosing are either true or false. In Astraea the authors claim that degenerate equilibria in which voters agree on a constant value disregarding the actual answer can be prevented tying the voting rewards to behaviors of the certifiers.

In [15], authors explore the possibility to use Merkle structures to guarantee integrity of Universal Description, Discovery and Integration (UDDI) web service registries. The approach introduces the possibility to prove the integrity of parts of an XML document with respect to hierarchical signature. Since the signature schemes are thought for off-chain applications, the the range of possible proofs considered is focused on proof of membership of single nodes. In [16] a system for the deploy of verifiable data feeds on a smart contract platform is presented. The approach is based on an architecture that requires data providers to deploy a special contract to which they cryptographically commit data updates. Actual data content is stored offline, at a given URL. The contract only allows for data appends, and offers an interface through witch other contracts can check the latest state of the feed. When a user wants to provide a client contract some information to trigger some clause execution, it obtains the data, computes the membership Merkle proof, and submits it to the relying contract, that can then check the proof against the updated signature value of the authoritative contract. The client contract will have to parse the information, and behave accordingly. Note that the system requires the data provider to actively interact with the blockchain, and that the relying contract must be able to parse a complete item of the feed. Approach presented in [16] is based on updatable structures that are also resistant to tampering. Precursors of such structures were conceived in the definition of digital notaries [17]–[19]. The solution proposed in [16] is based on Merkle trees and was presented in [20].

Analysis of the literature shows that there is no approach that allows, with the minimum possible effort, to make web information accessible to a smart contract, without the introduction of some kind of third party. There are either approaches such as [21], that are general and don't require specific effort on the side of data providers, or systems like [16], that require data providers to continuously interact with the chain, and do not fit well with web information, which is mostly provided in a format too heavy to be parsed by on-chain logic. This void raises a research question: is it possible to make web information available to smart contracts

- with minimal perturbation of the current content publishing processes,
- reliably (tamper proof),
- in a decentralized way (censorship proof),
- cheaply (with affordable on chain computation cost).

In the remainder of the paper, we will propose an architecture that aims at satisfying the requirements posed by the above question. The paper is structured as follows: in section III we introduce our solution step-wise, starting from simple assumptions, and then refining it as we find new problems and introduce elements of complexity. In section IV, we present the architectural form of our solution. In section V we present an experimental setup to functionally validate the solution and to estimate the cost involved. In section VI we present the results of the tests and of the cost evaluation. In section VII we draw some conclusions, and provide some perspective for future work.

III. PROPOSED SOLUTION

We propose an oracle data pattern that seeks the generality of request-response methods, without the complexity and trust

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3028498, IEEE Access

requirements of an intermediary like Oraclize. The method is based on a signature scheme that allows data providers to offer their information in a form suitable for the generation of proofs that can be fed to smart contracts. In what follows, we introduce the method, starting from a naive solution and refining it, step-wise, to address issues as they arise. We will assume that the data is provided in the form of a structured document, be it a XML document or a HTML page. To start, imagine that we want a smart contract to be able to check if the temperature in Milan was 30 degrees on a given day. This translates in checking if, in a page of a given trusted source (e.g., the Wall Street Journal weather page) there is a table with id reported-temperatures in which a row contains two adjacent cells, "Milan" and "30". To start, we could ask data providers that want to be used as reference by smart contracts to digitally sign their pages. Since most http endpoints already secure data through the SSL protocol, they could use the same private key to sign the data. This request is also the subject of an RFC document from the W3C Digital Verification Community Group (https://w3c-dvcg.github.io/http-signatures/). A smart contract could then be coded to accept a digitally signed page, verify the signature, and check the conditions embedded in the content.

The problem with this solution is that the operations of parsing and checking the page are computationally too expensive to be carried out on-chain. With Ethereum (but the circumstance applies to most smart contract platforms), on-chain code execution must be paid to reward the validators. In the execution cost model, every instruction has a cost expressed in a unit called gas. The amount of gas to execute a given piece of code, given a certain initial state, is deterministic. The cost of a gas unit in ether (the currency of Ethereum), instead, is subject to a market, in which users compete for the on-chain execution time, offering higher gas prices to validators, that choose the most convenient transactions first. Moreover, to prevent programs from spinning in infinite loops, limits are enforced on the gas that can be consumed for a single transaction. As a reference, simply loading a 10 kilobytes long document on Ethereum would cost, at the time of writing, the equivalent of 35$ in ether. After being loaded, the document signature should be verified, and the page should be parsed, operation that would likely exceed gas limits. The high cost of execution implies that code to be run on-chain should be minimized and restricted to those parts that require the auditability, censorship resistance, and guaranteed execution properties guaranteed by the blockchain platform, while everything else should be done off chain.

A possible way for making on-chain verification cheaper would imply using a signature that preserves some input structure: instead of signing the whole page, the data provider could sign the root of a Merkle tree of the page text, using words as elements. A Merkle tree is an hashing schema in which elements of a sequence are hashed and combined in a hierarchical way, obtaining a single hash called Merkle root. Merkle roots exhibit all the properties of “normal” hashes, and in addition allows to generate cryptographic proofs of the fact that a given element was present in the input sequence.

Using a Merkle tree, the contract could be coded to check a proof of some condition (say, in the text is written "In Milan it was above 30 degrees"). The issue that arises next is that representing the page with a flat structure limits expressiveness of the proofs that can be generated. For instance, how is it possible to express the condition that a table contains a row that contains two adjacent cells with data "Milan" and "30"?

