Blockchain-Based, Confidentiality-Preserving Orchestration of Collaborative Workflows

Balázs Ádám Toldi, and Imre Kocsis

***Abstract*—Business process collaboration between independent parties is challenging when participants do not completely trust 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 Business Process Model and Notation (BPMN) models. However, compared to centralized business process orchestration services, smart contract state typically leaks potentially sensitive information about the state of the collaboration, limiting the applicability of decentralized process orchestration. This paper presents a novel, collaboration confidentiality-preserving approach where the process orchestrator smart contract only stores encrypted and hashed process states and validates participant actions against a BPMN model using zero-knowledge proofs. We cover a subset of BPMN, which is sufficient from the practical point of view, support message-passing between participants, and provide an open-source, end-to-end prototype implementation that automatically generates the key software artifacts.**

***Index Terms*—blockchain, BPMN, orchestration, collaboration, confidentiality, zero-knowledge proofs**

I. Introduction

In modern business science, *Business Process Management* (BPM) as a discipline [1] advocates process-focused thinking about internal activities and external collaborations to improve key performance indicators. Automating the execution of business processes is a key proposition of BPM and has been supported for a long time by various technical solutions [2]. Today, most of these, typically centralized, tools and services use the leading business process modeling standard, Business Process Model and Notation (BPMN) 2.0 [3] as a process definition language [4].

Distributed ledger technology (DLT), 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 cannot agree on a trusted (third) party as a middleman [5]. Blockchain-deployed smart contracts can impartially enforce the agreed-on sequences of activities and track sent and received messages. Smart contracts can also host data objects acted on by a process directly or anchor their changes in the blockchain via cryptographic commitments.

However, blockchain-assisted BPM is still a relatively new discipline – importantly, known BPMN-based solutions are inadequate from the privacy and confidentiality point of view. This paper presents a novel, collaboration confidentiality-preserving approach and end-to-end prototype tooling for the on-chain process orchestration of cross-organizational, BPMN-based collaborations using zero-knowledge proofs (ZKPs)[1]. Specifically, for a sufficient subset of BPMN, we present a transformation of the admissible state updates of BPMN process instances to programs of the ZoKrates [6] toolkit. We assemble state update validity provers from these programs for the participants and proof-verifying orchestrator smart contracts. We define an on-chain process state commitment update protocol, describe our open-source end-to-end implementation prototype[2] and evaluate practical viability.

Our contribution is novel from two aspects. First, to our knowledge, the confidentiality challenges of decentralized BPMN orchestration have not been addressed systematically and constructively yet. Second, we express BPMN execution as an incremental computation in a form amenable to commit-and-prove style zero-knowledge validation in smart contracts. This paves the way for further research on the computational representation of orchestrated BPMN execution against the continuously appearing ZKP advancements.

II. Motivation and problem statement

BPMN is a standardized approach to visually and precisely express *how* business processes should be performed. BPMN is used in many domains – including finance, banking, manufacturing, healthcare, logistics and telecommunications – for capturing processes with well-defined sequences of regularly repeated activities. The BPMN standard defines several model types, *process*, *collaboration* and *choreography* being the most widely used ones. *Process (flow)* models are the simplest: these express the sequence, preconditions and exception handling of a single process performed by a single organization. Collaborations model the individual processes performed by collaborating parties – usually business entities – and their messaging-based interactions. Choreography diagrams focus solely on the message exchanges between collaborating entities.

Dept. of Measurement and Information Systems Budapest University of Technology and Economics Budapest, Hungary
(E-mail: balazs.toldi@edu.bme.hu, kocsis.imre@vik.bme.hu)

DOI: 10.36244/ICJ.2023.3.8

[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: https://tdk.bme.hu/VIK/sw8/Kollaborativ-munkafolyamatok-titkossagmegorzo

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

A. Decentralized orchestration

For over a decade, software tools have been available to assist with *process execution*. The more sophisticated ones track and *orchestrate* activities according to a BPMN model, register activity-related data and perform decision-making on further process evolution. However, centralized orchestration introduces a trusted orchestrator party requirement when we move beyond single-entity processes. With the emergence of blockchain and distributed ledger technology, the potential of decentralizing various aspects of cross-organizational collaboration has been recognized quite early.

Consider the BPMN car leasing collaboration model in Figure 1[3], where the internal processes of a car dealership, a leasing company and a financing bank must be coordinated to accept a leasing application. Individual executions of models are called *instances* and have an *instance state*. The coloring in Figure 1 demonstrates a state: green denotes that the dealership has completed insurance processing and the leasing company and the bank would be able to begin processing. However, for the leasing company to proceed, active downpayment checking (orange) must be finished, then the downpayment filed, and a downpayment notification sent and received.

