Blockchain-Based, Confidentiality-Preserving Orchestration of Collaborative Workflows

Balázs Ádám Toldi, Imre Kocsis
Dept. of Measurement and Information Systems
Budapest University of Technology and Economics
Budapest, Hungary
balazs.toldi@edu.bme.hu, kocsis.imre@vik.bme.hu

***Abstract*—Business process collaboration between independent parties can be challenging, especially if the participants do not have complete trust in each other. Tracking actions and enforcing the activity authorizations of participants via blockchain-hosted smart contracts is an emerging solution to this lack of trust, with most state-of-the-art approaches generating the orchestrating smart contract logic from BPMN models. However, as a significant drawback in comparison to centralized business process orchestration, smart contract state typically leaks potentially sensitive information about the state of the collaboration.**

We describe a novel approach where the process manager smart contract only stores cryptographic commitments to the state and checks zero-knowledge proofs on update proposals. We cover a representative subset of BPMN, support message passing commitments between participants and provide an open-source end-to-end implementation. Under our approach, no party external to the collaboration can gain trustable knowledge of the current state of a process instance (barring collusion with a participant), even if it has full access to the blockchain history.

***Index Terms*—blockchain, BPMN, zero-knowledge proof, collaboration**

I. Introduction

In modern business science, *Business Process Management* (BPM) as a discipline [1] advocates process-focused thinking about internal activities and external collaborations and proved to be a very important tool for controlling and improving key performance indicators. Automating the execution of business processes is a key proposition of BPM and has been supported for a long time by a range of Workflow Engines, and Workflow Enactment Services [2]. Today most of these, typically centralized, tools use the leading business process modelling standard, BPMN 2.0 [3], as a process definition language [4].

Distributed ledger technology, generally implemented on a blockchain basis, is widely recognized as a compelling platform to support the cross-organisational execution of business processes – even when the organisations can not agree on a trusted (third) party as a middleman [5]. Importantly, smart contracts can be used to enforce and track the sequences of activities performed by organizations participating in collaborations; store sent and received messages; and either host shared data objects directly or anchor their changes in the blockchain via cryptographic commitments. However, blockchain-assisted BPM is still a relatively new discipline – among other challenges, the privacy and confidentiality aspects have not yet been sufficiently addressed.

In our paper, we present a novel approach[1] for orchestrating cross-organizational workflows – collaborations – with smart contracts in a confidentiality preserving way. The process logic is defined by BPMN models, and parties not participating in a process instance can not determine the state of the instance, even if they have full access to the transaction sequence of the underlying blockchain and complete knowledge of the process model.

Specifically, we encode the state updates of BPMN process instances as programs of the ZoKrates [6] toolkit, from which zero-knowledge proofs and state commitment update verifier smart contracts are generated. We also define an accompanying state commitment update protocol. We describe our open-source end-to-end framework implementation prototype[2] and provide an empirical demonstration of the practical viability of the presented approach.

To the best of our knowledge, our work is the first to offer robust confidentiality protection for blockchain-orchestrated, BPMN-specified collaborations.

II. Blockchain-based process orchestration

On most public and permissionless blockchains that allow smart contract deployment, as a rule, the developer cannot update the contract after it is deployed. Accompanied by the risks associated with data integrity errors in a blockchain-based distributed ledger which we wish to treat as a ”single source of truth”, minimizing the probability of smart contract faults is a major concern in the whole industry. Consequently, in addition to domain-specific languages, Model-Driven Engineering (MDE) techniques are steadily gaining ground in smart contract development [7]. In our context, this means that the dominant approach is a model – usually formulated in BPMN – to serve as a *specification*, and smart contract logic is generated automatically from the model.

Caterpillar [8] was the first open-source BPMN-to-Solidity compiler (Solidity is the primary smart contract development language for the Ethereum platform). Since its initial release, several forks have emerged. Some of these also come with an

[1]This paper is based on the Scientific Student Association report submitted by Balázs Ádám Toldi to the 2022 competition at the Budapest University of Technology and Economics, advised by Imre Kocsis: https://tdk.bme.hu/VIK/sw8/Kollaborativ-munkafolyamatok-titkossagmegorzo

[2]Available at https://github.com/ftsrg/zkWF

extended feature set, like Blockchain Studio [9], which adds role management, or [10], which adds time constraints.

Lorikeet [11] is a model-driven engineering approach that integrates assets into business processes. Lorikeet extends the BPMN 2.0 specification with support for asset registries and also transforms models into Solidity smart contracts. The smart contracts handle the orchestration of the process as well as interactions with the tokens.

Chorchain [12] is a tool that takes a BPMN *choreography* and generates an Ethereum smart contract that can be used to execute the model. ChorChain also includes a dedicated modelling tool. The same authors also released two additional but related tools: Multi-Chain [13] and FlexChain [14]. Multi-chain is similar to Chorchain, but it is also capable of generating smart contracts for Hyperledger Fabric [15]. FlexChain can only produce Solidity smart contracts, but the user can also define a ruleset for each choreography. If a condition in the ruleset is met, then an off-chain processor will perform its underlying action.

Our analysis showed that the process state, as well as the process trace, are easily recoverable from the process manager smart contracts for *all* the tools above.

The Baseline protocol[3] is a developing open standard that allows enterprises to synchronize complex, multi-party business processes on distributed ledger technologies. Business process workflows in Baseline are formed as state machines. The standard includes some essential and optional privacy-related requirements. The protocol has two reference implementations; however, at the time of this writing, neither of these actually supports privacy/confidentiality measures.