The solution we propose is a procedure for generating Merkle roots of HTML documents that take structure into consideration, and allow the generation of proofs of the presence and relationship of text elements and their metadata, that can be efficiently checked by smart contracts on-chain. Using the scheme proposed, data providers can generate a compact signature that allows to generate proofs about the content of document, taking into account its structure.

The proposed signature can be exploited in smart contracts to trigger some behavior when a proof that a given document, coming from a given source (identified by its public key, for example), contains certain elements in a given relationship (e.g., "there is a row in the table with id temperatures in which the first cell contains the text "Milan" and the second the text "30"). Contractors can then check the condition and decide if they trust the source and how it is encoded. If and when the data provider produces a document that makes the condition true, the interested party can generate, using the signature, a proof of the condition occurrence, and then feed it to the smart contract, that in turn will trigger the predefined behavior. In this way most of the computational effort, that is spent parsing and generating the proof, is carried out off-chain, leaving on-chain only the critical parts in terms of trust, that is the checking of the condition proof. Moreover, no intermediary is needed. The party interested in proving the condition carries out the proof computation, and communicates it to the smart contract. If the proofs that can be generated are sufficiently expressive, data providers don’t need to change the structure of their content, and only need to add a signing plug-in to their http server, in a way that is completely transparent to the publishing process/flow.

In the following sections we detail the signing procedure, we show how proofs can be generated and checked, and we propose the experiments to evaluate off and on-chain costs for some examples.

IV. ARCHITECTURE

Although our approach initially targets HTML structured information, in order to propose an approach as general as possible, we represent HTML pages as XML trees and we propose a process to build an XML-Merkle tree. The idea, as presented in Section III, is to create a compact representation to generate proofs of the presence of some information in a signed document. To do that, we define a structure that integrates the information contained in the XML trees into a Merkle tree, enhancing the expressiveness of Merkle proofs while preserving a lightweight structure.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3028498, IEEE Access

An XML tree has its own set of nodes [22], [23], each with three main components:

1) a set of attributes (e.g., *id*, *href*), all optional except the field tag (e.g., <*div*>, <*figure*>);
2) an optional text field (e.g., *optional text*);
3) an optional list of child nodes (e.g., <*div*> *child div* <*div*>).

Hashing all the information in a node would introduce a computationally complex search operation on-chain and would also move part of the proof computation on-chain. On the other hand, directly generating a Merkle tree with each node by hashing child information in a bottom-up fashion would lose two components of their information. In order to avoid these drawbacks, in our approach we use Merkle trees to individually represent the three components of each XML node.

A. REPRESENTING THE TEXT IN A NODE

We propose to represent the text in a node as a Merkle tree. The granularity to use when mapping words into the Merkle tree is the first problem to be addressed. Naive solutions, like removing stop words or consider groups of words, could weaken expressive power. For this reason we propose to encode each word as a single leaf of the tree. In this work we use binary Merkle trees but, nevertheless, n-ary trees may also be used to build the text sub-tree of each node.

B. REPRESENTING THE ATTRIBUTES OF A NODE

Node attributes are a set of key-value pairs (e.g., $id = element1$, $color = red$). The only exception is the HTML tag (e.g., div, $figure$, ...), which is used to define the name of the field. The HTML tag is always the first word in the node. A common solution for representing node attributes are hash tables, but they are not a prime choice for storage in blockchain applications because of overhead costs [24].

We thus represent the attributes as a Merkle binary-tree: instead of storing each key-value pair as a single leaf, we alphabetically order the keys, and place each key-value pair in a leaf. Alphabetical ordering removes ambiguities from a node and allows to represent it in an deterministic way, while containing the computational cost of generating the tree and the evaluation proofs.

C. REPRESENTING THE CHILD NODES OF A NODE

Each child node can be represented as another XML-Merkle tree and consequently hashed. The child nodes of a node are thus represented as a list of root hashes (the root of each child node XML-Merkle tree). Nodes without child nodes are the first to be generated, while other nodes are afterward generated bottom-up.

D. THE ACTUAL NODE REPRESENTATION

As we just discussed, the three components of a node are:

1) the node attribute Merkle root;
2) the node text Merkle root;
3) the child nodes Merkle roots.

To simplify the representation of the node, we condense the first two components into a single hash, by hashing the concatenation of the two Merkle roots into a single element, which we can refer to as the *Attribute Text* (AT) hash. In the final representation each child node is represented as a hash of its content. The final structure of each node is the following:

1) the AT hash;
2) the child nodes hashes.

In this representation, each node is a list of $n + 1$ hashes, where the first hash is the AT hash, followed by the ordered child nodes hashes. The Merkle tree root of this representation is thus the hash of these $n + 1$ hashes concatenated.

E. EXAMPLE

Given the following HTML snippet of code:

```
<div id='main' class='text box'>
    <p>Hello world!<\p>
<\div>
```

The external node, the *div*, is converted into a Markle tree with three elements: the two attributes (*id* and *class*) and the tag (see Figure 1). Each attribute is hashed, the hashes are concatenated using a binary tree structure until the tree root is reached. The tree root is the "AT hash".