Using blockchain-deployed smart contracts that track *collaboration state*, the *execution enablement* and *execution obligation* of the activities of the parties can be enforced without a dedicated, trusted party. The transaction journal nature of blockchains can also ensure that the full trace is also stored in an immutable and irrepudiable way. While tracking the internal state of participant-internal processes on-chain is not always desirable, it is a valuable *option*; e.g., when decisions have to be made in a way verifiable by the other collaborating parties.

Orchestrating and journaling messages and collaborative data handling are two further collaboration aspects which can be improved with ”blockchainification”. In both cases, the orchestrator smart contracts usually only manage cryptographic (hash) commitments to externally handled messages and data modifications, to avoid storing sizeable data on-chain.

Tools and approaches exist to create orchestrator smart contracts from BPMN models (see Section III). However, no systematic solution exists to protect sensitive collaboration state information in the smart contract state from parties who can read the blockchain but do not participate in the collaboration. In our example, a leasing company may wish that its competitors do not see how many open cases they have, how long it takes to perform key steps in the process, or what lease rates they apply.

Fulfilling such requirements is a confidentiality challenge that contradicts core blockchain design principles. Blockchain nodes must be able to *validate* and *execute* incoming transaction requests to reach consensus on ledger updates, be those changes of the balances of a natively tracked cryptocurrency or state changes of deployed smart contracts. If the transaction details are made ”incomprehensible” to the nodes, e.g., by off-chain encryption, they can’t validate the preconditions for performing the transaction and compute state updates. For smart contracts, the dominant *cryptographic* answers to this dilemma are validating transactions with ZKPs and confidentiality-preserving execution using homomorphic encryption, with the prior being significantly better established currently.

[3] The model was created in the ”Digitisation, artificial intelligence and data age workgroup” of the ongoing BME-MNB cooperation project. (MNB is the Central Bank of Hungary.). For legibility, the process in Figure 1 is slightly simplified; the whole model is available in our project repository.

B. Problem statement: BPMN collaboration confidentiality

We set up our problem statement through a basic system model and the enumeration of required security properties. We target a simple form of collaboration confidentiality (see the properties below) under the assumption that it is not in the interest of any process participant to leak information about process instances; participants neither directly leak information nor help external parties to compromise confidentiality. This is one of the realistic models for our setting, even though the participants do not completely trust the *actions* of each other. We will touch briefly on stronger models in Section IX.

1) Basic assumptions and terminology: participants wish to collaborate in the execution of an *instance* of a previously agreed-on BPMN collaboration definition. All other parties are *process external*. All participants have a cryptographic key pair for signature-based authentication and process activity authorisation. The underlying process model is public knowledge, but the public keys are shared only between the participants. We assume the absence of private key compromises.

For the underlying blockchain, we assume complete integrity (no successful attack on the consensus) and, for the sake of simplicity, deterministic finality (accepted blocks do not get retracted). Note that even blockchains with probabilistic block finality are usually quasi-deterministically final already at the time scale of a few blocks. On the other hand, process external parties have complete visibility of blockchain transactions. 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 request load. While, in practice, blockchain platforms have strongly varying fault and threat models and sensitivity (see, e.g., [7]), these are basic assumptions of normal operational conditions. As a part of platform selection, security and dependability analysis should evaluate the risk of these assumptions not being met.

2) System model: the classic Business Process Orchestrator (BPO) middleware pattern [8] facilitates business process execution by providing a message broker and extending it with state management and persistent state storage. The solutions in the state-of-the-art closely match this pattern. (Technically, message passing is only *coordinated* and journaled by the smart contract their core.) The smart contract as a Process Controller [8] also performs authentication and authorization based on the BPMN model to ensure that the stored state sequence never deviates from the model semantics. We also aim to employ a blockchain-deployed smart contract as a BPO.

3) Security properties: we target a set of integrity, availability and confidentiality guarantees. Integrity and availability properties are already covered by the prior art; our contributions lie in establishing collaboration confidentiality, as defined by properties **C1** and **C2**, despite using smart contracts. BPO-SC refers to a per-process instance BPO smart contract.