III. A CONFIDENTIALITY-PRESERVING APPROACH

In this paper, we present an approach for tracking and orchestrating business processes through smart contracts, conceptually in a very similar way to [8],[12],[13],[14],[11],[9], and [10], but instead of storing the process state on-chain, we apply cryptographic state commitments and accept new commitments carrying state commitment update proposals on the presentation of proper Zero-Knowledge Proofs (ZKPs). Our approach aims at rendering process state and data information undiscoverable from the smart contract state – to parties not involved in the execution of the process instance.

ZKPs are cryptographic methods to prove that various statements are true – without revealing any additional information about the statement. Quoting the ZKProof Community Reference ([16], p1), *"A zero-knowledge proof makes it possible to prove a statement is true while preserving the confidentiality of secret information. This makes sense when the veracity of the statement is not obvious on its own, but the prover knows relevant secret information (or has a skill, like super computation ability) that enables producing proofs. The notion of secrecy is used here in the sense of prohibited leakage, but a ZKP makes sense even if the 'secret' (or any portion of it) is known apriori by the verifier(s)"*.

In this work, we rely on zk-SNARKs, a family of *noninteractive* ("single-shot" message passing from prover to verifier), and *succinct* (small and cheaply verifiable proofs) ZKPs. We use the ZoKrates toolkit as a ZKP front-end [6], which currently supports the Groth16 [17] and GM17 [18] schemes.

[3]https://docs.baseline-protocol.org/

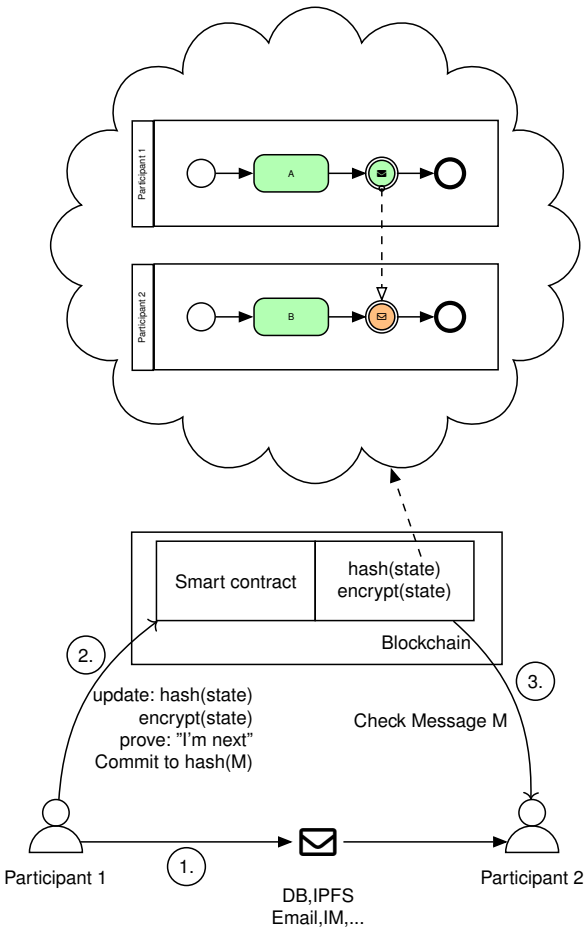

Fig. 1. Overview of the zkWF protocol

A. The zero-knowledge WorkFlow (zkWF) protocol

Our approach relies on two key conceptual components: our zkWF ("zero knowledge WorkFlow") protocol and what we call "zkWF programs".

> The **zkWF protocol** is a hash commitment style protocol that allows the participants of a business process to follow and step the execution of a business process, governed by a smart contract. Meanwhile, the state and trace of the process execution remain hidden from external parties.

Figure 1 presents a high-level overview of the protocol. The protocol requires to have a smart contract deployed on a blockchain for each process instance. This smart contract contains a hash commitment of the current state and an encrypted version of the state.

During process execution, the collaborating parties can send messages to each other by off-chain means. These are captured in the underlying process specification as intermediate message throw and capture events; the commitment scheme includes commitments to the message hashes.

When a participant wishes to update the state stored in the smart contract (hash commitment and state ciphertext) – that is, to "step the process" –, it has to create a ZKP that the state transition they propose is valid. This new state includes the hash of the message they sent beforehand if the step involves message sending. When the execution arrives at a point where a participant receives a message in the next stage of the execution, the receiving party checks the hash and only accepts if the hashes match.

Participant authorization is tied to proving EdDSA private key ownership in the ZKPs; the public keys of the participants are defined over the underlying process model as a parameterization. Additionally, we require the participants to have a common means for encrypting and decrypting stored state ciphertexts (this aspect is not constrained by the protocol).

This scheme enables adapting the protocol to different distributed ledgers in a straightforward way (we provide an implementation for Ethereum and Hyperledger Fabric); as well as masking updater identity on pseudonymizing platforms (such as Ethereum) by facilitating the use of single-use transaction source addresses. While the updates themselves and the contract state are unintelligible to parties outside the collaboration, statistical and model trace analyses of the update sequences may still be a threat; our mitigation is the inclusion of a "fake" update transaction variant (no actual state update), which all participants are authorized to use freely.

B. zkWF programs

zkWF programs serve as a bridge between process specification and proof computation/verification. These programs are generated from a representative subset of the BPMN specification, with extensions for cryptographic checks, as detailed later.

A **zkWF program** is a ZoKrates program that, for a given BPMN model instance (parameterized model), can decide whether a given actor is authorized to execute a state transition in a given execution state. The ZoKrates program can be used to generate the zero-knowledge proofs and the proof verification code used by the **zkWF protocol** participants and zkWF smart contracts.