As previously discussed, the final node hash is generated concatenating the AT hash with the node hash of each child. In the proposed example, the only child of the node *div* is the node *p* (the one containing the text "Hello world!"). *p* does not have any child, thus its representation is the AT hash generated using the attribute pair ("tag", "p") and the text "Hello world!". The node *div* is finally generated concatenating the AT hash (with its tag and attributes) and the node hash of the child *p*. An example of the final representation for a generic node is shown in figure 2.

F. PROOF TYPES

For each node in the proposed solution, four different types of proofs can be performed: audit proofs, text proofs, attribute proofs and parental proofs. The first three proofs are also properties of the basic Merkle trees, while the parental proof is an additional property of the proposed XML-Merkle tree.

1) Audit Proofs

Audit proofs are used to verify if a record is in a tree. They consist in reconstructing the root hash from a subset of hashes in the tree. Their implementation in XML-Merkle tree is slightly more complex than the one used for binary trees. This is due to the different technique used to build the tree.

Audit proofs can be used to verify (I) the presence of a Merkle root of a sub-tree (text or attribute) in a node or (II) the presence of a node in the XML-Merkle tree.

The procedure used to demonstrate the first case (I) requires the verifier to check whether the AT hash can be

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3028498, IEEE Access

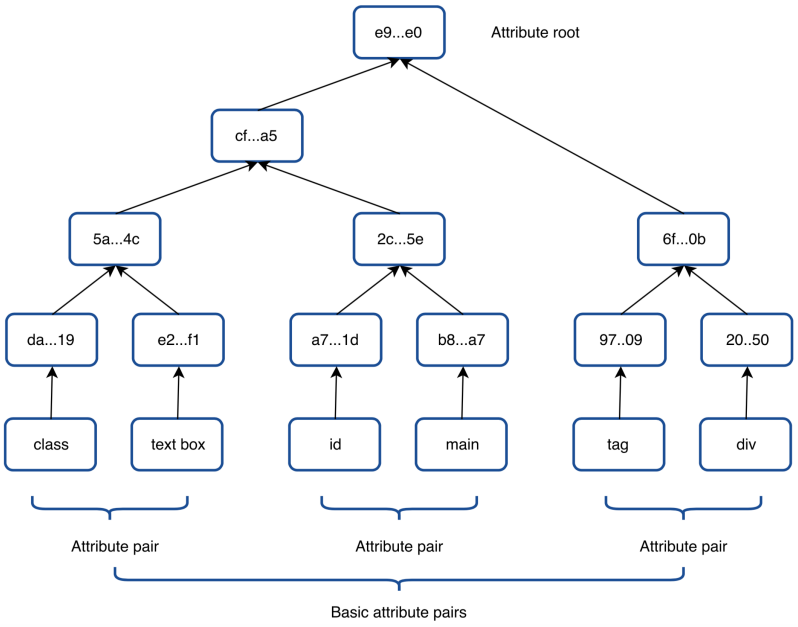

FIGURE 1. Example of HTML attribute tree.

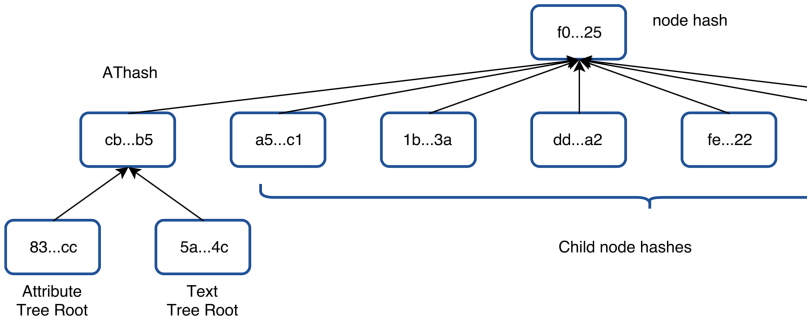

FIGURE 2. Example of HTML node.

obtained by hashing the first two elements, or to check that the concatenation of the computed AT hash with the other child nodes gives the node hash.

The inputs required are:

1) the computed Merkle root of the sub-tree - we are proving this element is in the tree;
2) the Merkle root of the other sub-trees;
3a) either node AT hash;
3b) or other child nodes hashes.

To validate the second scenario (II), the verifier has to check whether or not the root of the XML-Merkle tree previously signed matches the root of a reconstructed validation tree. To construct a validation tree the following information is needed:

1) node's hash - we are proving this element is in the tree;
2) an array of arrays of Merkle proof elements. Each sub-array is relative to the next parent node's child nodes. It contains the AT hash and $n-1$ child nodes hashes, where n is the number of child nodes;
3) an index array, listing at which index i each child is in relation to its parent;
4) the root of the XML-Merkle tree.

For each sub-array, the verifier computes the parent node hash by concatenating and hashing the first i nodes hashes, followed by the hash of the checked node, followed by the remaining $n-i$ child nodes' hashes. This procedure is recursively applied until the root of the tree is reached. The root is compared with the root of the original XML-Merkle tree and the proof is generated.

2) Text Proofs

In text proofs we want to prove that a text is in a given tree. The raw text is hashed to obtain a leaf value and the audit proof is calculated. The audit proof is the list of missing node hashes required to compute all of the nodes between the leaf

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3028498, IEEE Access

and the tree root. When the proof is required, it is enough to check if the root hash computed from the audit path matches the established Merkle root. If true, the leaf, and thus the given text, exists in the tree.