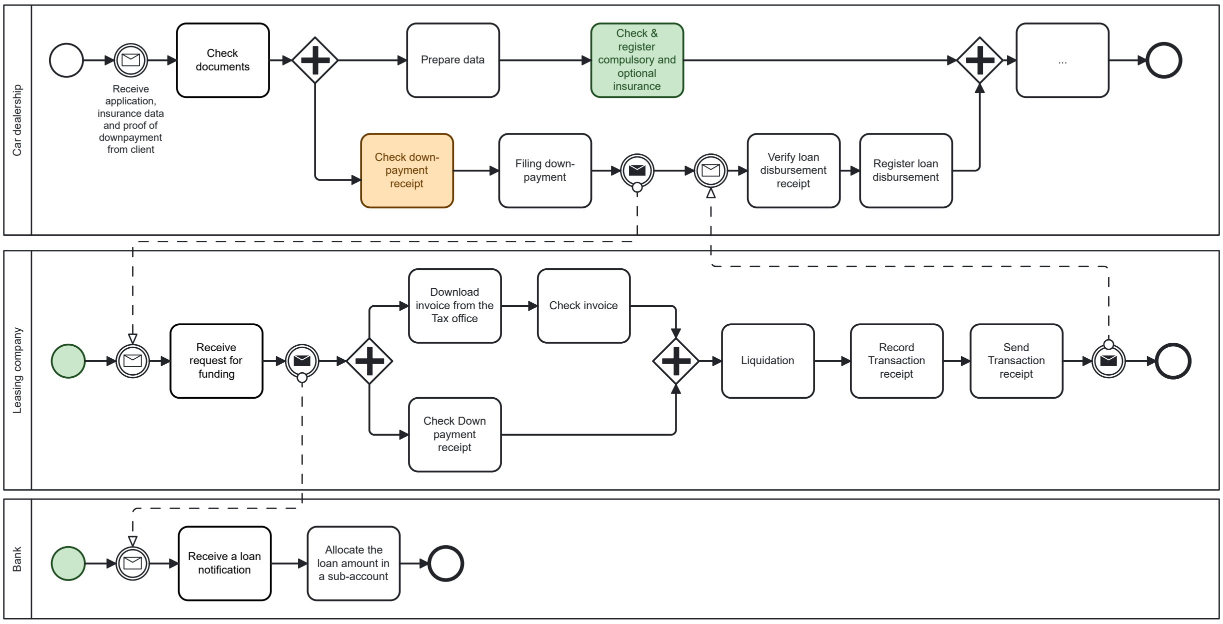

Fig. 1. Car leasing BPMN collaboration example (simplified for presentation). In the depicted example state, green denotes "completed"; orange is "active".

I1: The state traces enforced by the BPO-SC always adhere to the operational semantics of the underlying BPMN model.

I2: Process-external parties cannot influence the BPO-SC state.

I3: Process state updates can be initiated only by the participants authorized by the model and instance state.

A1: No external party can influence authorised participants' ability to perform state updates in a bounded time.

A2: No participant can influence the ability of authorized participants to perform state updates in a bounded time.

A3: Participants can always learn the trace and current state.

C1: External parties cannot determine participant identities.

C2: No external party can learn more about the trajectory, timings and stepwise properties (e.g., process variables and message contents) of the trace during and after execution than the fact that an instance has been started.

III. RELATED WORK

Smart contracts, as a rule, cannot be altered after deployment; thus, to minimize the probability of software faults, domain-specific languages and Model-Driven Engineering (MDE) are steadily gaining ground in smart contract development [9]. In our context, the established approach is a BPMN model to serve as a *specification*, and orchestrator smart contract logic is generated automatically from the model.

A. Decentralized business process orchestration

Caterpillar [10] 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 extended feature set, like Blockchain Studio [11], which adds role management, or [12], which adds time constraints. Lorikeet [13] 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 [14] takes a BPMN *choreography* and generates an Ethereum smart contract that can be used to execute the model. ChorChain also includes a dedicated modeling tool. The same authors released two further tools: Multi-Chain [15] and FlexChain [16]. Multichain is similar to Chorchain, but it also supports Hyperledger Fabric [17]. 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 and trace are easily recoverable from the process manager smart contracts for *all* the tools above.

B. Commit-and-prove ZKP with smart contracts

Zero-Knowledge Proofs (ZKPs) are cryptographic methods to prove the validity of various statements without revealing any additional information [18]. ZKP verification in a smart contract requires a scheme with "single-shot" message passing from prover to verifier; in this work, we rely on zk-SNARKs, a family of *noninteractive*, and also *succinct* (small and cheaply verifiable proofs) ZKPs. We use the ZoKrates toolkit as a ZKP front-end with a high-level programming language [6]. ZoKrates currently supports the Groth16 [19], GM17 [20] and Marlin [21] proving schemes.

Our contribution implements a commit-and-prove approach. In commit-and-prove schemes, a party first commits to an

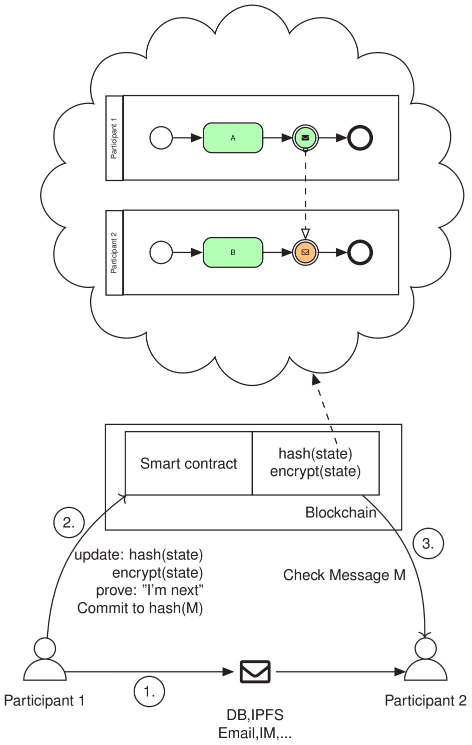

Fig. 2. Overview of the zkWF protocol

input and, possibly later, proves some predicate about the input – without revealing it [22]. This is a widely used pattern in the smart contract-based application of zero-knowledge proofs. Recent surveys on ZKP schemes, technologies and applications can be found in [23] and [24].