C. Toolchain overview

We have created a prototype of an end-to-end toolchain to support the proposed approach, as depicted in figure 2, from modelling through code synthesis to deployment and operation. The figure also delineates the newly created software components (and those with novel generators).

In the *modelling phase*, a BPMN model is annotated with metadata for process instantiation, and our Kotlin-based interpreter and translator creates the corresponding zkWF program.

In the *synthesis phase*, the ZoKrates toolkit is used to set up the *prover key* and *verifier key* (note that ZoKrates supports *multi-party ceremonies*) and generates the verifier smart contract in Solidity. We created novel support for generating verifier code for Hyperledger Fabric [15]. We also created the (necessarily application-specific) code generation facilities for

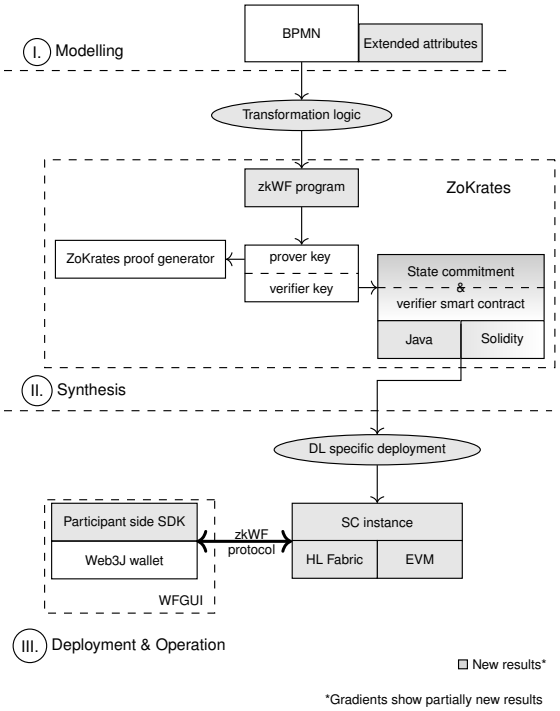

Fig. 2. Toolchain overview

the state commitment management part of the smart contracts for both platforms.

For the deployment phase, we created automation facilities for deployment to Ethereum (and other blockchains using a compatible RPC API); and an SDK and GUI application for the client side. Here we integrate the ZoKrates toolkit as a proof generator.

D. Adversarial model and security goals

A *participant* of a *business process instance* is a party who has a private EdDSA key necessary to enact certain state changes in the process instance, as encoded in the underlying BPMN model instance with public key annotations on model elements. All other parties are deemed *process external*.

We assume that the underlying process model is public knowledge, but the set of public keys serving as parameterization for a process instance is shared only between the participants. We also assume no private key compromises.

We assume full integrity for the blockchain (no successful attack on the consensus) but also the full observability of transactions targeting the process manager smart contract and the smart contract state sequence by process external parties. For the sake of simplicity, we assume deterministic transaction finality for the blockchain (largely equivalent to "waiting for a few blocks" for treating transactions as blockchain-included under probabilistic finality models). We treat the blockchain as *fair* – any transaction submitted by a participant is included in a block in a reasonable time, irrespective of concurrent transaction proposal load.

Our *integrity goals* are the following.

- Process external parties should not be able to influence the smart contract state commitment and stored encrypted state.
- Participants should be able to influence the smart contract state commitment and stored encrypted state only when they are authorized by their private key(s). Accepted state commitment updates should always conform to the execution semantics of the underlying process model.
- Should a participant successfully submit an encrypted state not conforming to the accompanying commitment, it has to be evident to all participants in a non-repudiable way.

The first two integrity goals largely carry over from the earlier cited state of the art; the third is a reasonable relaxation in view of a currently still missing ZKP capability, as explained later.

Our *availability goal* is that no process external party should be able to influence the update capability of the authorized participants.

Lastly, our *confidentiality goal* is that no external party should be able to determine the process state either fully or partially (beyond the fact that it exists and that it has been started) without collusion with at least one participant.

IV. BPMN SUBSET AND EXECUTION SEMANTICS

In this paper, our main focus is on BPMN *collaboration* models. According to the specification, they can contain *processes* or *choreographies*; we work with *collaborative processes*.

Currently, we support a limited but representative set of elements from the BPMN specification, as summarized by table I. Notably, in addition to the Basic Modeling Elements of BPMN 2.0 (see [3], p28), we also support message throw and catch events, which are of particular importance in collaborative settings. The table denotes those elements as "stateful" which have non-instantaneous execution semantics (as declared by the BPMN specification), and these will determine the structure of our execution state vector. In the context of this paper, we will refer to these elements as the "executable" ones in the BPMN subset we address.

TABLE I
SUPPORTED BPMN MODELLING ELEMENTS

Element name	Notation	Stateful?
Start event		no
End Event		no
Activity	Task	yes
Sequence flow		no
Message flow		no
Parallel gateway	+	no
Exclusive gateway	×	no
Message		no
Message Intermediate Catch event (executable)		yes
Message Intermediate Throw event (executable)		yes
Pool	Party1	no
Lane	Party1 A B	no

A. BPMN extensions and structural constraints

In order to capture properties that are necessary for our zero-knowledge approach, we added two types of extended attributes on top of the existing BPMN specification. On the one hand, a `zkp:publicKey` attribute is used to separate the tasks of different participants by attaching an EdDSA public key to a pool, a lane, or an executable element. Applying this attribute to an element directly or indirectly (e.g. through inclusion in a pool) is mandatory; however, there are no facilities for key overriding in the model hierarchy yet. The intended usage is to equip either pools or lanes with public keys.

