



Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Blockchain-based, confidentiality-preserving orchestration of collaborative workflows

Scientific Students' Association Report

Author:

Balázs Ádám Toldi

Advisor:

dr. Imre Kocsis

2022

Contents

Kivonat	i
Abstract	ii
1 Introduction	1
2 Background	3
2.1 Business process modelling with BPMN	3
2.2 Blockchain-based process orchestration	3
2.2.1 Caterpillar	6
2.2.2 Lorikeet	6
2.2.3 Chorchain	6
2.3 Privacy on blockchains, in general	7
2.3.1 Zero-knowledge proofs	7
2.3.2 Ring signatures	7
2.3.3 Mixing	7
2.4 Zero-Knowledge Proofs	7
2.4.1 Definition	7
2.4.2 A simple example	7
2.4.3 Constructions	8
2.4.3.1 zk-SNARK	8
2.4.3.2 zk-STARK	9
2.5 Zero-knowledge proofs on blockchains	9
2.5.1 Zcash	10
2.5.2 Rollup solutions	10
2.5.3 Baseline protocol	10
2.6 Zero-knowledge proof development tools	10
2.6.1 libsark	11
2.6.2 ZoKrates	11

2.6.3	Cairo	12
2.6.4	zkEVM	12
2.7	Zero-knowledge proofs for business processes	12
2.8	Takeaway	12
3	The zkWF approach	14
3.1	High-level overview	14
3.2	Modeling	17
3.2.1	BPMN Modeling elements	18
3.2.2	Extended attributes	19
3.3	zkWF program design	20
3.3.1	Model definition	22
3.3.2	Process state definition	22
3.3.3	Proving scheme	23
3.3.4	Encoding the model	23
3.3.5	ZKP framework	25
3.3.6	BPMN state change validity check	25
3.3.6.1	Checking the supplied hash	26
3.3.6.2	Proving that a state vector update is valid	26
3.3.6.3	Authorisation	27
3.4	zkWF protocol design	29
3.4.1	Security assumptions	29
3.4.2	Architecture overview	30
3.4.3	Process manager smart contract design	30
3.5	Security guarantees	32
3.6	Limitations	32
3.6.1	Limitations of BPMN models	33
3.6.2	Limitations of the proving scheme	33
3.7	Deployment and operation	33
3.7.1	Choosing a distributed ledger	33
3.7.2	Deploying the smart-contract	33
3.7.3	Stepping the execution	34
3.8	Testing	35
3.8.1	Test cases	35
3.8.1.1	Simple & corner cases	35
3.8.1.2	Representative test	38
3.8.1.3	Testing framework	38

3.8.2	Future ideas for testing	38
4	Implementation	41
4.1	Modeller	41
4.2	zkWF implementation	41
4.2.1	ZKP framework	42
4.2.2	Zokrates implementation	42
4.2.2.1	Files	42
4.2.2.2	Checking the supplied hash	42
4.2.2.3	Proving that a state update is valid	42
4.2.2.4	Authorisation	42
4.2.2.5	Verification of an encrypted ciphertext	42
4.3	Code generator	43
4.4	Zokrates wrapper	43
4.5	Smart contract implementation	43
4.5.1	EVM	43
4.5.2	Hyperledger Fabric	45
4.6	WFGUI	45
4.6.1	Modeler tab	45
4.6.2	Testing tab	45
4.6.3	Deployment and operation tab	46
4.6.4	Video presentation	47
5	Results	48
5.1	Simple & corner cases	48
5.1.1	Hardware used for testing	48
5.1.2	Software used for testing	48
5.1.3	Comparing test cases	48
5.1.4	Results	49
5.2	Complex test	49
5.2.1	Size of the model	50
5.2.2	Results	50
5.2.2.1	Model execution	50
5.2.3	Proving time	50
5.2.4	Gas usage	51
5.3	Comparison between existing solutions	51
6	Conclusion	53

Appendix	58
A.1 State vector checking	58
A.2 Hasing ZoKrates code template	60
A.3 The root ZoKrates code	60

Kivonat

Ez a dolgozat egy új, tudásmentes bizonyítékon alapuló megközelítést mutat be az üzleti folyamatokon alapuló együttműködések okos szerződésen alapuló összehangolására. A meglévő megközelítésekkel ellentétben az én megoldásom modellalapú, és nem tárol egyidejűleg értelmezhető adatokat a blokkláncon. (Vagyis a folyamatban részt nem vevő felek nem ismerhetik meg a folyamat állapotát).

A független felek közötti együttműködés kihívást jelenthet, különösen akkor, ha nem bíznak meg teljesen egymásban. Megközelítések születtek az együttműködési tevékenységek nyomon követésére, valamint az együttműködésben részt vevő felek tevékenységének kikényszerítésére és engedélyezésére blokkláncban üzemeltetett okosszerződések segítségével. Léteznek olyan megoldások is, amelyek automatikusan generálják az okos szerződés logikáját az együttműködés és a folyamatvégrehajtás modelljeiből, amelyeket például a BPMN-ben (Business Process Model and Notation) rögzítenek. Ezzel egyidejűleg gyorsan fejlődnek az okosszerződésekben a tudásmentes bizonyításon alapuló technikák alkalmazására szolgáló technológiák, amelyek célja a kriptográfai nem védett érzékeny adatok blokkláncon belüli tárolásának szükségességének enyhítése. Az együttműködési adatok védelme a modellalapú okosszerződésekben azonban újszerű felvetés, és ez képezi e munka témáját.

Munkámban először definiálom az általam választott BPMN modellezési nyelv elemeinek egy részhalmazát. Meghatározok továbbá egy kiterjesztést is ehhez a halmazhoz, hogy megragadjam a tudásmentes környezetben való futtathatósághoz szükséges tulajdonságokat.

Ezen az alapon definiálok egy transzformációs logikát a BPMN-ből a ZoKrates eszközkészlet bemeneti nyelvére. A ZoKrates képes tudásmentes bizonyítók és (okos szerződés alapú) verifikátorok generálására számítások széles körére. A számítási sablonom egy üzleti folyamat megengedett állapotátmeneteit és egy nyilvánosan tárolt titkosított állapot és hash commitment frissítéseit rögzíti. Ismertetem a prototípus implementációm is.

Bemutatok egy folyamatkezelő okos szerződést is, amely nyomon követi az üzleti folyamatok végrehajtásának aktuális állapotát. Ezen okos szerződés ellenőrzi a tudásmentes bizonyítékokat, mielőtt a blokkláncon történő változtatásokat engedélyezné. A ZoKrates által támogatott EVM-szerződések generálásán túlmenően ezen okos szerződések létrehozására szolgáló lehetőségeket implementáltam a Hyperledger Fabric-ra, mint alternatív blokkláncplatformra.

Ezt a módszert megvalósítottam és teljes mértékben integráltam egy eszközbe. Ez az eszköz tartalmaz egy modellezőt, egy résztvevői oldali SDK-t, egy pénztárcakezelőt és egy egyszerű vizuális felületet.

A megközelítem validálása jelenleg tesztelésen alapul, amelyhez létrehoztam egy tesztkészletet. A jövőbeni munka lehetséges irányaként megvizsgálom a BPMN működési szemantikájának való megfelelést más eszközökkel.

Abstract

This paper describes a novel, zero-knowledge proof-based approach for the smart-contract-based orchestration of business process-based collaborations. In contrast to existing approaches, my solution is model-based and does not simultaneously store meaningful data on the blockchain. (That is, parties not involved in the process cannot get to know its state).

Collaboration between independent parties can be challenging, especially if they do not have complete trust in each other. Approaches have been proposed for tracking collaboration actions and enforcing and authorizing parties in a collaboration to perform activities via blockchain-hosted smart contracts. Solutions also exist to automatically generate the orchestrating smart contract logic from models of collaboration and process execution importantly captured, e.g. in BPMN (Business Process Model and Notation). At the same time, technology for applying zero-knowledge proof-based techniques in smart contracts to alleviate the need to store cryptographically not protected sensitive data on-chain has been maturing rapidly. However, protecting collaboration data in model-based smart contracts is a novel proposition and forms the topic of this work.

In my work, I first define a subset of the BPMN modelling language elements I chose to use. I also define an extension to this set to capture properties necessary to be executable in a zero-knowledge environment.

On this basis, I define a transformation logic from BPMN to the input language of the Zokrates toolkit. ZoKrates can generate zero-knowledge provers and (smart contract-based) verifiers for a broad family of computations. My computational template captures permissible state transitions of a business process and updates to a publicly stored encrypted state and hash commitments. I also describe my prototype implementation.

I also introduce a process manager smart contract that keeps track of the current state of business process executions. This smart contract verifies the zero-knowledge proofs before allowing changes on the blockchain. In addition to the Zokrates-supported EVM contract generation, I implemented facilities for generating these smart contracts for Hyperledger Fabric as an alternative blockchain platform.

I implemented this approach and fully integrated it into a tool. This tool includes a modeller, a participant-side SDK, a wallet manager, and a simple visual interface.

Validation of my approach is currently based on testing for which I created a test suite. As a potential avenue for future work, I investigate the potential approaches for assuring conformance to BPMN operational semantics via other means.

Chapter 1

Introduction

In modern business science, process-focused thinking about internal activities and external collaborations is a very important tool for controlling and improving key performance indicators – through controlling and improving the *processes* of the business. While by today *Business Process Management* (BPM) became a mature and very broad discipline with many aspects [30], explicitly modelling processes to facilitate analysis and automation, as well as supporting process execution with purpose-built IT solutions remain central to BPM.

Automating the execution of business processes (originally, and much more narrowly in this domain: *workflows*) has been supported for a long time by a range of Workflow Engines and Workflow Enactment Services [27]. The leading business process modelling standard, BPMN 2.0, is certainly amenable to serve as a process definition to such, typically centralized, tools [7], but BPMN is not tied to the established automation approaches and tooling – the same way as UML or SysML is not tied to a specific code generator tool, especially so that formal BPMN model and execution semantics exist (either as a part of the standard, or in addition to it).

Blockchains have been recognised early 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 [25]. Smart contracts can be used to

- enforce, and irrevocably and irrepudiably track the sequences of activities to be performed and performed by participating organizations in process instances (that is, process states and state transitions);
- irrevocably and irrepudiably store sent and received messages – or anchor them via cryptographic (hash) commitments;
- and host data objects worked on by a process, or anchor their changes via cryptographic commitments.

Additionally, depending on the flavor of BPMN used, intra-organizational traces can also be captured on-chain, which can alleviate the need for a range of manual audit obligations across organizations. In the following, we will refer to these capabilities in general as *blockchain-based process orchestration*.

However, these technologies are still largely nascent and the privacy and confidentiality aspects have not yet been sufficiently addressed.

In my paper, I present the following new results.

- A novel construction for encoding state updates of BPMN collaboration instances as ZoKrates programs, from which zero-knowledge provers and checkers can be constructed; regarding the latter, and notably, smart contract based ones.
- A novel protocol for zero-knowledge blockchain-based business process orchestration.
- An end-to-end framework implementation.
- Empirical demonstration of the viability of the presented approach and implementation.