3) Attribute Proofs

A procedure similar to the one described for text proofs is performed for attribute proofs. Instead of hashing the raw text, the hashes of the attribute name and attribute value are computed separately and concatenated in a single value. The hash of this value is used as leaf value. Like for text proofs, a proof of the existence if an attribute can be produced by checking if the root hash computed from the audit path matches the established Merkle root.

4) Parental Proofs

Parental proofs can be used to verify parent-child relationships within the XML-Merkle tree. These proofs are very convenient for verifying specific properties of a node. In particular, they are especially useful to demonstrate key relationships between siblings which are necessary for some use-cases, for example to prove that two adjacent elements contain relevant information. As an example, in a scenario where we want to prove the outcome of a soccer match associated to a bet, the information is structured so that two adjacent elements contain respectively the team's name and its score. The easiest way to match a team's name and score is to demonstrate that a team's name is in a certain field and that there is a score in the adjacent field.

To perform this proof we need the following inputs:

1) the child node hash - the element we are demonstrating is the child of some node;
2) the child node index in the sibling set;
3) the parent node hash - the element we are demonstrating is the parent of some node;
4) the child node siblings;

To compute this proof, the siblings are concatenated according to their index: the ones with an index lower than the child index, followed by the child node hash, followed by the remaining siblings. The verifier checks whether or not the generated hash corresponds to the parent hash and generates the proof.

It is worth noting that proving that a node A is child of a node B, and then performing an audit proof of node B implicitly performs an audit proof of node A. However, the opposite is not true. Therefore, we cannot assert whether the parent node B is in the tree via an audit proof of node A.

In a similar way, to prove a sibling relationship between two nodes, it is sufficient to prove that both nodes are children of the same parent node. This can be extended with the use of indexes to prove ordinal relationships between siblings.

V. EXPERIMENTAL SETUP

In this section we propose two test scenarios to verify an end-to-end implementation of the proposed technique. Both cases are based on structured HTML web pages, with standard HTML tags, as a source of information. The tests were performed within a blockchain proof-of-concept prototype that is composed of:

- a document parser and Merkle signature generator, embedded in a plugin for an http server;
- a proof generator that accepts a document, the type of proofs, and computes the proof;
- a set of smart contracts that check a given proof. The contracts have been deployed on a test chain, to check the correct behavior and precisely assess cost figures.

The parser and signature generator, and the proof generator were coded in Python.

In our experiments, two actors participate in each contract:

1) the Maker: who gets the winnings in case of no valid proof presented, also known as the lose-by-default case;
2) the Taker: who gets the winnings after successfully constructing and executing the on-chain proofs.

To address the lose-by-default case, we use the following payout logic: the taker has to give a complete proof of the claim within a reasonable time frame. As a result, every contract has a maximum time frame by which the taker is expected to give a full proof. The contract specifies a timestamp by which the last proof must occur.

A. DESCRIPTION OF THE EXPERIMENTS

1) Scenario 1: News Headlines

This scenario revolves around guessing the headlines for a given day in the future. In our tests, we bet on the fact that a given word will appear in the news headlines of a predetermined newspaper on the following day.

In order to instantiate this contract, we need the following specific information:

1) a public key associated with the newspaper or publisher;
2) the date at which the word has to appear in the headlines;
3) the word that has to appear in the headlines.

Furthermore, we need to identify what qualifies as a title within the document, for example identifying the HTML tags and/or the classes that uniquely identify the title.

In this scenario there is no additional contract logic required, because the proof of a word being in a headline can be described by verification of attribute and text proofs.

2) Scenario 2: Temperature Prediction

In this scenario we want to determine whether or not a certain temperature will be reached on a given day in the future. To determine the temperature, we use a popular weather aggregator which reports the temperature for all major cities. In order to instantiate this contract, we need the following specific information:

1) a public key associated with the publisher;
2) the date by which the temperature has to be reached;
3) the name of the city where the temperature is being monitored;
4) the temperature threshold to be reached.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3028498, IEEE Access

Furthermore, we need an easy way to find the temperature for a given city in the page. Once identified the HTML elements and classes that contain the whether forecast, a proof for this scenario can be designed by:

1) initializing the contract with the name of the city and the target temperature;
2) proving that there is an HTML element with a text that contains the name of the selected city (attribute proofs, followed by an audit proof);
3) proving that there is an HTML element with a text that contains the temperature (attribute and text proofs followed by an audit proof). The smart contract compares the value of the temperature with the threshold defined during the contract initialization;
4) proving the sibling relationship between these two fields by providing that exists a parent field with these two nodes as children. This can be achieved by using parental proofs with fixed indices for the position of each child;
5) a final audit proof for the parent field, proving this subtree is in the page.

It is worth noting that, in this case, additional contract logic is required for comparing values to thresholds.