`zkp:variables` extended attributes can be applied to *activities*. These declare process instance global variables and that the variable may be written by that activity (reads are allowed for all activities). These variables can be used in expressions for exclusive gateways. The gateways support boolean expressions over these global variables.

Some constraints apply to the structure of the BPMN models currently admissible in our scheme.

- We support binary gateways (at most two incoming/outgoing edges).
- Activities are *atomic*; i.e., subprocesses are not supported.
- The model cannot contain loops; sequence and message flows must form a directed acyclic graph.

We plan to eliminate these constraints in the future; the required modifications of the state representation and the zkWF program construction are largely incremental.

B. State representation

Our notion of process instance execution state encompasses the following aspects (for the specific encoding in zkWF programs, please refer to the implementation).

- A vector v of the current state of executable elements
- The current values of *global variables*
- Hashes of the messages already sent in the process

Let us represent a business process M as a tuple (V, E, T), where V is the set of non-flow model elements, E is the set of model edges (flows), and $T \subset V$ is the set of all executable

elements in the business process. Then, v is a vector of $|T|$ size and $\forall v_i \in v$ can have one of the following three values:

- 0 (Inactive) – The element has not been reached yet
- 1 (Active) – The element is ready to be executed or is being executed by a party
- 2 (Completed) – The execution of the element has been completed

Note that this state space model is a simplification, especially in terms of the full BPMN activity lifecycle; however, it is a reasonable simplification in the sense that it is of sufficient expressive power for important applications in our context (as we show later). Further research will investigate incorporating the full lifecycle model.

C. Capturing token passing semantics

As described by the standard, BPMN 2.0 models have token flow-based execution semantics. For the purposes of supporting a different ZKP use case, Aivo et al. [19] introduce a technique for representing valid BPMN execution state changes through enumerating the possible composite token marking deltas of the elements upon stepping the process.

Specifically, [19] introduces an array P, where each element of P is a list of token change and element identifier pairs – essentially, P enumerates the token changes for each allowed stepping of the BPMN model (not unlike Petri net incidence matrices do).

We construct a very similar P array and embed it into the zkWF program to enable checking whether a proposed state update is valid from the BPMN execution logic point of view.

Our token passing-based operational semantics is the following. Initially, we create a token for every start event and pass it to the first executable element connected to it. Each executable element has one incoming and one outgoing edge. When an executable event has a token, it is marked as "active". After completing the execution of the element, the element is marked as "completed", and we pass its token to the next executable element – based on the token holder element's outgoing edge. This approach can be modelled as adding a token $(+1)$ when we mark an executable event as "active", and we subtract this token (-1) when we mark the event as "done".

Gateways change the token flow differently. Parallel gateways can split a token on one end and merge them back together on the other end. Exclusive gateways can have many outgoing edges, but only one can be taken based on its assigned expression. A default outgoing edge can also be set, as described in the BPMN specification. End events can have multiple incoming edges but no outgoing edges. They mark the end of a token flow.

To limit the size of our version of array P (necessary to ensure reasonable proof computation times), in our approach, a single step of a model can induce only three token changes at most. (Hence the structural restriction on parallel gateways.) Thus, the array P describing one-step token marking changes for a model M consists of 3-tuples with elements from the set $\mathcal{N}$:

$$\mathcal{N} = (+1, -1\} \times T) \cup \{(0, -1)\} \tag{1}$$

For T, we apply a simple integer encoding; the -1 in the "no-token-change" pair is a don't care placeholder.

V. zkWF programs and their construction

Informally, a zkWF program is the vehicle with which process participants prove that no "illegal" moves – in terms of the agreed-upon rules and current state of the process – are being proposed as a business process step. It is a program shared by the participants for a given process instance and used for creating ZKPs about the state updates they submit to the process manager smart contract.

As a ZoKrates program, a zkWF program has public as well as private inputs, and an output. Private inputs are only visible to the prover. Public inputs are visible to the prover as well as the verifier, and they are necessary to verify proofs (in our context, the process manager smart contract performs the proof verification).

In general terms, the computation happens against a current public process state commitment, stored by the managing smart contract, which is the hash of the current process state, salted by some randomness. This ensures that parties outside the collaboration can't easily guess states from their hash commitments. The computation also relies on the knowledge of the current state and the randomness used for salting the commitment of the current state – these are shared between the collaborating parties in an encrypted form through the smart contract.

However, due to computational limitations, the congruence of this cipertext to the public state commitment is not a part of the ZKP scheme. For this reason, part of the public input (also tracked by the smart contract) is a *signature commitment*: the current hash commitment and the *previous* hash commitment signed by the last acting party (using their application-level EdDSA identity).

Should a participant erroneously or maliciously commit a ciphertext which does not hash to the stated, proven and accepted commitment, this signature ensures that the offending participant can be irrepudiably identified by the other collaborating parties. The zkWF program contains the public keys of the participants for the purposes of the collaboration, thus, proofs are able to imply that a signature commitment was done with the same public key that authorized a party to make an authorized state update. The identity of the party who made the update can be recovered from the participant-tied public keys (known to the collaborators) and the signature commitment. A number of mitigative and corrective schemes are possible for such cases, but these fall outside the scope of the current paper. Further details on the state commitment management protocol are given in section VI.