The source code of my work is available at this link¹. In the future, I plan to release it under a free license as open-source software.

¹https://bmeedu-my.sharepoint.com/:u:/g/personal/balazs_toldi_edu_bme_hu/EW06Rx-GNDBOr9otdeDF0j0Bi51eL2TUg3kawzgpUvSwQ?e=NBQOCU

Chapter 2

Background

In this chapter, I briefly introduce the background of the key components in my research. First, I review the state of the art of blockchain-based process orchestration. Then I introduce Zero-Knowledge Proofs (ZKPs), their current applications in the general blockchain context and I offer a brief overview of current best of breed ZKP tooling. I also discuss the very limited, openly accessible application of ZKPs towards business orchestration privacy and confidentiality that I am aware of.

2.1 Business process modelling with BPMN

Business processes can be modelled in a variety of ways. The BPMN (Business Process Model and Notation) standard is one such method. Its purpose is to specify business process models graphically. The 2.0 [16] version was released in 2011 by OMG. It defines the notation of *process*, *collaboration*, and *choreography* diagrams.

Processes describe a set of Activities in a sequence to carry out work. It is visualised as a graph of Activities, Events, Gateways and Sequence flows.

A *choreography* is a type of process. Their primary focus is on the interaction between the participants rather than the orchestration of the work performed. It formalises the way participants exchange information (Messages).

The BPMN specification states that a *collaboration* is a set of participants with corresponding pools. Pools **may** include processes or choreographies. Collaborations may also include message flows between the pools. They represent communication between the modelled participants.

The standard also defines the set of elements that can be used to model business processes. These are organised into separate categories. The five basic categories of elements are Flow Objects, Data, Connecting Objects, Swimlanes, and Artifacts. The main graphical elements that describe the behaviour of a business process are the Flow Objects. There are three types of Flow Objects: Events, Activities, and Gateways. Figure 2.1 showcases a simple process diagram (a "one-participant collaboration").

2.2 Blockchain-based process orchestration

On most public and permissionless blockchains that allow smart contract deployment, the developer cannot update the contract after it is deployed. This issue has raised the

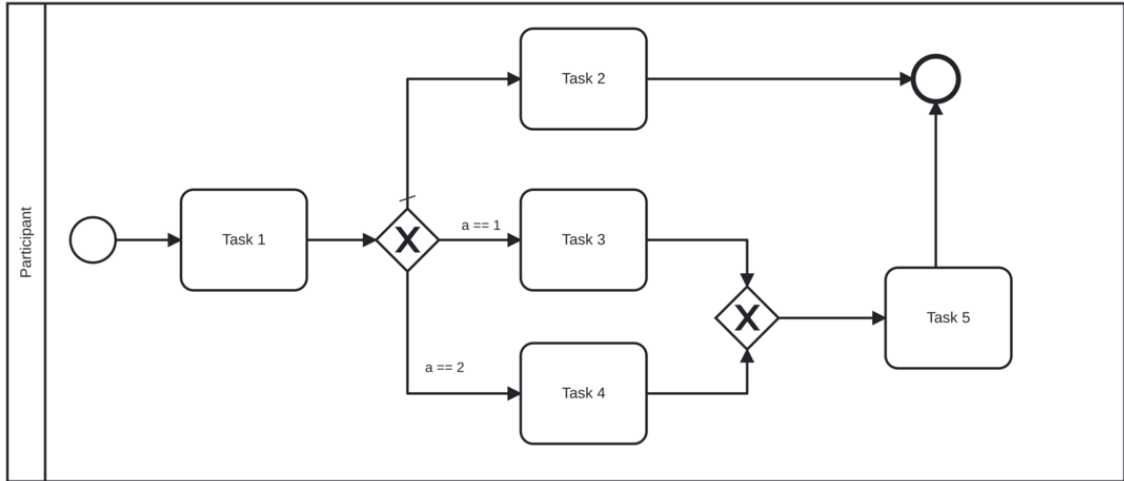


Figure 2.1: example of a BPMN 2.0 model

question of how to make smart contract development more reliable. One approach – and research-wise, the dominant approach – is using such Model-Driven Engineering (MDE) techniques, where a model (e.g., BPMN) serves as a *specification* and smart contract logic is generated automatically.

The current relevant state of the art has been summarised in a survey [4]. The survey covers BPMN and UML based approaches for multiple Distributed Ledger systems.

The survey points out the advantages and limitations of each approach with various criteria. It also reveals that the examined approaches do not support data privacy and confidentiality when dealing with permissionless blockchains. Table 2.1 summarises the surveyed tools.

Table 2.1: Survey of MDE for blockchain-based process orchestration (source of the table: [4])

Name	Modeling tech.	Contribution	Limitation
Rocha et al. [28]	UML, BPMN, ERD	Explore current modeling languages	Not an extension of a language
Mavridou et al. [24]	WebGME	SC Modeling, security extensions, formal modeling	Limited modeled elements, no off-chain modeling
Lopez-Pintado et al., [22]	BPMN	On-chain BPMS and modeled elements	No access control, no full support of oracles
Mercenne et al., [26]	BPMN	Extension of caterpillar with access control	No full support of oracles

Continued on next page

Table 2.1 – continued from previous page

Name	Modeling tech.	Contribution	Limitation
Silva et al., [29]	DEMO	Meta-model of DEMO and HLF concepts	No code generation, no off-chain components modeling
Hornkov et al., [19]	DEMO	Highlights importance of interaction with off-chain	No code generation, no off-chain components modeling
Ladleif et al., [20]	BPMN Choreography	Extension of BPMN 2.0 choreography diagrams	No modeling of escrow, or oracles
Corradini et al., [10]	BPMN Choreography	Translation of BPMN choreography to SC, dApp lifecycle	No full support of oracles
Weber et al., [31]	BPMN Choreography	Triggers as a communication method for on- and off-chain	Triggers not modeled, no full support of oracles
Marchesi et al., [23]	UML	Extension of UML, agile dApp development method	No code generation, no full support of oracles
Hamdaqa et al., [17]	Domain specific lang.	Unified reference model for SC of multiple blockchains	No code generation, no off-chain components modeling
Garamvolgyi et al., [12]	UML Statecharts	Modeling of SC as a state machine	No code generation, no off-chain components modeling
Lu et al., [21]	BPMN	SC interfaces and tokens model, off-chain communication	No access control policies, no full support of oracles
Babkin et al., [3]	ArchiMate	Automatic translation between ArchiMate and HLC	Manual work before SC deployment

Continued on next page

Table 2.1 – continued from previous page

Name	Modeling tech.	Contribution	Limitation
Boubeta-Puig et al., [6]	BPMN and EMF	Integration of CEP with Ethereum Platform	No full support of oracles

In my own review of the state of the art, three tools emerged as the most mature and most relevant to my research: Caterpillar, Lorikeet and Chorchain. These provide valuable templates for future research, but in the end, due to the core difference from my own approach, I could not reuse them.

2.2.1 Caterpillar

Caterpillar [22] is the first open-source BPMN-to-Solidity compiler. Its primary purpose is to execute collaborative business processes between mutually untrusting parties on blockchains. It supports a large variety of elements of the BPMN 2.0 specifications.

Since its initial release, several forks have emerged. Some of these also come with an extended feature set, like Blockchain Studio [26], which adds role management, or Amal et al. [1], which adds time constraints.

2.2.2 Lorikeet

Lorikeet [21] is a model-driven engineering approach which integrates assets into business processes. For modelling, this method extends the BPMN 2.0 specification to support the modelling of asset registries (e.g., for fungible/non-fungible asset registration, escrow for conditional payment, and asset swap). These models are then transformed into a Solidity smart contract by this tool. It handles the orchestration of the process and the interactions with the tokens.

2.2.3 Chorchain

Chorchain [10] is a tool that takes a BPMN choreography and generates an Ethereum smart contract that can be used to execute the model. Chorchain also has a modelling tool to create models. These models can be used to create instances. Instances can be configured in different ways (e.g. different participants). Later they can be used to generate and deploy a smart-contract representation of the instance. After it is deployed, the execution can be done in the tool or by manually interacting with the smart contract.

The same authors also released two additional studies related to this paper: Multi-chain [8] and FlexChain [9]. Multi-chain is similar to Chorchain. However, Multi-chain is also capable of generating chain code for Hyperledger Fabric. 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.

2.3 Privacy on blockchains, in general

Due to many blockchains' public and permissionless nature, data stored on-chain can be read by any party. To overcome this issue, various solutions have appeared.

2.3.1 Zero-knowledge proofs

One general way to achieve anonymity on public distributed ledgers is to use zero-knowledge proofs (see section 2.4.1 for more details). They can hide the origin (the sender), the receiver and contents of transactions.

2.3.2 Ring signatures

Another way to mask the authors behind a transaction is to use Ring signatures. A famous example is Monero, a blockchain system with privacy-enhancing features, including ring signatures, zero-knowledge proofs and "stealth addresses".

2.3.3 Mixing

Cryptocurrency mixing services obfuscate the origins of transactions, so the sender stays anonymous. This is usually done by mixing many incoming funds into a large pool and periodically spitting them to their desired destination.

2.4 Zero-Knowledge Proofs

Zero-Knowledge Proofs are cryptographic methods to prove that various statements are true – without revealing any additional information about the statement.

2.4.1 Definition

A zero-knowledge proof [32] (ZKP) 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 a proof. 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).¹

2.4.2 A simple example

Suppose Alice and her friend Bob want to play a game of 'Where's Wally'. Alice claims that she found Wally on one of the puzzles. She (the prover) wants to prove that what she claims is true to Bob (the verifier).

There is an obvious way to do this. She can point to Wally's location on the puzzle. In this case, Bob will be sure that Alice's claims are valid. However, Alice would be upset because Bob now knows where Wally is.

¹Source: [32] Section 1.1.1

For a zero-knowledge approach, Alice needs a paper at least twice the puzzle size. Then, she needs to cut a Wally-shaped hole in the middle of it. She then tapes the paper to the puzzle, only revealing Wally in the small hole cut previously.

This way, Bob will be convinced that Alice knows where Wally is without telling Bob where he is.

2.4.3 Constructions

Zero-knowledge proofs can be built on top of NP problems. In fact, Goldreich et al. [13] prove that all languages in NP have zero-knowledge proofs. For the practice, the currently two most important families of ZKP constructions are zk-SNARKs and zk-STARKs.

2.4.3.1 zk-SNARK

The acronym zk-SNARK stands for Zero-Knowledge Succinct Non-Interactive Argument of Knowledge. It describes the properties of the protocol.

Zero-Knowledge The protocol is "Zero-Knowledge", as described in section 2.4.1.

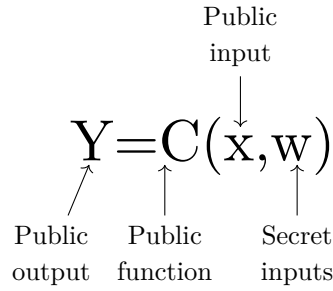
Succinct This means that proofs stay relatively small (i.e. a few hundred bytes), even for extensive programs, and they can be verified in a short amount of time (i.e. milliseconds).