IV. A CONFIDENTIALITY-PRESERVING APPROACH

The fundamental difference of our approach from [10–16] is that instead of storing the process state on-chain in an easily interpretable form in an orchestrator smart contract, we store encrypted states and cryptographic state commitments and accept update proposals on the presentation of ZKPs over the current and proposed commitment. The approach relies on two key conceptual components: our zkWF ("zero knowledge WorkFlow") protocol and what we call "zkWF programs".

A. The zero-knowledge WorkFlow (zkWF) protocol

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. Figure 2 presents a high-level overview.

At the centre of the scheme is a smart contract instance on a blockchain for an instance of a collaboration model. This smart contract stores and manages the state of the collaboration – as specified by the underlying BPMN model – in an encrypted and a hashed form.

During process execution, the collaborating parties can send messages to each other by off-chain means ①. These are captured in the underlying process specification as intermediate message throw and capture events; our state commitment scheme includes commitments to the message hashes.

When a participant wishes to update the state stored in the smart contract – that is, to "step the process" –, it has to create a ZKP that the proposed state transition is valid. This new state includes the hash of the message they sent beforehand if the step involves message sending. It sends the new state hash commitment, the encrypted new state and the ZKP proof of state transition validity to the smart contract as a blockchain transaction ②; the smart contract updates its state only if it can successfully check the ZKP.

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 (and proceeds with its part of the collaboration) if the hashes match ③.

Participant authentication is tied to proving private key ownership in the ZKPs. The public keys are defined over the participant group-shared process model as a parameterization. These are cooperation-private, "application-level" key pairs; on pseudonymizing platforms, such as Ethereum, updater identity can and should be masked by using independent, single-use transaction source addresses (i.e., public keys).

Additionally, we require the participants to have a common means for encrypting and decrypting stored state ciphertexts. The protocol does not constrain the encryption used.

The protocol can be realized straightforwardly on a wide range of DLTs; we provide an implementation for Ethereum and Hyperledger Fabric [17]. While the updates and the contract state are unintelligible to parties outside the collaboration, statistical and model trace analyses of the update sequences are still a threat. We enable mitigations by including a "fake" update transaction variant (no actual state update), which all participants are authorized to use.

B. zkWF programs

zkWF programs are generated from BPMN specifications and serve as a bridge between process definition and proof computation/verification. 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. We use the zkWF program to generate the zero-knowledge proofs and proof verification code for the orchestrator smart contract.

C. Workflow and toolchain

We created an end-to-end toolchain prototype for our approach, as depicted in Figure 3.

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

In the *synthesis phase*, the ZoKrates toolkit is used to set up the *prover key* and *verifier key* and generates the verifier smart contract in Solidity. We created novel support for generating verifier code for Hyperledger Fabric in Java. We also created the code generation facilities for both platforms' state commitment management part of the smart contracts.

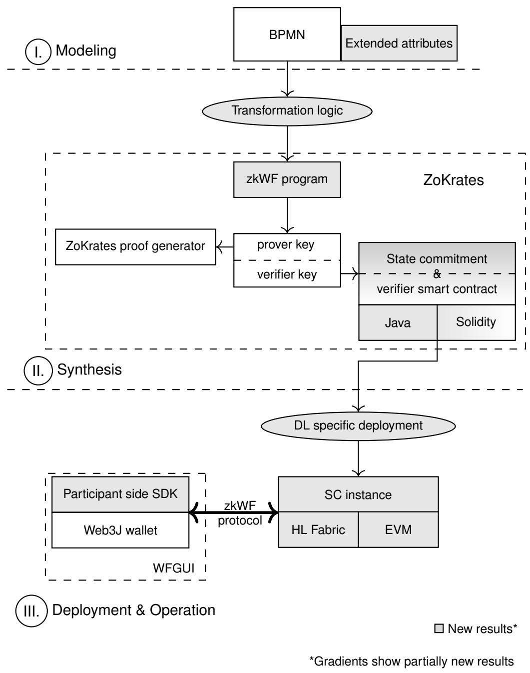

Fig. 3. Toolchain overview

Some secret values used when creating zk-SNARK prover and verifier keys are considered "toxic waste": an adversary can use them to break the scheme, e.g., forge fake proofs. Thus, security relies on the waste having been deleted. The associated risk can be mitigated by using a reliable party for the key generation or performing so-called multi-party trusted setup ceremonies, where a (large) group of actors assembles the keys. In this case, security requires only at least one of them to delete the waste. Such ceremonies tend to be complicated and thus can pose a problem for by-program setup. Universal schemes also exist (e.g., [21]), where the results of a single program-agnostic ceremony can be used to derive program-specific keys publicly and securely. Choosing the right approach requires deployment-specific risk analysis; ZoKrates supports all of the above.