B. EXAMPLE

In this example, relative to scenario 2, we want to determine whether or not a certain temperature is reached in a city in Italy. To determine the temperature, we use a popular weather aggregator which reports the temperature for all major cities. The listing of the HTML page retrieved from the web is the following (listing 1):

```
<div class="odd2 row">
  <div class="col-xs-5 col-sm-8">
    <a href="http://meteo.corriere.it/
        italia/lazio/roma/" title="meteo
        Roma">Roma</a>
  </div>
  <div class="col-xs-3 col-sm-2">
    <div class="localita18
        l18-pioggia_30"/>
  </div>
  <div class="col-xs-4 col-sm-2">5</div>
</div>

<div class="odd1 row">
  <div class="col-xs-5 col-sm-8">
    <a href="http://meteo.corriere.it/
        italia/piemonte/torino/" title="
        meteo Torino" >Torino</a>
  </div>
  <div class="col-xs-3 col-sm-2">
    <div class="localita18
        l18-nuvoloso_70"/>
  </div>
  <div class="col-xs-4 col-sm-2">1</div>
</div>

<div class="odd2 row">
  <div class="col-xs-5 col-sm-8">
    <a href="http://meteo.corriere.it/
        italia/trentino-alto-adige/trento
        /" title="meteo Trento">Trento</a
        >
  </div>
  <div class="col-xs-3 col-sm-2">
    <div class="localita18
        l18-molto_nuvoloso"/>
  </div>
  <div class="col-xs-4 col-sm-2">3</div>
</div>
```

Listing 1. HTML snippet of a weather aggregator.

This snippet shows the weather for the cities of Rome, Turin and Trento. The temperature of each city has the following structure: a "div" element contains a list of "div" children elements, the first "div" in the list has an "a" element with a tag "title=meteo *city name*" where *city name* is the name of a city. Identified the city, the last sibling of this field's parent node contains the temperature in its text section. Given this structure, it is possible to build the relative Merkle-tree and write the smart contract that verifies where the searched city is located in the HTML structure (using attribute proof on the tag "title") and which is the temperature associated to that city (using parental proofs). Additional logic to verify if the collected temperature is above or below the threshold is written in the smart contract.

VI. RESULTS AND DISCUSSION

In this section, we first explain the evaluation methods used to generate the proofs. Next, we discuss the results of each scenario and some possible improvements.

A. EVALUATION METHOD

One of the main objectives of any smart contract component or technique is to minimize the amount of gas required for the computation, since it has a direct and relevant impact on the execution costs.

The experiments are carried out on a local test-net. Since code execution on Ethereum is strictly deterministic, running the code on a local chain or remote chain requires the same quantity of gas. Of course, gas price on the main network will depend on the actual, current gas price market.

B. SCENARIO 1: NEWS HEADLINES

Five contracts are generated, compiled, and deployed on the test blockchain, each one collecting the headline of a newspaper on a specific day. Proofs are generated to demonstrate the existence of specific words on the headlines. For each case, the following information is collected and analysed:

1) size of the document (expressed as number of lines);
2) contract initialization cost;
3) cost of each proving step;
4) intrinsic gas costs of each operation (represented alongside each operation).

Each contract requires a total of six proving steps (from initialization to verification) and an additional step to obtain payout. The steps are:

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3028498, IEEE Access

1) both parties join the contract;
2) compute a signed valid Merkle root;
3) compute node hashes for the title node and its parent node;
4) verify child node properties;
5) verify parent node properties and audit proof for the parent node;
6) parental proof for the parent and child node;
7) obtain payouts.

The parental proof and the audit proof can be completed out-of-order, due to the independence of the two proofs. If both are resolved our claims are verified.

The results show that the contract initialization cost for the five test cases is similar, possibly due to the nature of the words being checked in the contract. It is worth noting that the difference between the contract initialization costs is the same as the difference in the intrinsic costs of the other steps. This is due to the fact that the contracts have the same implementation, the only differences (and so the only parameters that change the cost) being the word searched for and the timestamp used to define time windows.

Analysing the single steps costs in detail (Table 1), we observe that Steps 1 to 3 have identical costs in all the test cases. This is due to the nature of these actions, which are identical except for the run-time parameters. Moreover, these steps run in constant time and thus their resource consumption is independent from the input. It is interesting to note that also in Step 4, used for proving properties of a child node through binary Merkle proofs, the costs are similar, with a maximum cost difference below 10%. Moreover, there is a correlation between the growth of the intrinsic cost and total cost: the computational cost (total cost) increases along with the size of the input (intrinsic cost), A different behaviour is observed in Steps 5 and 6. In these steps, we observe that binary proofs are inexpensive, in contrast to audit proofs that are the most expensive part of the proving process. We still observe a strict correlation between input size and computational cost. This is consistent with the observed high variance in intrinsic and total cost across test cases, which can be derived from the different structure of the documents used and the position of the element within the document, which significantly influences the cost of the audit proofs. This can be explained by considering the type of proof: parental proofs are the concatenation and hashing of a node with its siblings. Therefore, nodes with a small number of siblings will incur in lower parental proof costs than those with a high number of siblings, independently of the rest of the document structure. Finally, as for the first three steps, Step 7 has identical costs across all the scenarios.

A summary of the costs is reported in Table 2, the overall cost of each operation ranges from 2.1M to 2.5M gas. Values for initialization and proof costs are inclusive of intrinsic costs. The intrinsic cost column shows what portion of the overall gas expense is due to data storage costs.

C. SCENARIO 2: TEMPERATURE PREDICTION

We built three test contracts taking temperature readings on multiple days, and building proofs for different cities on those pages. For each case, we report:

1) Size of document (expressed as number of lines);
2) Contract initialization cost;
3) Cost of each proving step;

There are a total of seven steps, implemented as contract methods, between initialization and verification and another one to obtain payouts.