Non-Interactive The proof consists of a single message from the prover to the verifier. No other messages can be sent to either party.

Argument of Knowledge The proof not only says that the statement is true, but the prover also knows why it is true.

Problems with zk-SNARK zk-SNARKs require a trusted setup: In this phase, a party (generator) generates the proving and the verifying key, but in the process, it generates "toxic waste" as well. With this toxic waste, anyone could generate "fake" proofs. For this reason, the setup is "trusted" because the prover and the verifier must trust that the generator keeps the toxic waste secret (or removes it). This shortcoming is usually addressed by multiparty trusted setup "ceremonies".

Figure 2.2 shows how the setup, proving, and verification activities work.



Setup: $(C, \cancel{w}) \rightarrow (pk, vk)$

Prove: $(pk, x, w) \rightarrow \text{Proof}$

Verify: $(vk, \text{Proof}, x, Y) \rightarrow \text{Bool}$

Figure 2.2: zk-SNARK setup, proving, and verification activities based on a computation definition

zk-SNARKs in DLT systems zk-SNARKs are relatively popular in public and permissionless blockchain systems since their proofs tend to be small and computationally inexpensive to check (hence "succinct"). Ethereum even has native support for verifying proofs making it cheaper to perform.

Proving schemes There are now many different zero-knowledge proving schemes. One of the first practical ones was Groth16 [14], developed in 2016 by Jens Groth. It formalised a proving system that significantly improved performance, making it possible to use for real-world applications. GM17[15] is an improved version of Groth16. It was developed by Jens Groth and Mary Maller in 2017. It improved security and efficiency over Groth16.

2.4.3.2 zk-STARK

zk-STARK stands for Zero-Knowledge Scalable Transparent Arguments of Knowledge. It is an advanced zero-knowledge protocol construction that does not require a trusted setup. It is a relatively new (released in 2018 [5]) protocol, so there are few tools for it. They are less popular because their proofs are larger, requiring more computation to verify them. The latter also means they are less suitable for public blockchains (e.g., on Ethereum, verifying them consumes more gas).

2.5 Zero-knowledge proofs on blockchains

Zero-knowledge proofs are widely used along with blockchain technologies. These methods are utilised to increase privacy and confidentiality and to reduce the cost of several transactions.

2.5.1 Zcash

Zcash [18] was the first real use-case for zero-knowledge proofs in blockchains. It allows the users to stay anonymous and keep their balance private using zero-knowledge proofs. Originally, ZCash used a zk-SNARK-based Groth16 proving system, but a few years ago, they transitioned to a more modern proving system they developed called Halo 2. Halo 2 also uses zk-SNARK proofs.

2.5.2 Rollup solutions

The second largest (by market capitalisation) cryptocurrency Ethereum has reached the network's current capacity. To overcome this issue, many "Layer 2" solutions have emerged. Generally speaking, these solutions handle transactions off Ethereum ("off-chain") and submit batches of them periodically to the "Layer 1" blockchain. This makes transactions much cheaper and faster at the price of the additional layer.

According to Ethereum's official website, the roll-up solutions are currently the preferred layer 2 scaling methods. Rollups bundle hundreds of transactions into one on the mainnet layer 1. There are two types of roll-up methods: optimistic and zero-knowledge roll-ups. Optimistic roll-ups assume that every transaction submitted is valid. However, there is a time frame where people can claim that a transaction is malicious. If it turns out that the transaction was, in fact, malevolent, the transaction is cancelled, and the person who reported this act gets a reward.

On the other hand, zk-rollups generate a zero-knowledge proof for a batch of transactions and submit the proof to the mainnet. This technique ensures that only valid transactions are uploaded to layer 1, making this approach secure and scalable. Unlike optimistic roll-ups, zk-rollups are currently "application specific" layer 2 solutions, meaning they do not provide the full capabilities of Ethereum. That is, these approaches can be used for payments and token transfers, but they cannot be used for deploying any custom smart contract.

2.5.3 Baseline protocol

The Baseline protocol² is a developing open standard that allows enterprises to synchronise complex, multi-party business processes on distributed ledger technologies. Their business process workflows are formed as state machines. The standard includes some essential and optional privacy-related requirements. The protocol has two reference implementations, but, at the time of this writing, neither of these has the privacy and confidentiality measures I enabled in my research.

2.6 Zero-knowledge proof development tools

Theoretically, anyone could create zero-knowledge proof systems by hand, but it is a very complicated and tedious job. It is the equivalent of writing programs in assembly. For this reason, several development tools were created.

²<https://docs.baseline-protocol.org/>

2.6.1 libsnark

libsnark³ is a popular low level library written in C++ for creating zk-SNARK applications. It is used by other applications, including other ZKP frameworks. In theory, it could be used directly for blockchain technologies, but creating a verifier smart contract for it would be difficult.

2.6.2 ZoKrates

ZoKrates [11] is a toolbox for zk-SNARKs on Ethereum. It provides a high-level programming language for creating zero-knowledge proofs. These proofs can then be verified by a command-line application or a verifier smart contract, automatically generated by ZoKrates. ZoKrates has its own programming language for specifying computations, their private and public inputs and outputs. Listing 2.1 provides a simple example.

```
def main(private field a, field b) {  
  assert(a * a == b);  
  return;  
}
```

Listing 2.1: Example ZoKrates code to prove knowledge of square root of b

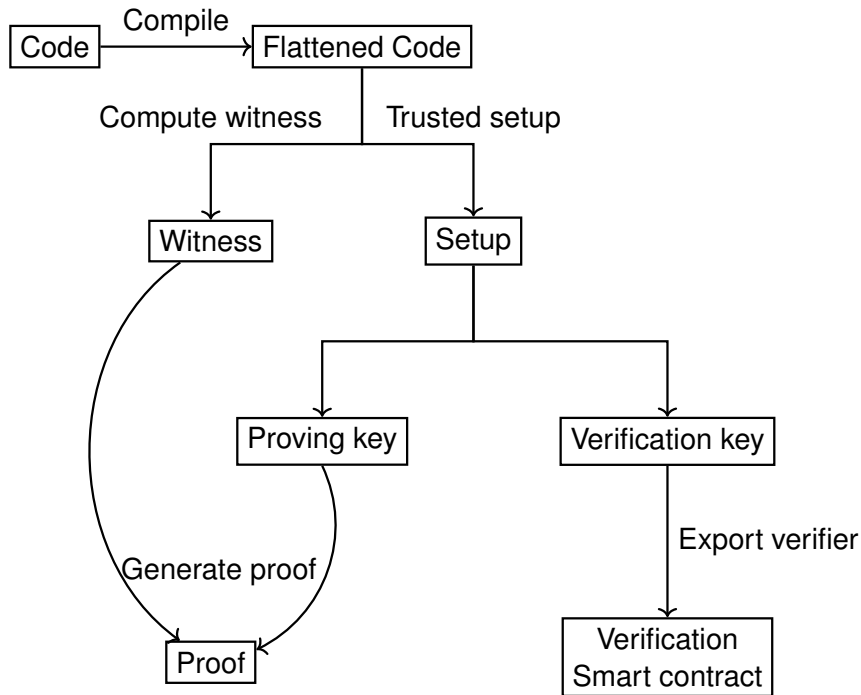


Figure 2.3: ZoKrates process

Figure 2.3 demonstrates the usage of the ZoKrates toolbox. First, the developer specifies a ZoKrates program in the high-level language. Then, it can be compiled into flattened code. With the flattened code, the user performs the trusted setup phase, which generates the proving and verification keys. When flattened code is executed, ZoKrates generates a witness of it. The prover generates the zero-knowledge proof using this witness and the

³<https://github.com/scipr-lab/libsnark>

proving key. The user can export a Solidity-based Ethereum verification smart contract with the verification key.

ZoKrates supports multiple proving schemes (including G16 and GM17), pairing-friendly elliptic curves, and proving backends.

It also comes with a feature-rich standard library, which supports hash functions, elliptic curve cryptography operations, and other utilities for type conversion.

2.6.3 Cairo

Cairo⁴ is a high-level-like programming language developed by StarkWare. It can be used to create smart-contract for StarkWare's ZK roll-up solution StarkNet. Their concept is very similar to ZoKrates, but it is highly dependent on their proprietary scaling solution.

2.6.4 zkEVM

The term zkEVM refers to a program that can generate zero-knowledge proofs for executing EVM opcodes. This is a fast-developing technology, but it is still in its early stages. Its primary purpose is to make Ethereum more scaleable, but it can also offer a lot more privacy for the users.

Currently, the two most mature zkEVM solutions are zkSync⁵ and Polygon Hermez⁶. These can provide a much cheaper way of executing smart-contract calls. However, they are not entirely EVM compatible, and their main focus is scalability. Privacy measures are currently not implemented.

2.7 Zero-knowledge proofs for business processes

Aivo et al. [2] is the only research paper that describes a technique that allows generating zero-knowledge proofs for checking the validity of traces in a BPMN model. Its primary focus is on proving that specific steps were made in a given order in the trace.

Conceptually, this result is similar to my goal, but adapting it for blockchain-based verification and step-by-step model enforcement on-chain proved to be far too involved in comparison to the approach I finally formulated and propose here.

2.8 Takeaway

There are several ways to model business processes and workflows. One of them is the use of the BPMN 2 standard. Many centralized tools use this specification, and several support the orchestration of these models' execution.

Much research has been published about how similar workflow engines could be implemented on a blockchain basis. Most approaches involve model-driven smart contract development, but none focus on privacy and confidentiality.

⁴<https://www.cairo-lang.org/>

⁵<https://docs.zksync.io/zkevm/>

⁶<https://polygon.technology/solutions/polygon-zkevm>

Zero-knowledge proofs provide a reliable way of proving that a given statement is true while avoiding the need to share more information other than the fact that the statement was, in fact, true. There are many ways to generate and use these proofs in certain areas.

Zero-knowledge proofs are increasingly widely used in the public and permissionless blockchain spaces. This technique provides a way to achieve superior privacy and confidentiality and significantly better scalability – at the expense of off-chain proof computation obligations.

Several tools are capable of generating zero-knowledge proofs. Some give the user a low-level control of the process, but some provide relatively easy usage.

So far, no blockchain-based methods have been implemented or designed to orchestrate business processes in a privacy and confidentiality preserving way. This paper aims to fill this gap.

Chapter 3

The zkWF approach

In this chapter, I present the zkWF ("zero knowledge WorkFlow") approach: a framework for blockchain-based business process orchestration, where zero knowledge proofs render on-chain stored process state and data information undiscoverable to parties not involved in the execution of the process instance. The framework consumes a representative subset of BPMN collaboration diagrams as input.

The zkWF approach relies on two key conceptual components: the zkWF protocol and zkWF programs.

The **zkWF protocol** is a hash commitment style protocol that allows the participants to follow and step the execution of the business process with a smart contract. Meanwhile, the current state of the workflow execution cannot be determined from the state stored in the smart contract.

A **zkWF program** is a ZoKrates [11] (see section 2.6.2) program that, for a given BPMN model, can decide whether a given actor of a process instance 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 its smart contract.