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 for the client side. Here, we integrate the ZoKrates toolkit as a proof generator.

V. BPMN SUBSET AND EXECUTION SEMANTICS

This paper targets the Basic Modeling Elements of BPMN 2.0 [3, p. 28], the core subset of the specification, with the restrictions that regarding events, we interpret only message throw and catch ones (among participants) and do not support sub-processes and data objects. We argue that this element set is already sufficient for practical applications. Statistical evidence [25] shows that the usage frequency of the 50 constructs in the BPMN specification follows a Zipfian distribution; we cover elements used at least in $\sim 25\%$ of the models in [25]. This is also the empirically established "Common Core of BPMN" in [25] with the addition of messaging between participants. Our earlier example showcases the supported element set (except for "exclusive gateways" for process variable-based choice paths and "lanes" for further subdividing pools).

A. BPMN extensions and structural constraints

We introduce two extended attributes for BPMN elements. `zkp:publicKey` separates the tasks of different participants by attaching a participant-specific public key to a pool, a lane, or a *participant-executable element* (activities, message throws and catches). `zkp:variables` applies to *activities* and declares process instance global variables, and that that activity may write the variable (reads are allowed for all activities). These variables can be used in boolean expressions for exclusive gateways.

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

- Gateways must be binary (two incoming/outgoing edges).
- Activities are *atomic*; i.e., subprocesses are not supported.
- The model must be acyclic (no loops).

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 report and 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 $M = (V, E, T)$ be a process model, 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 participant
- 2 (Completed) – The execution of the element has been completed

This state set is a subset of those in the standard activity lifecycle [3, p. 428] and serves as a reasonable simplification, as the main focus of the work described here is exploring the confidential execution aspect. Note that correctly implementing the full lifecycle is a significant software engineering effort, even in the centralized setting. Also, BPMN users tend to apply a similar simplified view during modeling, as the more sophisticated state aspects require experience and limit the ease of model understanding.

C. Capturing token passing semantics

BPMN 2.0 models have straightforward, token flow-based standard execution semantics: start events create tokens that move around as execution progresses. Parallel gateways split and join tokens. To support a different ZKP use case, [26] introduces a technique for representing valid BPMN execution state changes by enumerating the possible composite token marking deltas of the elements upon stepping the process. Specifically, [26] introduces an array P, where each element of P is a list of token change and element identifier pairs. We construct a similar P array under the token passing semantics 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 P array to describe one-step token marking changes for a model M consists of 3-tuples with elements from the set $\mathcal{N}$:

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

For T, we apply a simple integer encoding; the -1 in the ”no-token-change” pair second set is a don't care placeholder. Especially under our binary gateway condition, which is currently necessary to ensure reasonable proof computation times, it is straightforward to enumerate the admissible changes based on the BPMN model. For example, let's consider activities $a, b, c \in T$. a continues in a parallel gateway, which proceeds to b and c. When a transitions from ”Active” to ”Completed” and b and c from ”Inactive” to ”Active”, the following token marking change happens: $((-1, a), (+1, b), (+1, c)) \in \mathcal{N}$. The complete logic can be found in the referenced report.

VI. zkWF program and protocol design

A zkWF program is a ZoKrates program shared among the participants, with which process participants prove that a business process state transition they propose is allowed. In ZKP terms, the participants are the *provers*, and the orchestrator smart contract is the *verifier*.

ZoKrates programs have public as well as private inputs, and an output. Private inputs are only visible to the prover; public inputs are visible to the prover and the verifier, and they are necessary to verify proofs. In our case, the current commitment and the proposed one act as public inputs. Private inputs are more varied; only some are shared across the participants (e.g., the cleartext of the current state).

The key current deficiency of our scheme is that our proofs do not include showing the congruence of the on-chain stored state ciphertexts and the public state (hash) commitments. Combining established encryption algorithms with zk-SNARKs is hard; advances are being made (see, e.g., [27]), but these haven't appeared in any of the leading zk-SNARK frameworks yet as vetted and reusable ”gadgets”.

We apply the following measures to this deficiency. An additional part of our public input (and blockchain-stored data) will be a *signature commitment*: the current hash commitment and the *previous* hash commitment signed by the last acting party (using their application-level cryptographic identity). Should a participant erroneously or maliciously commit a ciphertext that 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.

Although several partially mitigative and corrective schemes can be built on this measure, we introduce the weakening assumption that the irrepudiable identifiability of participants halting execution this way is a sufficient disincentive.