1) both parties joining the contract (overhead);
2) set a signed valid Merkle root;
3) set node hashes for temperature and city nodes and their parent node;
4) verify node properties for the city name;
5) verify node properties for the temperature field and check if the threshold is met;
6) parental proof for the parent and child nodes;
7) audit proof for the parent node.
8) obtain payouts.

Our results suggest that the contract initialization costs, like in the previous scenario, are similar for all the test cases. Also in this scenario, differences between contract initialization costs are identical to those between intrinsic costs of the other steps (the contracts are identical except the word being checked and the timestamps used to define time windows).

By analysing the steps costs in details (Table 3), we observe that the first steps have near-identical costs for each operation. The variation in intrinsic cost does not seem to affect the computational cost in these scenarios, where total costs and intrinsic costs increment exactly by the same amount. This can be due to the size of proofs being constant. This is particularly clear in Step 7, where the audit proof occurs. In the other scenario, audit proofs gave very different results across several cases. In this scenario, however, the cost is quite similar in all the tests.

A summary of the costs is reported in Table 4, the overall cost of each operation is around 2.8M gas. Values for initialization and proving costs are inclusive of intrinsic costs. The intrinsic cost column shows what portion of the overall gas expense is due to data storage costs.

D. COST ANALYSIS

At the time of writing, the gas price ranges from 1 Gwei (estimated execution time <30 min) to 5 Gwei (estimated execution time <2 min), with a standard price of 2 Gwei (estimated execution time <5 min). The price in fiat currency for a Gwei is around 0.00000014$ (1 Ether = 140$). Considering these values, the current cost for the execution of the proposed scenarios ranges from 0.29$ (slow execution, 2.1M gas) to 1.75$ (fast execution, 2.5M gas) for scenario 1 and from 0.392$ (slow execution, 2.8M gas) to 1.96$ (fast execution, 2.8M gas) for scenario 2. However, while the prices of gas and Ether are dictated by the market, and as

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3028498, IEEE Access

	Case 1		Case 2		Case 3		Case 4		Case 5	
	Total cost	Intr. cost	Total cost	Intr. cost	Total cost	Intr. cost	Total cost	Intr. cost	Total cost	Intr. cost
Init. cost	1 314 843	308 964	1 314 843	308 964	1 314 715	308 836	1 314 971	309 092	1 314 907	309 028
Step 1	42 534	21 272	42 534	21 272	42 534	21 272	42 534	21 272	42 534	21 272
Step 2	74 765	28 312	74 765	28 312	74 769	28 316	74 765	28 312	74 765	28 312
Step 3	71 940	256 243	71 940	256 243	71 940	256 243	71 940	256 243	71 940	25 624
Step 4	86 279	30 232	86 279	30 232	89 077	32 728	91 462	34 840	91 654	35 032
Step 5	800 221	232 536	849 334	245 528	456 359	139 736	623 476	184 728	759 906	221 528
Step 6	119 635	39 192	79 403	282 248	87 561	30 552	95 592	32 728	79 403	28 248
Step 7	21 023	21 272	21 023	21 272	21 023	21 272	21 023	21 272	21 023	21 272

TABLE 1. Scenario 1: total and intrinsic cost of each step.

	Doc. size	Init. cost	Proving cost	Intrinsic cost
Case 1	12 006	1 314 843	1 226 391	707 404
Case 2	12 006	1 314 843	1 235 278	709 452
Case 3	6 128	1 314 715	853 323	608 396
Case 4	18 536	1 314 971	1 030 792	657 868
Case 5	12 006	1 314 907	1 151 225	690 316

TABLE 2. Scenario 1: document size, initialization, proving and intrinsic cost of each case study.

a result outside of our control, the cost of the computation can be modified. Some solutions, hereafter presented, could be adopted to limit the computation cost of the proofs.

a: Reducing the proof size

In the proposed solution the number of proof elements grows exponentially with respect to the amount of sibling nodes, rather than linearly. Condensing the sequence of nodes into a single binary Merkle tree would result in faster and lighter proofs for parental and audit proofs. This process greatly increases the number of hashing operations to be done, but hashing operations are less expensive than the storage costs given by the intrinsic gas costs. In terms of actual costs, using a binary Merkle tree for node children could reduce the size of proofs up to 60% of the cost of audit proof transactions.

b: Reducing document size

Web pages contain much more information than strictly relevant to a user's query, examples of relevant although useless information include fields containing images and fields containing ads. While they may contribute to the web experience, they are useless for the proving purposes of our work. Two solutions could be adopted to reduce the size of the document and, consequently, the storage cost: one solution is to streamline the operations by filtering out all elements which may clutter the proving process. The second is to directly sign more interesting nodes in the tree in place of the Merkle root. This approach could cut around 10% of the audit proof costs for the former method, and remove the need for audit proofs completely with the latter one.

c: Reducing contract size

There are some high level optimizations that can be applied on our contract code to further reduce the amount of gas used. Examples include using assembly code in place of array access, as well as other micro optimization routines such as merging multiple proving steps into the same function to reduce the amount of variables saved between steps. This may lead to a contract initialization cost reduction between 5% and 15%.

d: Reusable contracts

Initializing a contract on the blockchain has very high costs even if the code is very optimized. Building a single reusable contract for many similar scenarios may have notable overhead costs when initialized for the first time and may add development complexity. However, after being initialized once, further instantiations of similar scenarios may require approximately half of the costs.

e: Adopting Ethereum scaling solutions

Scaling solution for the Ethereum network, like Plasma [25], sharding [26] and staking can be used to reduce the proof cost. Scaling solutions aim at increasing the throughput of the network, by allowing anywhere from 10 times to 1000 times more transactions per second. If we assume the same amount of transactions occurring as they are currently, a higher throughput causes less competition for inclusion in a block. This would lead in turn to lower gas prices. However, the network and block validators may react and introduce new measures to influence gas price. However, we can assume that, at least in their initial phases, scaling solutions should allow even lower priced transactions to be included.