3.1 High-level overview

This section gives the reader a high-level overview of how the protocol works (see Figure 3.1). The protocol requires to have a smart contract deployed on a blockchain. This smart contract contains the hash of the current state and an encrypted version of the state.

Since we keep track of a business process collaboration, it likely has more than one participant. These parties can send messages to each other by off-chain means. The state contains the event of sending and receiving these messages and the hash of these messages to ensure that the receivers can verify them.

For a participant to update the state stored in the smart contract (the hash and the cyphertext), they have to create a zero-knowledge proof that the state transition they propose is valid. This new state includes the hash of the message they sent beforehand.

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.

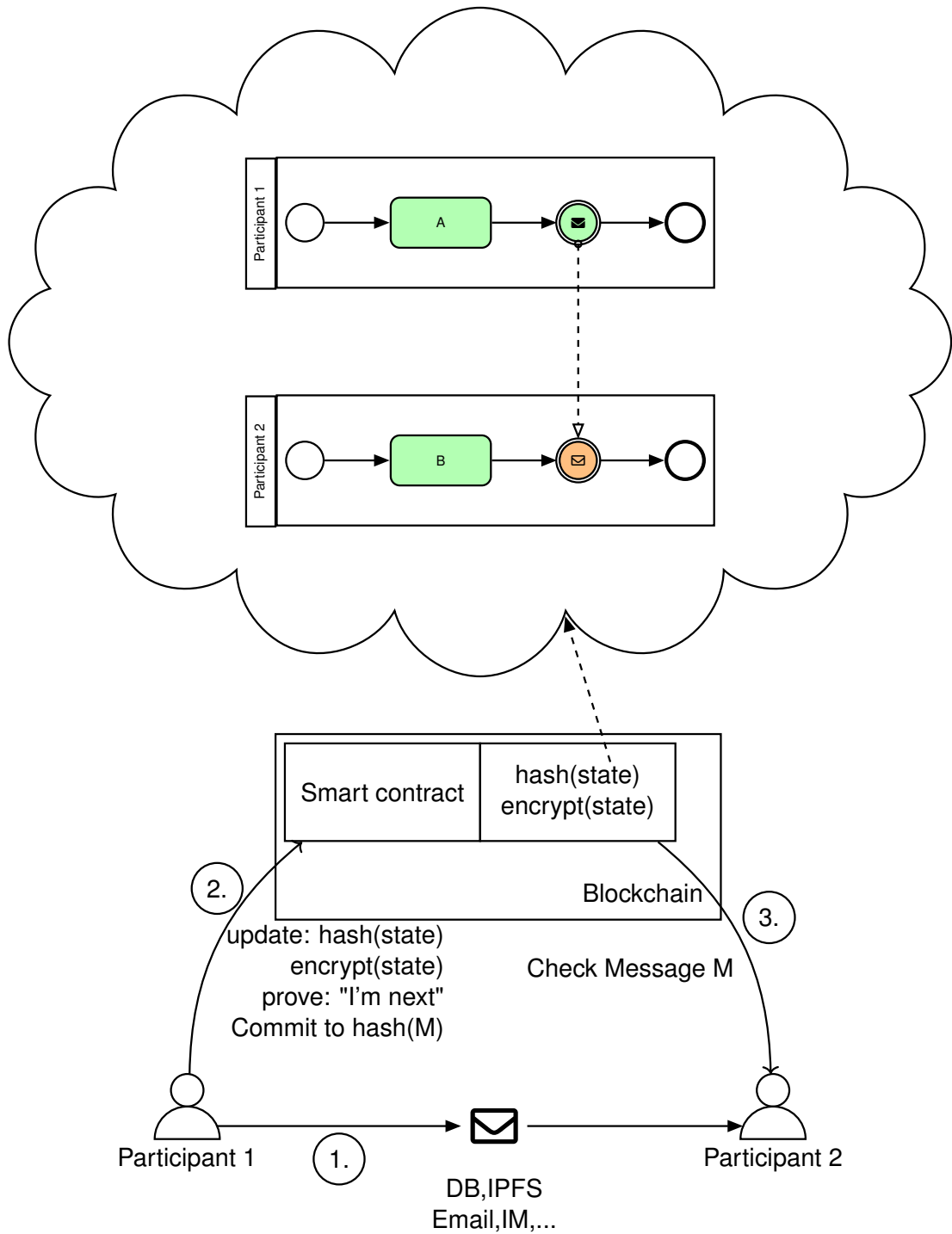


Figure 3.1: Basic Overview of the zkWF protocol

The remainder of this chapter is separated into four main parts. Figure 3.2 gives an overview of my contributions and the implemented toolchain, delineating my contributions, annotated with the respective.

The first part (section 3.2) describes how the business processes are modelled. It details what modelling language I used, what parts of it I support, and how to approach them. The second part introduces the BPMN-derived zkWF construction. The third part describes the zkWF protocol.

The second part describes how a zkWF program works, what it does, and how it is generated by synthesizing a business process model.

The third part (starting from section 3.4) introduces the details of the zkWF protocol. First, it describes the architecture of the program. Then it describes the process manager smart contract design and how participants can interact with it.

This chapter's fourth and last part presents how I tested this approach. It also describes the test suit I designed for this tool.

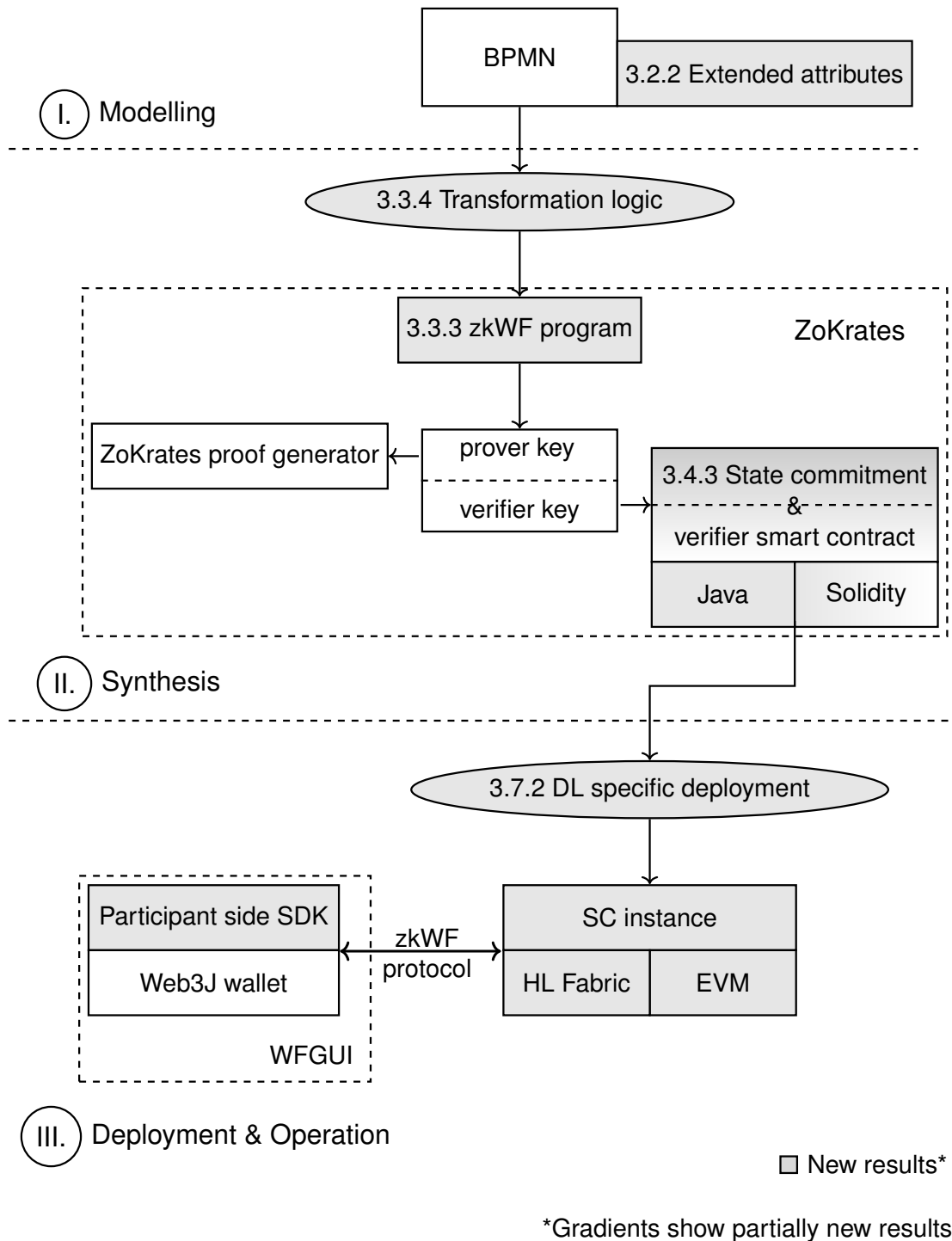


Figure 3.2: Overview of this paper

3.2 Modeling

This section describes how the business processes are modelled for the zkWF Program.

3.2.1 BPMN Modeling elements

In this paper, my main focus is on BPMN *collaboration* models. According to the specification, they could contain *processes* or *choreographies*, but I decided to only work with *collaborative processes*.

Only a limited set of elements from the BPMN specification are currently supported. Some of the element types are considered executable. Others are there to control the execution flow. Executable elements' (like that of activities) state is tracked by this tool.

I wanted to support at least the Basic Modeling Elements of BPMN 2.0¹ to prioritize common components. I have also taken into account making modelling collaborations more executable. This is why I also included Message flows and Intermediate Message events. I chose these elements because these are the ones that are necessary and sufficient to model most of the relevant use cases.

The currently supported elements can be seen in table 3.1.

¹See Business Process Model and Notation, v2.0 [16], page 28.













Element name	Notation	Executable
Start event		no
End Event		no
Activity (executable)		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		no
Lane		no

Table 3.1: Supported BPMN modelling elements

3.2.2 Extended attributes

In order to capture properties necessary to be executable in a zero-knowledge environment, I needed to define a way to attach some information to existing BPMN components. To achieve this, I added extended attributes on top of the existing BPMN specification.

zkg:publicKey This attribute is used to separate the tasks of different participants. As the name suggests, it must contain the public part of the participant's EdDSA key pair. It should be applied to pools, lanes, or any executable task. Applying this attribute to an event directly or indirectly (e.g. through a pool) is mandatory.

zkp:variables This optional parameter only is applied to tasks. It indicates the declaration of a global variable and that the variable may be used in that task (e.g. changing its value). These variables are later used in expressions for exclusive gateways.

3.3 zkWF program design

The zkWF program construction is a central contribution of this paper. It is generated for each BPMN model and used to generate zero-knowledge proofs.

In general, it proves that no illegal moves were made in a business process step. It has public and private inputs and an output. Private inputs are only visible to the prover. Public inputs are visible to everyone, and it is necessary to verify the proof. Outputs are similar to public inputs, but the user does not supply these; they result from the executed program with some private and public inputs.