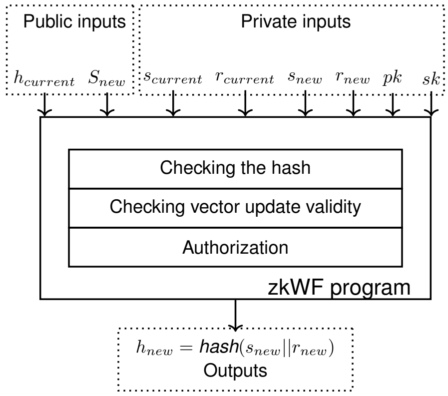

Fig. 4. The basic computation model of zkWF programs

A. zkWF computation model

Figure 4 illustrates the structure of the generated zkWF programs. For hashing, we use SHA-256; application-level signing uses the EdDSA implementation from the ZoKrates standard library (both widely used, NIST-standard algorithms). The *private* inputs of zkWF programs are as follows.

- $s_{current}$ - the current state of the process (subsec. V-B)
- $r_{current}$ - random salt 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 (subsec. IV-A)
- sk - private EdDSA key of the participant

The *public* inputs (|| denotes concatenation):

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

sig denotes signing by the party proposing the new hash commitment in the concatenation. Given these inputs, the following steps are performed.

1) Checking the group-shared secret current state and randomness against the public hash commitment to ensure ongoing integrity.
2) Checking that no illegal state transition is being proposed through s_{new} at the *process logic* level.
3) 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.

Most aspects of the computational model are straightforward; we only expand on the important details of BPMN model encoding and the state change validity checking logic.

B. BPMN model encoding and state change validation

The BPMN model logic is carried over into the zkWF program by a precomputed P array (Section V-C). To check whether the correct 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.

Regarding the executable element state vector, the 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 our 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 ($j \in 0 \dots 2$) at position $i \in 0 \dots |T| - 1$ in the vectors, we apply the following updates:

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

Any other combination of $v_{current}$ and v_{new} values is invalid. If P contains an element with the rows of A, then token passing-wise, the proposed state change is valid, as we essentially decoded the activity token marking changes (± 1) from the activity state changes: 0 - Inactive $\rightarrow$ 1 - Active $\rightarrow$ 2 - Completed.

Parallel gateway ends ("joins") induce an additional check: a transition from a state where *not both activities before the gateway are completed* to one where *both are* also requires that the activity after the gateway gets activated. State change validation 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 validation aspects. First, a message hash must be provided when a participant wants to mark a Message Throw event as "completed". We assume the actual message to be passed off-chain. Second, when a participant wants to mark a Message Catch event as "completed", we must ensure that the corresponding Message Throw event is also marked as completed. The receiver contrasts the message with the hash value; if this fails, we assume that the further steps are either captured in the process logic or the sender and receiver coordinate corrective transmission off-chain.

C. The zkWF protocol

The protocol flows through the orchestrator smart contract and is simple in light of the earlier sections. The smart contract state contains the following elements:

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

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

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

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

D. Side-channel attack protections

Public BPMN models facilitate side-channel attacks on confidentiality. Our work until now aimed to ensure that the trace steps of the BPMN finite automaton remain unintelligible to the external observer; however, the number and timings of transitions still carry information. Most BPMN models are simple enough to infer a usable probability distribution of possible states and traces from just these observations.

Constant-time execution and delay randomization are two apparent protection options, though both introduce artificial delays. Consider a constant-time token passing ring schedule with dummy operations as our already established scheme. For n participants, we determine a suitable time quantum t with which it is acceptable to wait for $(n-1)t$ to delay the "posting" of any state change. During process execution, at the beginning of the i-th epoch, participant $i \mod n$ checks whether it needs to send a state update transaction. If yes, it does; if not, it issues a "fake update" transaction. After terminating the process, a long fake update stream is advisable. As long as enough participants meet their fake update obligations and adhere to their epochs, external observers only see a heartbeat-like stream of uninterpretable transactions and can determine even the time of termination only with low probability.

VII. SECURITY PROPERTIES

The presented approach addresses the security requirements defined in Section II-B as discussed in this section.

A. Integrity

Property I1 holds in the sense that we carefully implement a strict subset of BPMN semantics, but we acknowledge that future work should create an explicit proof of conformance. I2 holds due to application-level cryptographic authentication; I3 due to cryptographic authentication and the very simple sub-logic of enabling activities and message operations.

B. Availability

A1 holds due to I2 and the blockchain fairness assumption – which is mild for high-throughput public and cross-organizational blockchains. A2 holds only under the disincentive assumption of Section VI. However, the assumption is not strong for domains with a credible threat of legal or regulatory action (e.g., finance). A participant can also perform a denial of service attack with a constant stream of malicious fake updates. The disincentive assumption applies here, too, but fake update regimen-dependent defences can also be introduced in the smart contract (e.g., epoch schedule enforcement). A3 holds due to a smart contract accounting for state and trace and the blockchain platform assumptions.