E. SOLUTION IMPROVEMENTS AND MODIFICATIONS

This work is based on Merkle trees and Merkle proofs, as is standard in the Ethereum ecosystem. However, many other options exist when it comes to proving something. Verifiable computation approaches like zk-snarks [27] and proof-of-computation may be employed to even further reduce the computation cost.

Another improvement revolves around a generalization process. Currently, ad-hoc contracts are built for each scenario, manually defining proving steps and proof generation each time. A problem analogous to this one exists in the field of financial derivative contracts, where an ad-hoc contract is required for each case. To circumvent this issue, Biryukov et al. proposed FinDel [28], a system leveraging the Domain Specific Language (DSL) [29] to define any and all derivative

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3028498, IEEE Access

	Case 1		Case 2		Case 3	
	Total cost	Intr. cost	Total cost	Intr. cost	Total cost	Intr. cost
Init. cost	1 536 912	358 960	1 537 040	359 088	1 537 040	359 088
Step 1	42 578	21 272	42 578	21 272	42 578	21 272
Step 2	74 870	28 376	74 870	28 376	74 870	28 376
Step 3	94 125	27 800	94 125	27 800	94 061	27 736
Step 4	52 556	32 408	52 620	32 472	52 620	32 472
Step 5	54 731	25 752	54 731	25 752	54 731	25 752
Step 6	107 565	28 504	107 565	28 504	107 565	28 504
Step 7	821 752	162 776	821 732	162 776	821 820	162 840
Step 8	31 067	21 272	31 067	21 272	31 067	21 272

TABLE 3. Scenario 2: total and intrinsic cost of each step.

	Doc. size	Init. cost	Proving cost	Intrinsic cost
Case 1	705	1 536 912	707 120	1 279 244
Case 2	705	1 537 040	707 312	1 279 288
Case 3	705	1 537 040	707 312	1 279 312

TABLE 4. Scenario 2: document size, initialization, proving and intrinsic cost of each case study.

contracts using a single language developed specifically for this purpose. Analogously, one could work toward developing a DSL to define proofs using XML-merkle trees, by formalizing different predicates to be used to represent different properties of nodes in a tree, and using this new DSL in place of ad-hoc contract to make proofs on chain. Pairing this with one or a few executor contracts, as those proposed earlier, may remove the need for contract initialization costs completely.

VII. CONCLUSIONS AND FUTURE WORK

The current state of the Ethereum ecosystem is at a point where it can scale to support the development of applications that may handle sensitive and complex operations with strong execution guarantees.

In this work, borrowing the Merkle tree design pattern which is present in many blockchain applications and by using digital signature, we aimed to supplement the Ethereum platform with a proposal to authenticate and verify properties of web information relayed by external agents (e.g., oracles) to a smart contract without requiring trust in other intermediaries.

This allows the design of smart contracts in which predefined reactions to a certain external world event are guaranteed to be executed (e.g., the payment of one of the actors of a bet when the result is known).

In general, the solution provides the possibility to maintain the same level of guarantee that decentralized execution strives for.

By requiring publishers to sign the information, there are no further trusted actors to be included in the pipeline contrary to current state of the art solutions, and any client can relay information to smart contracts from public sources.

As application examples we can imagine flight insurances, where verifying arrival times of a flight on a web page is necessary, as well as financial derivative settlements, where public web pages detailing price indexes of a stock ticker need to be accessed.

Furthermore, the presented flow can be easily integrated in existing websites with minimal effort for a publisher, requiring at most a simple plugin, which may be streamlined and customized to the publisher needs.

One direction for further work could be to explore alternative proving methods. Verifiable computation approaches like zk-snarks [22] and proof-of-computation may be employed to reduce the cost of computation. One could work also toward developing a DSL (Domain Specific Language) to define proofs and use it in place of ad-hoc contract to make proofs on chain. Moreover, one can imagine the emergence of specialized third parties that systematically provide signatures of pages (imagine a search engine or a version of the service archive.org [30] that allows to query the signature of a certain version of a page). Even if this would add another trusted actor to the system, signing the digest of a page, as opposed to relaying some specific information directly, would allow for a smaller tolerance for tampering and dishonest behavior, where making false claims becomes more difficult than simply relaying wrong information.

REFERENCES

[1] N. Szabo, "Formalizing and securing relationships on public networks," First Monday, vol. 2, 01 1997.

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Cryptography Mailing list at https://metzdowd.com, 03 2009.

[3] V. Buterin, "A next-generation smart contract and decentralized application platform," white paper, 2014, https://github.com/ethereum/wiki/wiki/White-Paper#decentralized-autonomous-organizations. [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper#decentralized-autonomous-organizations

[4] M. Team, "The dai stablecoin system," URl: https://makerdao.com/whitepaper/DaiDec17WP. pdf, 2017.