A participant can use the zkWF program with the following public inputs:

- $h_{current}$ - the hash of the current state and some randomness;
- S_{new} - the previous states and the current states' hashes concatenated, signed by the last acting participant.

and the following private inputs:

- $s_{current}$ - the current state of the process;
- $r_{current}$ - the randomness used in the current hash;
- s_{new} - the updated state of the process;
- r_{new} - randomness different from the current one (used for generating a new hash);
- pk - the public key of the participant;
- sk - the private key of the participant.

Figure 3.3 visualises the design of the zkWF program.

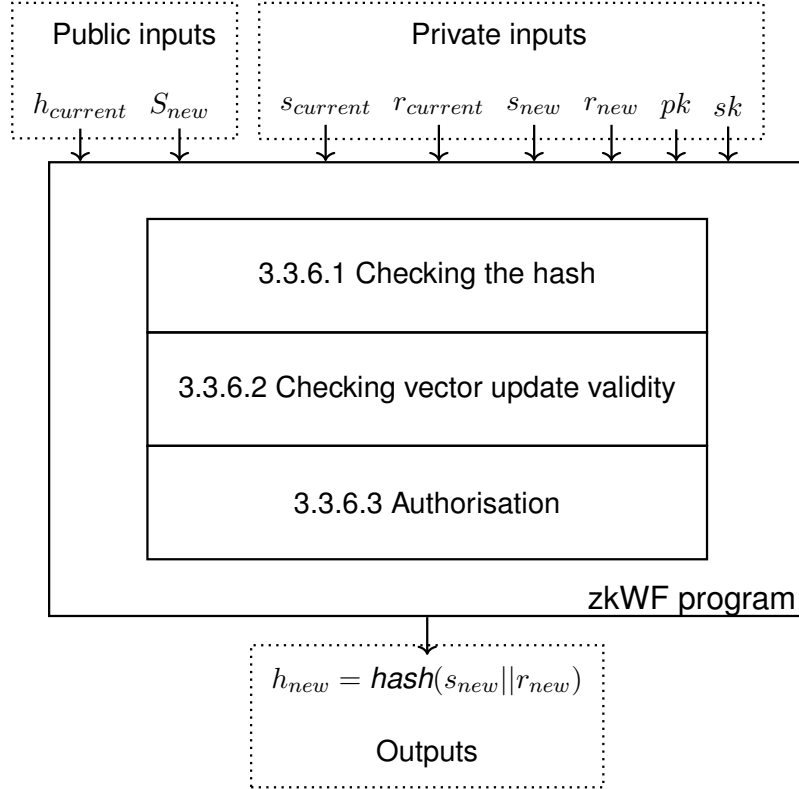


Figure 3.3: Computation model for the zkWF program

The hash of the current state ($h_{current}$) should be supplied from the process manager smart contract to the zkWF program. This ensures the integrity of the current state.

The zkWF program also checks the signature of the last acting participant. This signature must be the same as the one used as the new signature in the smart contract.

The current and the proposed new state objects ($s_{current}$ and s_{new}) and their corresponding randomnesses ($r_{current}$ and r_{new}) are used as private inputs in the zkWF program. The hash of the current state and the supplied randomness must match the hash given as public input. If the integrity of the current state is correct, the zkWF program can then check if no illegal moves were made during the proposed step.

The zkWF program also requires the participant's public and private key pair (pk and sk) to be supplied as a private input. These are used to check the signature given as public input, ensure the participant has the correct key pair, and authorize the participant for a given step.

The program outputs the hash of the new state $h_{new} = \text{hash}(s_{current} || s_{new})$.

The notations used for the parameters of the zkWF program and protocol are summarised in table 3.2.

Notation	Meaning
M	The business process model
V	The vertices of a model
E	The edges of a model
T	The executable vertices of a model
$s_{current}$	The current state
s_{new}	The new state
v	A states' state vector part
$h_{current}$	Hash of the current state
h_{new}	Hash of the new state
$r_{current}$	The current states' randomness
r_{new}	The new states' randomness
C_{curr}^{enc}	Encrypted version of the current state
C_{new}^{enc}	Encrypted version of the new state
pk	The public key of the participant
sk	The private key of the participant
$proof$	A zero-knowledge proof

Table 3.2: Parameters of the zkWF program and protocol

3.3.1 Model definition

Let the business process M to be a tuple (V, E, T) :

- V is the set of vertices and E is the set of edges
- $T \subset V$ the set of all executable events in the business process

3.3.2 Process state definition

The state of the process contains the following parameters:

- A vector v with $|T|$ elements. Each task can have three states:
 - 0 (Inactive) - The task has not been reached
 - 1 (Active) - The task can be executed now
 - 2 (Completed) - The task has been completed
- A structure of global variables
- A structure that contains the hashes of each message in the process

This state structure is generated from the model elements, as visualised on figure 3.4.

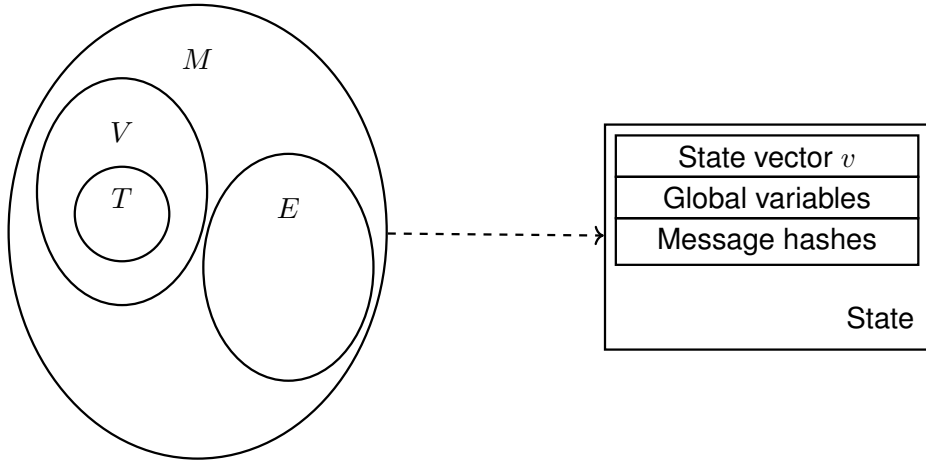


Figure 3.4: Model-based state representation

3.3.3 Proving scheme

The zkWF program needs to prove that

- hashing the current state and the randomness results in the hash given as public input,
- the proposed state update is valid in the BPMN process,
- the public key provided is authorized for this task
- and the participant own the corresponding private key.

3.3.4 Encoding the model

The zkWF program contains an encoded version of the business process model. This is done with the generation of the array P described in Aivo et al. [2] (page 25.). However, after generating this array, it is used in a very different way.

I look at business process executions as a token flow. I create a token for every start event and pass it to the first executable event connected to it. Each executable event has one incoming and one outgoing edge. When an executable event has a token, it shall be marked as "active". After completing the event, the event must be marked as "done" and pass its token to the next executable event based on the token holder's outgoing edge.

This approach can be modelled as adding a token (+1) when I mark an executable event as "active". Then I 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 the edge's arithmetic 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 the array P , I decided that an event (or gateway) can only make three token changes. This change is only necessary to make array P smaller in size. This ensures that proofs can be generated in a reasonable time frame. This limit also means a parallel

gateway start can only have two outgoing and a parallel gateway end can only have two incoming edges.

Considering these, let W be the amount of possible token changes in model M . Then take P as an array with the length of W . Each element in P is a triple of the elements of set \mathcal{N} , where

$$\mathcal{N} = (\{\text{incr, decr}\} \times T) \cup \{\text{nothing}\} \quad (3.1)$$

In general, $n \in \mathcal{N}$ is a pair of numbers describing a possible token change. The first component of the pair shows if the token is increased or decreased (+1 or -1). The second component i marks the token change for the executable event $T[i]$.

Since not every step consists of three token changes, n can also be an "empty" token change. This is used as a placeholder and is marked as $(0, -1)$.

Then, $P[i]$ shows how the process state can change in step i .

To check if the right path was chosen after an exclusive gateway, the expressions on the sequence flows after the gateways are also encoded in the program in the form of assertions. Messages passing and Variable write permissions are also encoded similarly.

Messages can only be sent in the proper intermediate message throw event and Messages cannot be sent before they were sent. Similarly, variables can only be written when permission is given to the activity currently performed.

To demonstrate how it works in practice, I chose the model shown in figure 3.5. This model was also used for testing purposes; the exact file can be found attached to this document.

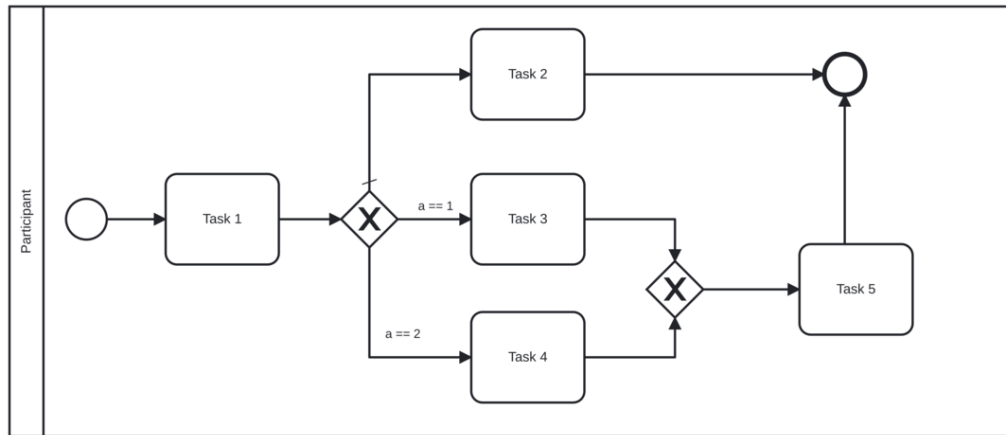


Figure 3.5: Example model

It has five executable tasks. So the set $T \subset V$ looks like this:

1. Task 4
2. Task 3
3. Task 2
4. Task 1
5. Task 5

Note that the exact BPMN file used this ordering. This model generates the array P shown in table 3.3. The variable write permissions are written as shown in Algorithm 2. Exclusive gateway validations are demonstrated in Algorithm 1.

-1	3	1	2	0	-1
-1	3	1	1	0	-1
-1	3	1	0	0	-1
-1	1	1	4	0	-1
-1	0	1	4	0	-1
-1	2	0	-1	0	-1
-1	4	0	-1	0	-1

Table 3.3: Example array P

Algorithm 1 Checking exclusive gateway path

procedure CHECKEXCLUSIVEGATEWAY

$s_{next} \leftarrow$ The new state
 $changes \leftarrow$ The changes made between the current and the new state
 $assert(changes[1]! = 3 || changes[0]! = 1 || s_{next}.a == 1)$
 $assert(changes[1]! = 3 || changes[0]! = 0 || s_{next}.a == 2)$
 $assert((changes[0]! = 3 || changes[1]! = 4) || !(s_{next}.a == 1) \& \& !(s_{next}.a == 2))$

Algorithm 2 Checking variable write permissions

procedure VARIABLECHECK

$s_{curr} \leftarrow$ The current state
 $s_{next} \leftarrow$ The new state
 $task \leftarrow$ The last activity marked as done in the new state
 $assert(task! = 2 || s_{curr}.a == s_{next}.a)$
 $assert(task! = 4 || s_{curr}.a == s_{next}.a)$
 $assert(task! = 6 || s_{curr}.a == s_{next}.a)$
 $assert(task! = 8 || s_{curr}.a == s_{next}.a)$

3.3.5 ZKP framework

To implement zkWF programs, I chose Zokrates (see 2.6.2) because this seemed like the most advanced solution at the time of writing. It also makes it easy to generate a verifier smart contract which would be tedious to write manually. I used ZoKrates version 0.7.13 at the time. It is the latest version at the time of writing.