A. zkWF computation model

Figure 3 illustrates the basic computational approach. Hashing is used heavily; from the selection of hashing algorithms in

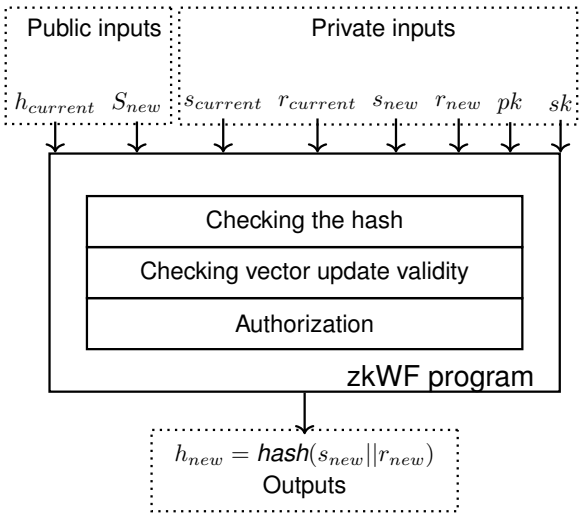

Fig. 3. The basic computation model of zkWF programs

the standard library of ZoKrates, we currently use SHA-256. The *private* inputs of zkWF programs are as follows.

- $s_{current}$ - the current state of the process
- $r_{current}$ - randomness for hashing $s_{current}$ (32 bits)
- s_{new} - the updated ("stepped") process state
- r_{new} - new randomness, for hashing s_{new}
- pk - public EdDSA key of the participant
- sk - private EdDSA key of the participant

In turn, the *public* inputs of the program are as follows. The symbol $||$ denotes concatenation.

- $h_{current} = hash(s_{current}||r_{current})$
- $S_{new} = sig(h_{current}||h_{new})$

$hash$ denotes hashing, sig denotes signing by the party who is proposing the new hash commitment in the concatenation. Based on these inputs, with reference to Figure 3, the computation can be broken down into the following major phases.

1) *"Checking the hash"*. Checking the group-shared secret current state and randomness against the public hash commitment to ensure ongoing integrity.
2) *"Checking vector update validity"*. Checking that no illegal state transition is being proposed through s_{new} at the *application logic* level (as specified by the BPMN model).
3) *"Authorization"*. Checking the new signature commitment given as a public input (based on pk and sk) and checking the authorization of the participant for the business process step.
4) The program outputs the hash of the new state $h_{new} = hash(s_{new}||r_{new})$.

Most aspects of the computational model are straightforward; for further details, the reader is kindly referred to the full report and the implementation. We only expand on some important aspects of the BPMN model encoding and the logic for checking BPMN state change validity.

B. BPMN model encoding

The BPMN logic is fundamentally carried over into the zkWF program through a precomputed P array. Additionally, to check whether the right paths are proposed for exclusive gateways, the expressions on the sequence flows after the gateways are also encoded in the program as assertions. Message passing and variable write permission checks are addressed similarly.

C. BPMN state change validity check

The zkWF program compares $v_{current}$ and v_{new} from $s_{current}$ and s_{new}. If the two are the same, the "change" is accepted (as a "step" under the fake update mechanism). Four or more differences (pairwise comparisons at the same indices) in the vectors are considered invalid. Otherwise, we construct a 3×3 matrix A with the initial value

$$A = \begin{bmatrix} 0 & -1 \\ 0 & -1 \\ 0 & -1 \end{bmatrix} \tag{2}$$

Then, for the j-th difference ($j \in 0 \ldots 2$) at position $i \in [0, |T|[$ in the vectors, we apply the following updates to A:

- $v_{current}[i] = 1 \ \& \ v_{new}[i] = 2 \Rightarrow A[j] \leftarrow [-1, i]$
- $v_{current}[i] = 0 \ \& \ v_{new}[i] = 1 \Rightarrow A[j] \leftarrow [1, i]$
- $v_{current}[i] = 0 \ \& \ v_{new}[i] = 2 \Rightarrow A[j] \leftarrow [1, i]$

Any other combination of $v_{current}$ and v_{new} values is immediately considered invalid. After matrix A is constructed, we compare each element in the array P to matrix A. If we find an element in P that contains every row in matrix A (in any order), the vector change is considered valid.

Parallel gateway ends induce an additional check. Before a parallel gateway end, two active tasks can exist with two different participants. Because of this, one task can be marked as "completed" without marking the task after the parallel gateway as "active". After both tasks before the parallel gateway are marked as "completed", the executable event on the other end of the gateway must be marked as "active" to continue the token flow.

The state change validity check also includes checking write permissions for global variables and contrasts the evaluation of arithmetic expressions with the proposed path for exclusive gateways.

Finally, the message-handling validation logic involves two major checking aspects. On the one hand, each time a participant wants to mark a Message Throw event as "completed", a message hash has to be uploaded. We assume the actual message to be passed off-chain. On the other hand, each time a participant wants to mark a Message Catch event as "completed", we need to make sure that the corresponding Message Throw event is also marked as completed (likely by another participant). Contrasting the received message with the hash value has to be done by the receiver; if this fails, we assume that the corresponding steps are codified in the process logic.