[5] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '16. New York, NY, USA: ACM, 2016, pp. 270–282. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978326

[6] B. van der Laan, O. Ersoy, and Z. Erkin, "Muscle: Authenticated external data retrieval from multiple sources for smart contracts," in Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, ser. SAC '19. New York, NY, USA: ACM, 2019, pp. 382–391. [Online]. Available: http://doi.acm.org/10.1145/3297280.3297320

[7] "Tlsnotary - a mechanism for independently audited https sessions," https://tlsnotary.org/TLSNotary.pdf, 2014.

[8] J. Peterson, J. Krug, M. Zoltu, A. K. Williams, and S. Alexander, "Augur: a decentralized oracle and prediction market platform," url: https://www.augur.net/whitepaper.pdf, 2018.
[9] P. A. Samuelson, "Rational theory of warrant pricing," Industrial Management Review, vol. 6, p. 13-39, 1965.
[10] E. Fama, "The behavior of stock-market prices," Journal of Business, vol. 38, p. 34-105, 1965.
[11] T. C. Schelling, The strategy of conflict (First ed.). Cambridge: Harvard University Press, 1960.
[12] F. Victor and S. Zickau, "Geofences on the blockchain: Enabling decentralized location-based services," in 2018 IEEE International Conference on Data Mining Workshops (ICDMW), Nov 2018, pp. 97–104.
[13] J. Adler, R. Berryhill, A. G. Veneris, Z. Poulos, N. Veira, and A. Kastania, "Astraea: A decentralized blockchain oracle," CoRR, vol. abs/1808.00528, 2018. [Online]. Available: http://arxiv.org/abs/1808.00528
[14] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena, "Demystifying incentives in the consensus computer," in Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '15. New York, NY, USA: ACM, 2015, pp. 706–719. [Online]. Available: http://doi.acm.org/10.1145/2810103.2813659
[15] E. Bertino, L. Martino, F. Paci, and A. C. Squicciarini, Security for Web Services and Service-Oriented Architectures. Springer, 2010. [Online]. Available: https://doi.org/10.1007/978-3-540-87742-4
[16] J. Guarnizo and P. Szalachowski, "PDFS: practical data feed service for smart contracts," in Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I, ser. Lecture Notes in Computer Science, K. Sako, S. Schneider, and P. Y. A. Ryan, Eds., vol. 11735. Springer, 2019, pp. 767–789. [Online]. Available: https://doi.org/10.1007/978-3-030-29959-0_37
[17] D. Bayer, S. Haber, and W. S. Stornetta, "Improving the efficiency and reliability of digital time-stamping," in Sequences II, R. Capocelli, A. De Santis, and U. Vaccaro, Eds. New York, NY: Springer New York, 1993, pp. 329–334.
[18] M. T. Goodrich, D. Nguyen, O. Ohrimenko, C. Papamanthou, R. Tamassia, N. Triandopoulos, and C. V. Lopes, "Efficient verification of web-content searching through authenticated web crawlers," Proc. VLDB Endow., vol. 5, no. 10, pp. 920–931, 2012. [Online]. Available: http://vldb.org/pvldb/vol5/p920_michaeltgoodrich_vldb2012.pdf
[19] S. Haber and W. S. Stornetta, "How to time-stamp a digital document," in Advances in Cryptology-CRYPTO' 90, A. J. Menezes and S. A. Vanstone, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 437–455.
[20] S. A. Crosby and D. S. Wallach, "Efficient data structures for tamper-evident logging," in Proceedings of the 18th Conference on USENIX Security Symposium, ser. SSYM'09. USA: USENIX Association, 2009, p. 317–334.
[21] T. Bertani, "A scalable architecture for on-demand, untrusted delivery of entropy." [Online]. Available: http://www.oraclize.it/papers/random_datasource-rev1.pdf
[22] W3C2019. (2019) Extensible markup language (xml) 1.0 (fifth edition). [Online]. Available: https://www.w3.org/TR/xml/REC-xml-20081126-review.html#sec-terminology
[23] W3C2014. (2014) Xml information set (second edition). [Online]. Available: http://www.w3.org/TR/xml-infoset/
[24] S. Asharaf and S. Adarsh, Decentralized Computing using Blockchain Technologies and Smart Contracts. IGI Global, 2017.
[25] J. S. Y. Poon, "Plasma: Scalable autonomous smart contracts," 2017.
[26] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena, "A secure sharding protocol for open blockchains," in Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, ser. CCS '16. ACM, 2016, pp. 17–30. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978389
[27] V. Dhillon, D. Metcalf, and M. Hooper, Recent Developments in Blockchain. Blockchain Enabled Applications, 2017, pp. 151–181.
[28] A. Biryukov, D. Khovratovich, and S. Tikhomirov, "Findel: Secure derivative contracts for ethereum," in Financial Cryptography and Data Security, M. Brenner, K. Rohloff, J. Bonneau, A. Miller, P. Y. Ryan, V. Teague, A. Bracciali, M. Sala, F. Pintore, and M. Jakobsson, Eds. Springer International Publishing, 2017, pp. 453–467.
[29] S. Peyton Jones, J.-M. Eber, and J. Seward, "Composing contracts: An adventure in financial engineering (functional pearl)," in Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ser. ICFP '00. New York, NY, USA: ACM, 2000, pp. 280–292. [Online]. Available: http://doi.acm.org/10.1145/351240.351267
[30] "The internet archive," https://archive.org, 2014.