It is necessary to mention this here because the prover method relies heavily on this toolkit.

3.3.6 BPMN state change validity check

I have introduced the inputs and outputs of the zkWF programs; described the core approach for encoding BPMN models; and specified the ZKP framework to use.

Now we are in a position to discuss the main steps of zkWF programs, as also identified on Figure 3.3. Note that as the zkWF programs are synthesized predominantly by templating,

the here described logic is essentially supplied in implementation in the ZoKrates templates in the Appendix.

3.3.6.1 Checking the supplied hash

The hash of the current state and its corresponding randomness must be the hash present in the smart contract. This step ensures the integrity of the business process execution. A pseudo-code of this procedure is shown in Algorithm 3.

Algorithm 3 Checking the supplied hash

```

procedure HASHCHECK
   $s \leftarrow$  The current state
   $r \leftarrow$  The randomness
   $h \leftarrow$  The supplied hash
   $result \leftarrow hash([s, r])$ 
  assert  $result == h$ 
  return false

```

ZoKrates's standard library contains many different hash algorithms. I chose to use sha256 because it is one of the most widely used hashing algorithms.

3.3.6.2 Proving that a state vector update is valid

This proving process step ensures that a proposed step is valid in the BPMN model.

The program compares the current and proposed (new) state's task vectors, v_{old} and v_{new} . It constructs a new matrix A in the following way:

1. At first, take matrix A as $\begin{bmatrix} 0 & -1 \\ 0 & -1 \\ 0 & -1 \end{bmatrix}$ (no changes) and $j = 0$ as a counter
2. Compare every $v_{old}[i]$ and $v_{new}[i]$, where $i \in [0, T[$
 - If $v_{old}[i] = 1$ and $v_{new}[i] = 2$, replace $A[j]$ with $[-1, i]$
 - If $v_{old}[i] = 0$ and $v_{new}[i] = 1$, replace $A[j]$ with $[1, i]$
 - If $v_{old}[i] = 0$ and $v_{new}[i] = 2$, replace $A[j]$ with $[1, i]$
 - If none of the above are true, but $v_{old}[i] \neq v_{new}[i]$, replace $A[j]$ with $[-1, -1]$ (invalid change)
 - If $v_{old}[i] \neq v_{new}[i]$ then increase j by one

After matrix A is constructed, I compare each element in the array P to matrix A . If I find an element in P that contains every row in matrix A (in any order), the vector change is considered valid.

Zero changes are also considered valid. This makes it easy to generate "fake" state changes: the process state in the smart-contract changes, but in reality, the state vector does not. This can be useful to mask the current state of the process execution. See section 3.4.3 for more details.

Four or more changes in the process state are considered invalid. The reason behind it is described in section 3.3.4.

Problem with parallel gateway ends Before a parallel gateway end, two 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 is marked as "completed", the executable event on the other end of the gateway must be marked as "active" to continue the token flow.

Variable write permission The program must ensure that the global variables can only change in the tasks that have the write permissions for that specific variable. This is done in my custom BPMN attributes described in section 3.2.2.

Exclusive gateway validation After the state's task vectors are validated, I need to ensure that the right path was chosen after the start of an exclusive gateway. This is why the arithmetic expression on the chosen edge is evaluated.

Message validation This program has a minimal message-handling protocol:

- Each time a participant wants to mark a Message Throw Event as "completed", a message hash has to be uploaded. The message should be sent directly to the participant in a secure channel.
- Each time a participant wants to mark a Message Catch Event as "completed", I need to make sure that the corresponding Message Throw Event is also marked as completed (likely by another participant). The participant should also check if the message they received has the same hash value as the one in the current state.

Pseudo code A detailed pseudo-code can be seen in Algorithm 4 and 5.

Example To make this section more understandable, I demonstrate it on an example based on the model shown in figure 3.5.

Let us say I want to take the step from Task 1 to Task 4. Then, the initial state vector would be the one shown in table 3.4, and the new state vector is shown in table 3.5.

$$0 \quad 0 \quad 0 \quad 1 \quad 0$$

Table 3.4: Initial state vector (v_{old})

$$1 \quad 0 \quad 0 \quad 2 \quad 0$$

Table 3.5: New state vector (v_{new})

Then, I generate the matrix A , as described above. The result is shown in figure 3.6. After that, I need to find an element in the array P that contains all the rows of matrix A . As you can see, the fourth element of the array P (see table 3.3) matches that.

3.3.6.3 Authorisation

To authorize a participant, the program proves that the participant has the private key, which corresponds to the task's specified public key.

Algorithm 4 Proving that a state vector update is valid

```
procedure STATECHANGE
   $v_{curr} \leftarrow$  Current state vector
   $v_{next} \leftarrow$  New state vector
   $A \leftarrow [nothing; 3]$ 
  for  $i$  in  $0..len(T)$  do
    assert  $v_{curr}[i] \leq 2$ 
    assert  $v_{next}[i] \leq 2$ 
   $changeCount \leftarrow 0$ 
  for  $i$  in  $0..len(T)$  do
    if  $v_{next} == v_{curr}$  then continue
    else if  $v_{curr} == 1$  and  $v_{next} == 2$  then
       $A[changeCount] \leftarrow [-1, i]$ 
    else if  $v_{curr} == 0$  and  $v_{next} == 1$  then
       $A[changeCount] \leftarrow [1, i]$ 
    else
       $A[changeCount] \leftarrow [-1, -1]$ 
       $changeCount \leftarrow changeCount + 1$ 
  assert  $changeCount \leq 3$ 
  if  $changeCount == 0$  then
    return true
   $result \leftarrow$  false
  for  $i$  in  $0..W$  do
     $allPairMatch \leftarrow$  true
    for  $j$  in  $0..3$  do
       $pariFound \leftarrow$  false
      for  $k$  in  $0..3$  do
        if  $A[k][0] == P[i][j][0]$  and  $A[k][1] == P[i][j][1]$  then
           $pariFound \leftarrow$  true
        if  $\neg pariFound$  then
           $allPairMatch \leftarrow$  false
          continue
      if  $allPairMatch$  then
         $result \leftarrow$  true
  // Testing for parallel gateway ends
  if  $changeCount == 1$  then
     $ParallelTest(A)$ 
  return result
```

$$\begin{bmatrix} -1 & 3 \\ 1 & 0 \\ 0 & -1 \end{bmatrix} \quad (3.2)$$

Figure 3.6: Matrix A after comparing v_{old} and v_{new}

The program also proves that the signature supplied as public input can be decrypted with that public key. The signed message has to be the hash of the previous state (with the randomness) and the hash of the new, proposed state (with a new random number). This signature is then stored in the smart contract. An EdDSa implementation is available in

Algorithm 5 Testing for parallel gateway ends

```
procedure PARALLELTEST
   $A \leftarrow$  From argument
  for  $i$  in  $0..len(T)$  do
    for  $i$  in  $0..W$  do
       $minusCount \leftarrow 0$ 
       $other \leftarrow 0$ 
       $contains \leftarrow \text{false}$ 
      for  $j$  in  $0..3$  do
        if  $P[i][j][0] == -1$  then
           $minusCount \leftarrow minusCount + 1$ 
        if  $A[0][0] == P[i][j][0]$  and  $A[0][1] == P[i][j][1]$  then
           $contains \leftarrow \text{true}$ 
        if  $A[0][0] == P[i][j][0]$  and  $A[0][1] \neq P[i][j][1]$  then
           $other \leftarrow P[i][j][1]$ 
      if  $minusCount == 2$  and  $contains$  and  $v_{next}[other] \neq 2$  then
        return true
  return false
```

Algorithm 6 Authorization of the participant

```
procedure PARTICIPANTAUTHORISATION
   $keys \leftarrow$  The array of keys generated generated for each executable state
   $step \leftarrow$  The id of the step taken
   $S \leftarrow$  The signature from the participant
   $sk \leftarrow$  The participants private key
   $pk \leftarrow$  The participants public key
   $hc \leftarrow$  The hash of the current state
   $hn \leftarrow$  The hash of the new state
  assert  $keys[step] == pk$ 
  assert  $proofOfOwnership(pk, sk)$ 
  assert  $verifySignature(pk, sk, [hc, hn])$ 
```

the Zokrates standard library. It can verify that a party has a private key with a corresponding public key and verify signatures. A pseudo-code of this procedure is available in Algorithm 6.

3.4 zkWF protocol design

This section describes how the zkWF protocol works and the smart contract design used for the protocol. The zkWF protocol certainly largely follows the general "proofs over commitments and proposed commitment updates" pattern customary in blockchain applications of ZKPs (as can be also readily seen in the accompanying zkWF program construction).

3.4.1 Security assumptions

- The participants can agree on what process model they want to use.

- Each participant has to generate an EdDSA key pair independently and share the public part (only) with the other participants.
- The participants can agree on a shared key, which will be used to encrypt data on the blockchain.

3.4.2 Architecture overview

Architecturally, my framework takes a BPMN process collaboration model, serializes it and generates a zkWF program. An encoded version of the model is included in the program. The full program can prove that specific state changes are valid (or not).

A verifier smart contract can verify these zero-knowledge proofs. A process manager smart contract uses this verifier smart contract to make sure no illegal moves in the process are feasible and stores the sequence of state update commitments.

For the participants, I've also created a GUI interface, which helps to generate zkWF programs, deploy smart contracts, and step the execution process with a visual interface. An overview of the architecture design can be seen on figure 3.7.

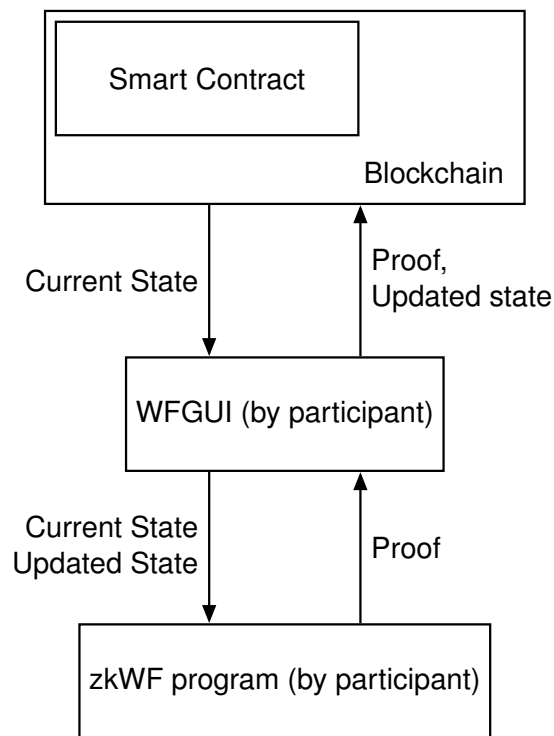


Figure 3.7: Architecture design of the runtime components

3.4.3 Process manager smart contract design

This smart contract contains the logic that is executed on the blockchain. It is responsible for the integrity of the business process execution. It must not contain the current state in plain text. It is designed to be relatively simple since most of the computation is off-chain. Figure 3.8 shows the overall design of this smart contract.