C. Confidentiality

The C1 guarantee has two layers. At the platform level, all transactions can originate from single-use addresses on pseudonymizing platforms – e.g., Ethereum. In Hyperledger Fabric, the Identity Mixer protocol suite for transactor anonymization and unlinkability can be used similarly. At the application level, transaction payloads and smart contract states contain only hashed, signed and encrypted data. Hashing is straightforward; for the signed content, note that EdDSA signatures do not provide a way to recover the signer's public key from the signature or to determine whether the same key was used to sign two different messages. For the encrypted state, if not a single, group-shared secret is used, an application should choose an encryption scheme where the participant keys cannot be recovered.

C2 depends on external data and transaction uninterpretability, which flows from the cryptographic measures, and transaction unlinkability, which also relies on the measures for C1. It also requires sufficient side-channel protection, for which we have at least one strong (not necessarily efficient) option.

VIII. IMPLEMENTATION, TESTING AND PERFORMANCE

The ZoKrates toolkit is a central component in our framework; the current implementation uses version 0.7.13[4]. ZoKrates was the ZKP toolkit with the best-fitting programming language and ZKP scheme support during our research.

A. Code generation

Our code generator, implementing the transformation logic denoted in Figure 4, is a custom development in Kotlin. This component generates a zkWF program from an XML-serialized BPMN model, relying on ZoKrates template files. First, the model is encoded, as we outlined earlier; then, it generates the code for the described stages of computation and checks.We also generate the orchestrator smart contracts for EVM-based blockchains (Solidity version 0.8.0) and Hyperledger Fabric (Java "chaincode").

B. Client side

We created a simple participant-side SDK, which wraps ZoKrates and incorporates the Web3J wallet library. We also created a TornadoFX-based desktop GUI application ("WorkFlow GUI") for testing and demonstration purposes. 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.

WFGUI also incorporates a process modeller for our BPMN subset and extensions through an embedding of bpmn-js[5]; supports testing through preassembled smart contract call sequences; and supports process manager smart contract deployment to Ethereum-based blockchains. A demonstrational video is available in our repository.

[4] See https://github.com/Zokrates/ZoKrates/releases
[5] See https://bpmn.io/toolkit/bpmn-js/

C. Functional testing

We assembled a suite of *simple test cases*, based on the test model suites of the tools cited in Section III. BPMN model size and complexity influence zkWF program size and complexity, which, in turn, determine proof computation times and on-chain verification costs. To evaluate the practical feasibility of our approach, the leasing model from Section II was used as our *representative test case*.

D. Performance evaluation

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).

In Ethereum, smart contract execution steps, measured in "gas", incur a cryptocurrency cost, paid by the transaction-requesting user. For measurements of gas used, we used a private, one-node Ethereum test network with version 1.10.25 of geth, the official Go implementation of the Ethereum protocol. Blockchain-side efficiency measurements are largely irrelevant for Hyperledger Fabric, which has no "gas" notion and where the smart contract execution layer is highly resource-scalable. Table I summarizes the relevant size metrics of our test cases.

TABLE I
BPMN MODEL AND TEST CASE CHARACTERISTICS

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 II summarizes the runtimes of the off-chain computations. Compilation and zk-SNARK setup were executed once; proving time is the sum of computing the witness and generating the proof, and we give an average over the scenarios. The measurements indicate that our approach is practically feasible for real-life models.

TABLE II
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 III summarizes the gas costs of smart contract deployment and 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 driven by the size of the encrypted version of the current state. In the measurements, we use state cleartext instead of ciphertext to eliminate the impact of the not-constrained encryption.

TABLE III
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

Due to the novelty of our approach, it is comparable with the state of the art only in gas costs. Deployment is on par with, or is better than, the existing solutions. However, the cost of updating the state is significantly higher; ChorChain uses about 92,905 gas on average for each message and Caterpillar is similar to ChorChain.

This ”confidentiality premium” is certainly not acceptable on the Ethereum mainnet. Still, it can be argued that the high gas price on the mainnet has ”priced out” all use cases that were not strictly crypto-financial years ago. On the other hand, at the time of this writing, on multiple alternative EVM-based public blockchains, the gas costs of our operations translate to fractions of 1 USD. Additionally, our approach has evident usage potential on purpose-created, permissioned, cross-organizational blockchains; in this case, the gas cost is a technical consideration and low enough to allow for dozens of transactions per block under customary block gas targets. Lastly, we store encrypted state on-chain ”only” to fulfil requirement A3 the simplest way; highly available off-chain data storage with blockchain-based integrity assurance is a common technique.

IX. THREATS TO VALIDITY AND FUTURE WORK

We see compliance with BPMN operational semantics as a non-negligible threat to validity, especially after our planned future extension of the supported BPMN subset. For the approach presented in this paper, we only tested compliant behavior and not formally prove it; this remains future work.

Impractical proof time for much larger BPMN models is also a threat. We plan to introduce the capability to handle *hierarchical* process models. We expect that we can instantiate orchestrator smart contracts for sub-processes in a way that coordinates the commitment-management across the levels, but controls proof obligation complexity by requiring proof generation only for a limited-size model part for each update.