VI. zkWF protocol design

The zkWF protocol largely follows the general "proofs over commitments and proposed commitment updates" pattern customary in blockchain applications of ZKPs (as depicted in

Figure 1). The *process manager smart contract* component is fairly simple: on the one hand, it stores and updates commitments, and on the other hand, it checks ZKPs over the new commitments proposed in incoming blockchain transactions. Specifically, the process manager smart contract stores the following data (using the notations introduced earlier):

- $h_{current} = hash(s_{current}||r_{current})$
- $C^{enc}_{curr} = enc(s_{current})$
- $S_{current} = sig(h_{prev}||h_{current})$

where enc denotes encryption with the group encryption key and method (see Section III). Update request transactions of the smart contract carry the following arguments:

- $h_{new} = hash(s_{new}||r_{new})$
- $C^{enc}_{new} = enc(s_{new})$
- $S_{new} = sig(h_{current}||h_{new})$
- $p(h_{current}, S_{new}, h_{new})$

The last argument is a ZKP of the correspondence of $h_{current}$, S_{new} and h_{new}, under the pre-agreed zkWF program. The process manager smart contract checks the validity of this proof, before accepting the smart contract state change carried by the other arguments.

VII. Implementation and testing

The ZoKrates toolkit is a central component in our framework; the current implementation uses version 0.7.13[4]. Although the state of the art is changing very rapidly, ZoKrates is the noninteractive ZKP toolkit which has the richest programming language from the point of view of our purposes and is mature enough at the same time.

A. Code generation

Our code generator, implementing the transformation logic denoted on Figure 2, is a custom development in Kotlin. This component generates a zkWF program from a BPMN model (serialized in XML the standard way), relying on a set of ZoKrates template files. First, the model is encoded, as we outlined earlier; then, it generates the code for calculating the state hashes, checking the variable write permissions, ensuring exclusive gateway paths, and verifying message sending. Distribution and validation of the resulting zkWF program among the process participants are not covered by the framework.

As mentioned earlier, in addition to generating zkWF programs, we also support the generation of process manager smart contracts for EVM-based blockchains (via Solidity code generation with a language version 0.8.0 target) as well as for Hyperledger Fabric. Solidity smart contracts are derived from the verifier smart contracts ZoKrates generates for zkWF programs.

In Fabric, smart contracts – "chaincodes" – run in containers and can be developed is classic programming languages, as Java, JavaScript and Go. Consequently, we opted to create a custom chaincode container, where a Java-based chaincode receives smart contract invocations and uses a container-resident copy of the ZoKrates toolkit for proof verification. The structure of the solution and its support of the zkWF protocol are the same as in the case of Solidity.

[4]See https://github.com/Zokrates/ZoKrates/releases

B. Client side

We created a simple participant-side SDK, which wraps ZoKrates (for proof generation) and incorporates the Web3J wallet library. For testing and demonstration purposes, we also created a TornadoFX-based desktop GUI application (WFGUI, standing for "WorkFlow GUI"). The GUI supports all key participant-side actions: monitoring a process manager smart contract for changes, retrieving state, creating process step proposals, computing their witnesses and proofs, and submitting update proposals according to the zkWF protocol.

WFGUI also incorporates a process modeller for our BPMN subset with our extensions through an embedding of bpmn-js[5]; supports testing through preassembled smart contract call sequences, which use different keys for different participants; and supports process manager smart contract deployment to Ethereum-based blockchains.

C. Testing

We assembled a suite of *simple test cases* and known important *"corner cases"*, based on the test suites accompanying the tools we referred to in section II. These can be found in the code repository.

BPMN model size and complexity influence zkWF program size and complexity, which, in turn, determine proof computation times and on-chain verification costs. Consequently, to demonstrate the practical feasibility of our approach, as a *representative test case*, we used an anonymized version of a car leasing process model we created in the "Digitisation, artificial intelligence and data age workgroup" of the ongoing BME-MNB cooperation project. (MNB is the Hungarian National Bank.) The process involves four participants – client, dealer, leasing company, and commercial bank – and the model has 68 vertices and 69 edges. The representative process model is included in the repository, where a demonstrational video is also available.

Not only is this process model representative, we argue that real-life use cases will not necessarily involve much larger models. Significantly larger models are usually transformed into hierarchical process models in the practice – and, while in this paper we have not addressed this question yet, there is a clear path from the current solution towards a set of process manager smart contracts, which collectively manage the state commitments of a process hierarchy and remain efficient from the point of view of proof generation and verification costs.

The test cases are executed by a custom test scenario runner framework, which has a CLI interface (for CI/CD pipeline integration) in addition to its integration into WFGUI.

VIII. Performance

In addition to functional testing (compliance with model semantics, proper enforcement of authorization aspects and proper handling of compliant/noncompliant proofs), we used our test suite to evaluate key performance metrics of the approach. Performance tests were performed on a desktop PC (AMD Ryzen 7 2700, 16 GB of DDR4 memory); for

[5]See https://bpmn.io/toolkit/bpmn-js/

gas measurements, we used a private Ethereum test network with geth version 1.10.25. Note that blockchain-side efficiency measurements are largely irrelevant for Hyperledger Fabric (which has no "gas" notion and where the smart contract execution layer is very highly resource-scalable). Table II summarizes the relevant size metrics of our test cases. The size of P is understood in the number of 3-tuples in the array.

TABLE II
SIZE CHARACTERISTICS OF THE TEST CASES AND SCENARIO COUNTS

Case	Vertices	Edges	Executable	Size of P	Scenarios
Test 1	5	4	3	3	3
Test 2	9	10	5	7	9
Test 3	8	8	4	4	4
Test 4	6	5	2	3	2
Test 5	14	12	10	10	10
Repr.	68	69	50	54	52

Table III summarizes the runtimes of the off-chain computations. The compilation and zk-SNARK setup phases were executed once; proving time is the sum of computing the witness and generating the proof, and we give an average over the scenarios.