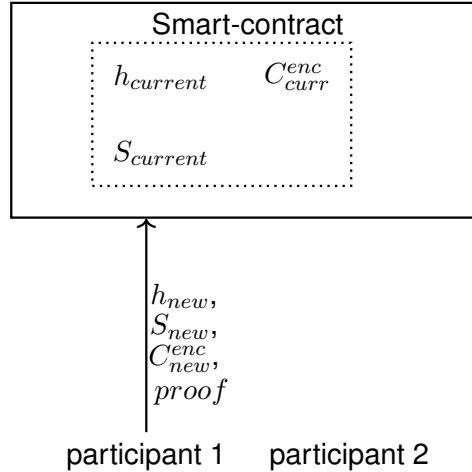


Figure 3.8: Architecture design of the smart-contract

The process manager smart contract must store the following data:

- $h_{current} = hash(s_{current}||r_{current})$ - the hash of the current state and some randomness
- $C_{curr}^{enc} = enc(s_{current})$ - the ciphertext of the current state and the randomness used in the hashing encrypted with a common, predefined key (each participant should have this key)
- $S_{current} = sig(h_{prev}||h_{current})$ - the previous states' and the current states' hashes concatenated, signed by the last acting participant.

The hash of the current state ($h_{current}$) is stored in the smart contract. The purpose of this is to ensure that verifying the state's integrity can be done quickly. It is also necessary for the verifying process. The randomness part in the hash is vital for various reasons.

First, it can be used to "mask" the current state. If an attacker somehow knows the particular business process instance, they could guess it by using the brute-force method. On top of that, the randomness can also be used to generate "fake" state changes, where the state object stays the same, but the randomness changes. Doing that, the hash also alters, making it seem like a new step was made to the outside world. This way, an attacker cannot speculate on the current state based on the number of state changes, even if they have the interrelated BPMN model. If I did not incorporate the randomness part, the hashes would be the same, making this defensive method useless.

It is also important to note that any participant can do these "faked" stage modifications at any point in the business process execution. This randomness should also change when a party makes a "real" step.

For obvious reasons, I cannot store the present form of the execution on the blockchain as plain text. However, I also need to ensure that the execution participants know the current state in (close-to) real-time. This is why an encrypted version of the current state (C_{curr}^{enc}) should be stored in this smart contract. The method of encryption is irrelevant to this paper. The important thing is that the participant can agree on a particular encryption key or a method to generate a key for each state change.

Unfortunately, most ZKP frameworks do not provide any way to verify that a specific message was encrypted to a specified ciphertext with a given key. Implementing this is

possible in theory, but it would be challenging to do it properly. Not only because it has to be secure, but it should also be possible to generate proofs for it reasonably on a standard computer.

This means a malicious participant can update the current state on the blockchain (with valid proof) without sharing it with the other participants. This is because this participant could change the ciphertext stored on the blockchain to some "junk" bytes. The problem also gets more complicated when any participant can upload empty state changes when only the randomness used for hashing is changed.

To make these kinds of attacks more difficult, I made sure that a malicious participant like this is identifiable by the others. This is why I also include a valid participant signature ($S_{current}$) as public input. If I did not have it in place, an attacker with participant rights could anonymously (participants could only guess from the pseudo-anonymous addresses used on the blockchain), making it impossible to stop the execution again at any time of the process.

A participant must be able to update these variables with a function. This function must have the following arguments but may have others:

- $h_{new} = hash(s_{new}||r_{new})$ - the hash of the new state and some randomness
- C_{new}^{enc} - the ciphertext of the new state and the randomness used in the hashing encrypted with a common, predefined key (each participant should have this key)
- S_{new} - a signature from the last acting participant of the process
- *proof* - a valid proof generated by the corresponding zkWF program.

The process manager smart contract must check the validity of the given proof p before accepting the state change. This ensures that only valid state changes can be uploaded to the blockchain.

3.5 Security guarantees

This protocol ensures the following statements:

- The smart contract does not store the logic of the business process.
- Parties not participating cannot read or guess the current state of the business process execution based on the data stored in the smart contract. Process simulation by contract calls can be made infeasible by the above mentioned "junk" steps.
- No party can make an illegal move during the orchestration.
- Messages between the participants can be verified.

3.6 Limitations

This section shows the limitations of this approach and the reason behind them.

3.6.1 Limitations of BPMN models

As I described in the previous section, "special" BPMN models are needed for the program to work.

- The model must be a collaboration: It must have at least one pool.
- It only supports a limited subset of the BPMN specification. See section 3.2.1.
- All tasks should have a public key assigned to them. This should be done at the pool/lane level, not individually.
- A parallel gateway can only have two outgoing (and one incoming) edges OR two incoming (and one outgoing) edges. The reason behind it is described in section 3.3.4.
- A gateway must be followed by an executable event.

3.6.2 Limitations of the proving scheme

Unfortunately, ZoKrates cannot verify that a given ciphertext is the encrypted form of a given message with a given key. The issue that comes with this property is described in section 3.4.3.

3.7 Deployment and operation

This section demonstrates how the necessary components are deployed and operated.

3.7.1 Choosing a distributed ledger

The state manager smart contract can run on two different distributed ledgers:

1. Ethereum (or other EVM-compatible ledgers)
2. Hyperledger Fabric

The verifier part of the smart contract for Ethereum is exported from ZoKrates. For Hyperledger Fabric I used the Zokrates fabric verifier I wrote (see implementation details in section 4.5.2).

Note that currently, only Ethereum is supported in the WFGUI.

3.7.2 Deploying the smart-contract

After generating the zkWF program and choosing a distributed ledger, the state manager smart contract should be deployed. To deploy it for Ethereum, there is a dedicated button in the WFGUI, but you can also deploy it manually (e.g., with the Remix IDE). Deploying the smart contract to Hyperledger Fabric has to be done manually.

When deploying, some variables need to be set in the constructor:

- The hash of the initial state with some randomness

- The ciphertext of the initial state encrypted with the common key

The WFGUI automatically sets these.

Note that the WFGUI generates a new random number at every deployment. This means previously generated proof cannot be used.

3.7.3 Stepping the execution

After deploying the smart contract, the WFGUI should show the user the current state of the process. It also polls the blockchain for updates so the user can see the execution process in (near) real-time.

Before stepping the model, the user has to generate the zero-knowledge proofs. This can be done through the WFGUI or generated manually with the zkWF program.

Behind the scenes, the WFGUI first calculates the hash of the new state. Then, it uses the participants' EdDSa private key to sign the current and the new state hashes concatenation.

After gathering all the public and private inputs for the zkWF program, the WFGUI computes the witness of the zkWF program. Using this witness, the Zokrates toolkit constructs the zero-knowledge proof.

After generating the proof, the user can choose the generated proof in the WFGUI along with a preferred Ethereum address. These Ethereum addresses are handled by web3j and must have enough gas to call the process manager smart contract.

This process is illustrated in figure 3.9 in a UML sequence diagram. The protocols events are demonstrated on a pseudo-code in Algorithm 7.

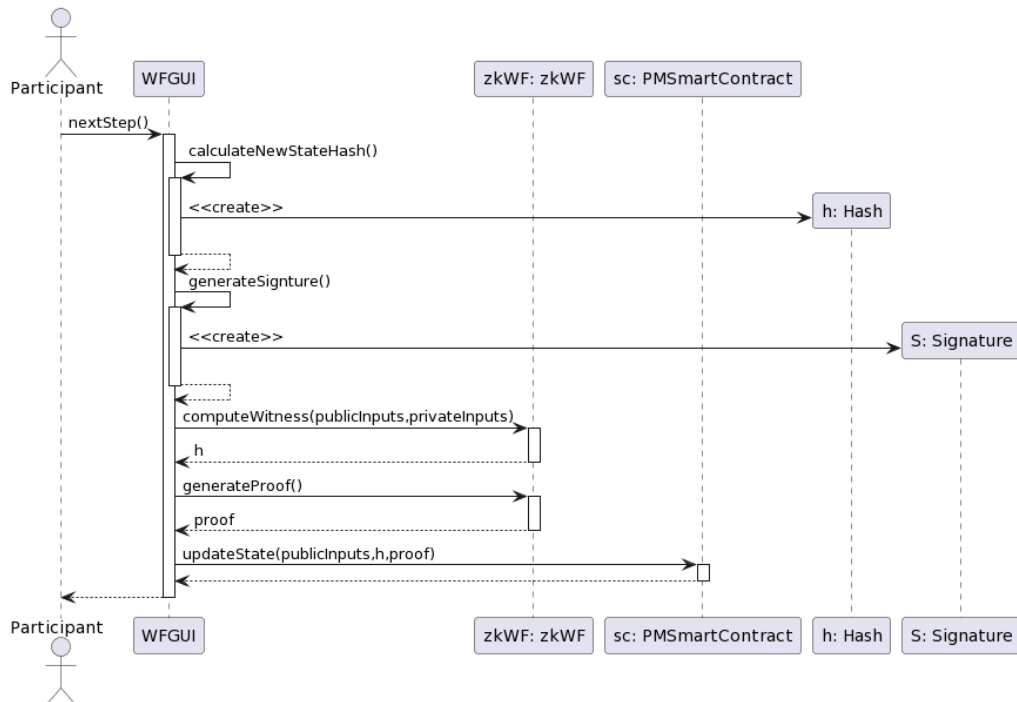


Figure 3.9: Sequence diagram of the state updating process

Algorithm 7 Protocol Events

```
upon init do
  doTrustedSetup()
  deploySmartContract()

upon generateProof(publicInputs,privateInputs) do
  hash_new  $\leftarrow$  calculateNewStateHash(privateInputs)
  generateSignature(hash_current, hash_new)
  witness  $\leftarrow$  computeWitness(publicInputs,privateInputs)
  proof  $\leftarrow$  generateProof(witness)

upon stateUpdate(state, proof) do
  if isProofValid(proof) then
    saveNewState(state)
    Send(NEWSTATE, state)
```

3.8 Testing

To make sure this method of executing business processes is correct, I propose a few ways of challenging it.

3.8.1 Test cases

My first method of testing my approach is by defining test cases and seeing if they can be used in my tool.

3.8.1.1 Simple & corner cases

These test cases were designed to ensure that every supported element works. I also wanted to know how the program reacts to corner cases where the model syntax is correct but the semantics are questionable.

These test cases can be seen below in figure 3.10, 3.11, 3.12, 3.13 and 3.14.

Test 1

This is the most basic test and could be considered the smoke test. It only tests if the simplest form of state transitioning works.