While the ring schedule ”fake updates” approach is evidently correct for adhering participants (and, we surmise, for mostly adhering participants), side-channel protections is an open line of research. We plan to analyse the ring schedule scheme under various participant failure models and compare it with delay randomization schemes. Metrics for measuring the guaranteed level of protection through fake updates are necessary, too. Differential privacy metrics worked out for publicly observable messaging settings with a ”hide-in-the-crowd” approach similar to ours [28] promise to be adaptable.

Lastly, we note that there are stronger versions of our collaboration confidentiality model through additional *inter-collaborator confidentiality constraints*; it is an interesting question how our approach can be extended to such settings.

X. CONCLUSION

In this paper, we presented a collaboration 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, first-of-its-kind solution, which we validated functionally 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).

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, pp. 1084–1091. DOI: 10.1109/Cybermatics 2018.2018.00199.
[7] C. Cachin and M. Vukolić, “Blockchain consensus protocols in the wild,” in *31st International Symposium on Distributed Computing (DISC 2017)*, 2017. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2017/8016/pdf/LIPIcs-DISC-2017-1.pdf
[8] I. Gorton, *Essential Software Architecture*, 2nd ed. Springer Berlin, Heidelberg, 2011.
[9] 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, pp. 785–790, 2021. DOI: 10.1016/j.procs.2021.03.097
[10] 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, pp. 1162–1193, 2019. DOI: 10.1002/spe.2702
[11] 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. pp. 1215–1220 DOI: 10.1109/IEMCON.2018.8614879.

[12] 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. LNCS, 2020. pp. 325–341, DOI: 10.1007/978-3-030-41568-6_21.

[13] 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, pp. 1059–1079, 2021. DOI: 10.1002/spe.2931

[14] 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.

[15] ——, “Model-driven engineering for multi-party business processes on multiple blockchains,” *Blockchain: Research and Applications*, vol. 2, no. 3, p. 100 018, 2021. DOI: 10.1016/j.bcra.2021.100018

[16] ——, “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, pp. 25–32. DOI: 10.1145/3528226.3528369.

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

[18] ZKProof Community, “ZKProof Community Reference,” 2022. [Online]. Available: https://docs.zkproof.org/reference.pdf

[19] J. Groth, “On the Size of Pairing-based Non-interactive Arguments,” in *Advances in Cryptology – EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 2016, pp. 305–326. DOI: 10.1007/978-3-662-49896-5_11.

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

[21] A. Chiesa, Y. Hu, M. Maller, P. Mishra, P. Vesely, and N. Ward, “Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS,” Cryptology ePrint Archive, Paper 2019/1047, 2019. [Online]. Available: https://eprint.iacr.org/2019/1047

[22] D. Bennaroch, M. Campanelli, D. Fiore, J. Kim, J. Lee, H. Oh, and A. Querol, “Proposal: Commit-and-Prove Zero-Knowledge Proof Systems and Extensions,” https://docs.zkproof.org/standards/proposals, presented at the 4th workshop of the ZKProof Community, 19-29 April 2021, online.

[23] X. Sun, F. R. Yu, P. Zhang, Z. Sun, W. Xie, and X. Peng, “A Survey on Zero-Knowledge Proof in Blockchain,” *IEEE Network*, vol. 35, no. 4, pp. 198–205, 2021. DOI: 10.1109/MNET.011.2000473

[24] J. Partala, T. H. Nguyen, and S. Pirttikangas, “Non-interactive zero-knowledge for blockchain: A survey,” *IEEE Access*, vol. 8, pp. 227 945–227 961, 2020. DOI: 10.1109/ACCESS.2020.3046025

[25] M. z. Muehlen and J. Recker, “How Much Language Is Enough? Theoretical and Practical Use of the Business Process Modeling Notation,” in *Seminal Contributions to Information Systems Engineering: 25 Years of CAiSE*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 429–443. DOI: 10.1007/978-3-642-36926-1_35.

[26] T. Aivo, “Zero-Knowledge Proofs for Business Processes,” Master’s thesis, Univ. of Tartu, 2020.

[27] J. Lee, J. Choi, J. Kim, and H. Oh, “SAVER: SNARK- friendly, Additively-homomorphic, and Verifiable Encryption and decryption with Rerandomization,” Cryptology ePrint Archive, Paper 2019/1270, 2019. [Online]. Available: https://eprint.iacr.org/2019/1270

[28] I. A. Seres, B. Pejó, and P. Burcsi, “The Effect of False Positives: Why Fuzzy Message Detection Leads to Fuzzy Privacy Guarantees?” in *Financial Cryptography and Data Security*, ser. LNCS. Cham: Springer International Publishing, 2022, pp. 123–148. DOI: 10.1007/978-3-031-18283-9_7.

Balázs Ádám Toldi 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.