In summary, the results indicate that deploying a smart contract and stepping the execution can be done in just a few minutes. We can also see that, although the representative model is 5-6 times larger than the larger ones, proofs only took about 2.3 times more time to generate. We interpret this as a strong indication that our approach is practically feasible for real-life models.

TABLE III
OFF-CHAIN COMPUTATION RUNTIMES

Case	Compilation time	Setup time	Proving time avg.
Test 1	27.22 s	129.58 s	55.0 s
Test 2	48.32 s	182.80 s	88.67 s
Test 3	28.55 s	129.69 s	53.40 s
Test 4	27.14 s	128.82 s	53.21 s
Test 5	30.74 s	133.44 s	54.10 s
Repr.	81.02 s	187.33 s	122.47s

Table IV summarizes smart contract deployment cost to Ethereum and the average gas cost of (updating) smart contract calls in the zkWF protocol. Note that although the representative model is 5-6 times larger than the simple ones, the smart contract call gas cost is only moderately higher. As the hashes, signatures, and proofs have a fixed length, gas usage variability is essentially driven by the size of the encrypted version of the current state.

TABLE IV
GAS USAGE ON ETHEREUM

Case	Deployment gas usage	Update gas usage avg.
Test 1	2,098,786 gas	490,507 gas
Test 2	2,098,990 gas	497,780 gas
Test 3	2,098,498 gas	493,705 gas
Test 4	2,078,071 gas	503,817 gas
Test 5	2,161,039 gas	491,783 gas
Repr.	2,408,635 gas	548, 898 gas

Performance-wise, it is hard to compare our approach to others, since, to our knowledge, we are the first to use zero-knowledge proofs to hide the current state of BPMN execution on-chain. Despite some common points, even [19] remains incommensurable due to differences in goals, basic approach and tools.

That said, the Ethereum smart contracts' gas usage can be certainly compared to existing techniques. The deployment cost is on par with or is less than the existing solutions. In our case, the cost of updating the state is significantly higher compared to previous approaches like Chorchain [12] or Caterpillar [8]. Chorchain [12] uses about 92,905 gas on average for each message, while Caterpillar [8] requires similar amounts of gas on average.

This "confidentiality premium", while it may not be currently acceptable on the mainnet (at the time of this writing, the block target size is 15 million gas), can be fully acceptable on permissioned EVM blockchains and sidechains. Additionally, support for the efficient checking of ZKP commitments is being developed very intensively for the Ethereum platform.

IX. THREATS TO VALIDITY AND FUTURE WORK

Earlier, we addressed our current constraints with respect to BPMN models which are admissible under our current approach; future work will target lifting these limitations. We dealt with the inability of ZoKrates not being able to verify that a given ciphertext is the encrypted form of a given message with a given key by constructing the protocol so that parties submitting a noncompliant ciphertext can always be identified irrepudiably. We accept smart contract gas costs as an outstanding issue; however, it is one which does not truly limit the applicability of our approach on permissioned-consensus platforms and one which we can reasonably expect to become a non-issue for the Ethereum mainnet, too. We will also investigate whether we can sidestep this issue by transforming our approach into a Layer 2 ZKP rollup scheme, where we can, at the very least, amortize gas costs across batches of process manager smart contract updates.

A major remaining threat to validity is compliance with BPMN operational semantics, especially after the planned future extension of the supported BPMN subset. For the approach presented in this paper, we only tested compliance and not formally proved it. Here we note that as long as all participants are aware of the way BPMN models are translated to admissible and nonadmissible state changes in zkWF programs, even divergences from the standard-prescribed and commonly accepted operational semantics may be acceptable, but this is clearly not a fully satisfactory answer to the issue.

Formally proving the operational semantics-preserving nature of our transformation logic would be a possible approach, but that would first need migrating the implementation to a transformation model where the translation rules themselves are first-class objects (such as in the model transformation platform VIATRA [20]), and even then, it would be a highly complicated endeavour.

Instead, future work will first investigate proving behavioural equivalence between reference BPMN behaviour

and zkWF programs on a *model-by-model basis*, as a prerequisite check integrated into the toolchain. Recent work has formalized BPMN collaboration semantics in a way amenable for state space model checking [21]; this opens up the possibility to perform bisimulation analyses between the state transitions of models under "authoritative" semantics and under zkWF program encoded semantics. This approach will have the added benefit that it facilitates the introduction of property checking (soundness, safeness and application-specific requirements) on the BPMN models themselves.

Last but not least, meeting our confidentiality goal partially relies on the participants blending their actual updates into a stream of "fake" updates. Currently, we are developing three cooperative fake update regimens: a by-participant random, a participant-cyclic and a participant-periodic one; however, these still require formal analysis of their state confidentiality preserving properties (in terms of the vector v).

X. Conclusion

In this paper, we presented a confidentiality-preserving approach for the smart contract-based orchestration of business collaborations, captured as BPMN 2.0 models. Our protocol is a novel, and to our knowledge, one of its kind solution, which we validated semantically as well as evaluated from the resource usage and gas cost points of view. We also described a full toolchain prototype which we made available as open-source software.

Acknowledgment

This work was partially created under, and financed through, the Cooperation Agreement between the Hungarian National Bank (MNB) and the Budapest University of Technology and Economics (BME) in the Digitisation, artificial intelligence and data age workgroup. The work of Imre Kocsis was partially funded by the National Research, Development, and Innovation Fund of Hungary under Grant TKP2021-EGA-02.

References

[1] W. M. Van Der Aalst, M. La Rosa, and F. M. Santoro, "Business process management: Don't forget to improve the process!" *Business & Information Systems Engineering*, vol. 58, no. 1, pp. 1–6, 2016. doi: 10.1007/s12599-015-0409-x

[2] S. Pourmirza, S. Peters, R. Dijkman, and P. Grefen, "A systematic literature review on the architecture of business process management systems," *Information Systems*, vol. 66, pp. 43–58, 2017. doi: 10.1016/j.is.2017.01.007

[3] Object Management Group, "Business Process Model and Notation (BPMN), Version 2.0." [Online]. Available: https://www.omg.org/spec/BPMN/2.0/

[4] M. Chinosi and A. Trombetta, "BPMN: An introduction to the standard," *Computer Standards & Interfaces*, vol. 34, no. 1, pp. 124–134, 2012. doi: 10.1016/j.csi.2011.06.002

[5] J. Mendling *et al.*, "Blockchains for Business Process Management - Challenges and Opportunities," *ACM Trans. Manage. Inf. Syst.*, vol. 9, no. 1, 2018. doi: 10.1145/3183367

[6] J. Eberhardt and S. Tai, "ZoKrates - Scalable Privacy-Preserving Off-Chain Computations," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, 2018. doi: 10.1109/Cybermatics_2018.2018.00199 p. 1084–1091.

[7] Y. Ait Hsain, N. Laaz, and S. Mbarki, "Ethereum's Smart Contracts Construction and Development using Model Driven Engineering Technologies: a Review," *Procedia Computer Science*, vol. 184, p. 785–790, 2021. doi: 10.1016/j.procs.2021.03.097

[8] O. López-Pintado, L. García-Bañuelos, M. Dumas, I. Weber, and A. Ponomarev, "Caterpillar: A business process execution engine on the ethereum blockchain," *Software: Practice and Experience*, vol. 49, no. 7, p. 1162–1193, 2019. doi: 10.1002/spe.2702

[9] L. Mercenne, K.-L. Brousmiche, and E. B. Hamida, "Blockchain Studio: A Role-Based Business Workflows Management System," in *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, 2018. doi: 10.1109/IEMCON.2018.8614879 p. 1215–1220.

[10] A. Abid, S. Cheikhrouhou, and M. Jmaiel, "Modelling and Executing Time-Aware Processes in Trustless Blockchain Environment," in *Risks and Security of Internet and Systems*, ser. Lecture Notes in Computer Science, 2020. doi: 10.1007/978-3-030-41568-6_21 p. 325–341.

[11] Q. Lu *et al.*, "Integrated model-driven engineering of blockchain applications for business processes and asset management," *Software: Practice and Experience*, vol. 51, no. 5, p. 1059–1079, 2021. doi: 10.1002/spe.2931

[12] F. Corradini *et al.*, "ChorChain: A Model-Driven Framework for Choreography-Based Systems Using Blockchain," in *Proc. of the 1st Italian Forum on Business Process Management (ITBPM)*, 2021, pp. 26–32.

[13] ——, "Model-driven engineering for multi-party business processes on multiple blockchains," *Blockchain: Research and Applications*, vol. 2, no. 3, p. 100018, 2021. doi: 10.1016/j.bcra.2021.100018

[14] ——, "Flexible Execution of Multi-Party Business Processes on Blockchain," in *2022 IEEE/ACM 5th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2022. doi: 10.1145/3528226.3528369 p. 25–32.

[15] E. Androulaki *et al.*, "Hyperledger Fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, 2018. doi: 10.1145/3190508.3190538 p. 1–15.

[16] ZKProof Community, "ZKProof Community Reference," 2022. [Online]. Available: https://docs.zkproof.org/reference.pdf

[17] J. Groth, "On the Size of Pairing-based Non-interactive Arguments," in *Advances in Cryptology – EUROCRYPT 2016: 35th Annual International Conference on the The-*

ory and Applications of Cryptographic Techniques, 2016. doi: 10.1007/978-3-662-49896-5_11 pp. 305–326.

[18] J. Groth and M. Maller, "Snarky signatures: Minimal signatures of knowledge from simulation-extractable SNARKs," in *Advances in Cryptology – CRYPTO 2017: 37th Annual International Cryptology Conference*, 2017. doi: 10.1007/978-3-319-63715-0_20 pp. 581–612.

[19] T. Aivo, "Zero-Knowledge Proofs for Business Processes," Master's thesis, Univ. of Tartu, 2020.

[20] G. Bergmann *et al.*, "VIATRA 3: A Reactive Model Transformation Platform," in *Theory and Practice of Model Transformations*, 2015. doi: 10.1007/978-3-319-21155-8_8 pp. 101–110.

[21] F. Corradini *et al.*, "A formal approach for the analysis of BPMN collaboration models," *Journal of Systems and Software*, vol. 180, p. 111007, 2021. doi: https://doi.org/10.1016/j.jss.2021.111007

Balázs Ádám Toldi was born in Budapest, Hungary, in 2000. He received his BSc in computer engineering in 2023 from the Budapest University of Technology and Economics (BME). Currently, he is an MSc student at BME, with a primary specialization in cybersecurity and a secondary specialization in critical systems.

Imre Kocsis received his PhD from the Budapest University of Technology and Economics (BME) in 2019. Currently, he serves as a senior lecturer and leading blockchain researcher at the Critical Systems research group of the Dept. of Measurement and Information Systems of BME. He leads the activities of the group in conjunction with the Hyperledger Foundation and the university's participation in the European Blockchain Services Infrastructure (EBSI) network.