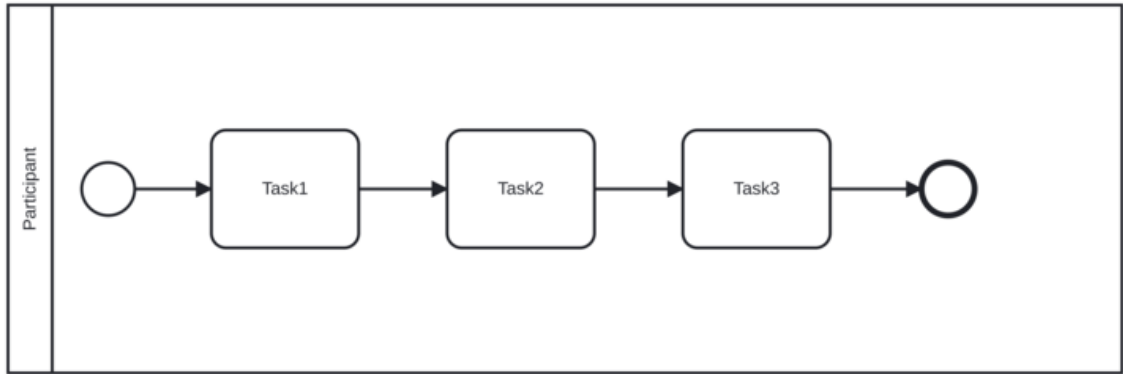


Figure 3.10: Test case 1

Test 2

This test case tests several small but viral functionalities. The most significant of these is exclusive gateways. In this case, the exclusive gateway can choose three different paths: if the global variable "a" is equal to 1, then the state shall transition to task 3 in the next step. If the value is 2, then it will transition to task 4. In any other case, it will select the route to task 2. It also tests the expression serialization.

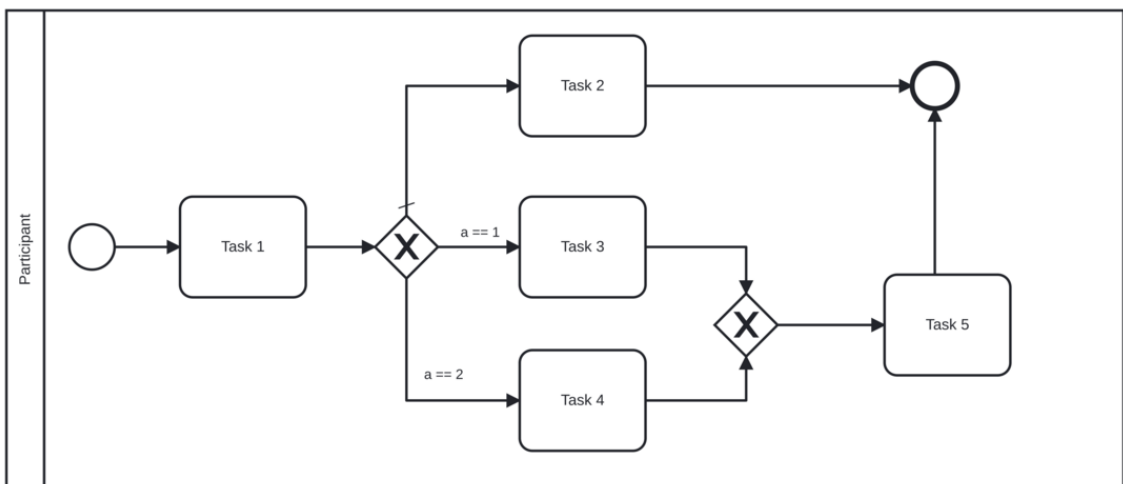


Figure 3.11: Test case 2

Test 3

This simple test case tests the basic functionality of parallel gateways.

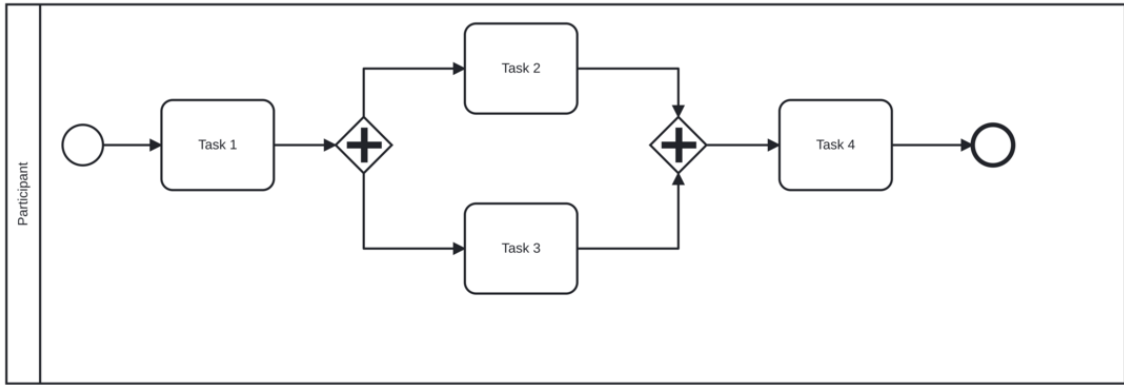


Figure 3.12: Test case 3

Test 4

This test case checks if malformed parallel gateways do not break the program’s functionality.

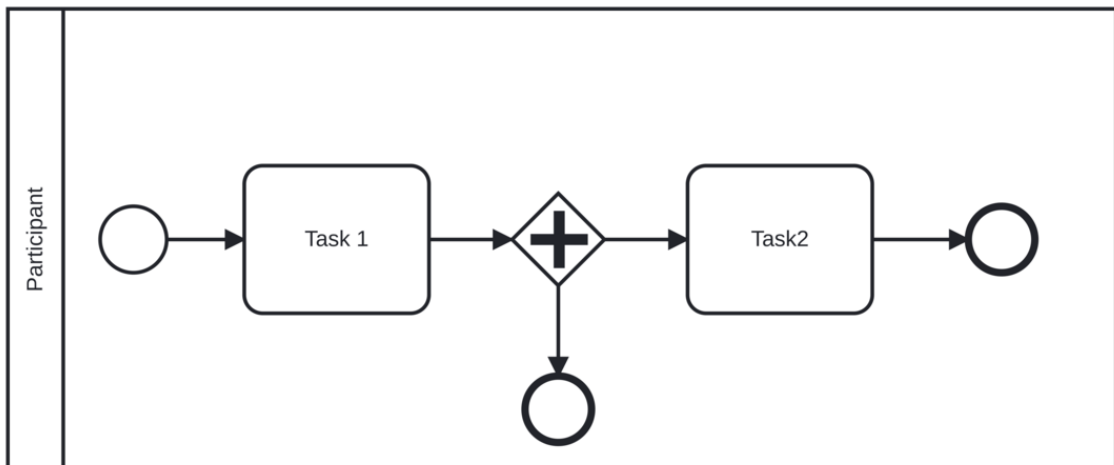


Figure 3.13: Test case 4

Test 5

This test case has two participants. It test if parallel execution of the business process is possible. It also has a two-way messaging flow with intermediate message throw/catch events.

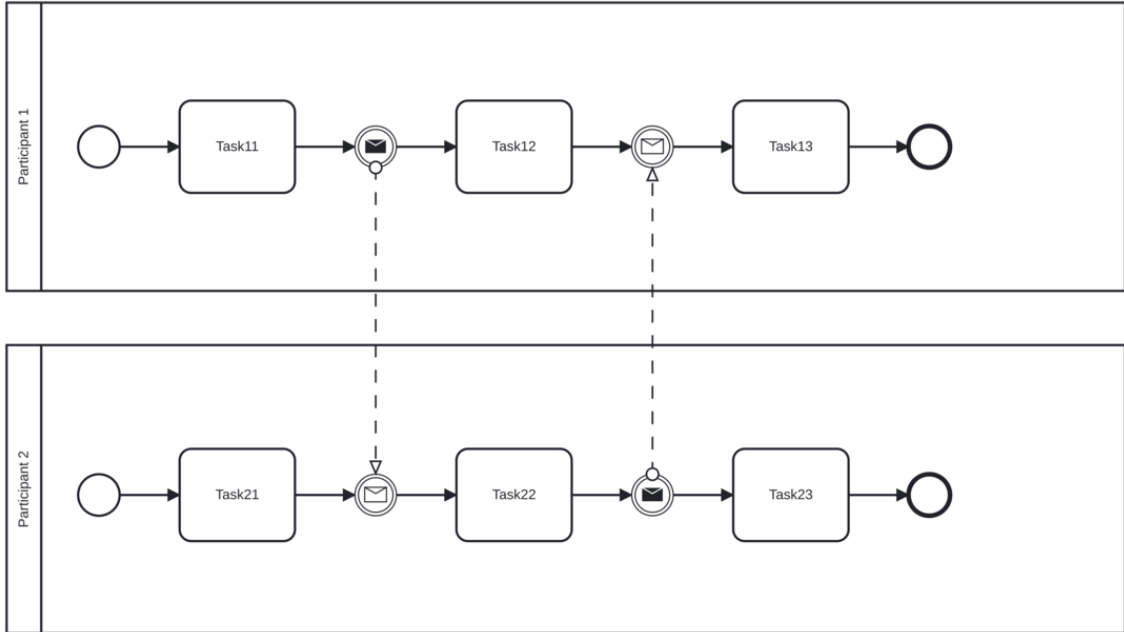


Figure 3.14: Test case 5

3.8.1.2 Representative test

As a Representative test, I wanted to use a complex, real-world example BPMN model. The goal is to see how the program reacts to more oversized state objects (e.g., more executable events, messages, etc.).

In figure 3.15, we can see the model I used for this purpose. It models the process of making a leasing contract.

Note: The model captures a leasing process, where payment related tasks are coloured in blue. These are of no significance at the current stage of my research.

3.8.1.3 Testing framework

To run these tests I designed a testing framework. This framework can run individual test scenarios (i.g. separate steps) and run them as a batch.

The framework also has a GUI and CLI interface. The GUI can be accessed from WFGUI. The CLI interface is designed to work without user interaction. This makes it easy to integrate as a CI/CD pipeline.

3.8.2 Future ideas for testing

An apparent insufficiency of the current approach is that it is not formally proven that the accepted language of the zkWF program conforms to formal specifications of BPMN model and execution semantics.

This will be certainly a worthwhile area of future research. In addition to classic approaches, in this specific context, due to the relative simplicity of the involved components, it seems to be feasible to create a joint executable model of the smart contract, the prover,

the communication protocol and lastly, an executable specification of BPMN semantics. This approach may be feasible even in the form of direct bytecode level model checking of an imperative language as Java (using, e.g., Java PathFinder for analysis). Then, operational semantics conformance can become checkable at least on a model-by-model basis.

Chapter 4

Implementation

4.1 Modeller

The custom attributes we previously defined (See section 3.2.2) cannot be used in a traditional modeller. It is possible to add them to a model manually, but this method can be tedious, and it is easy to make mistakes. To make things more convenient, I implemented a BPMN modeller with built-in support for these attributes.

It is written in javascript, and it makes use of the bpmn-js library's built-in modeller.

An example screenshot can be seen in figure 4.1

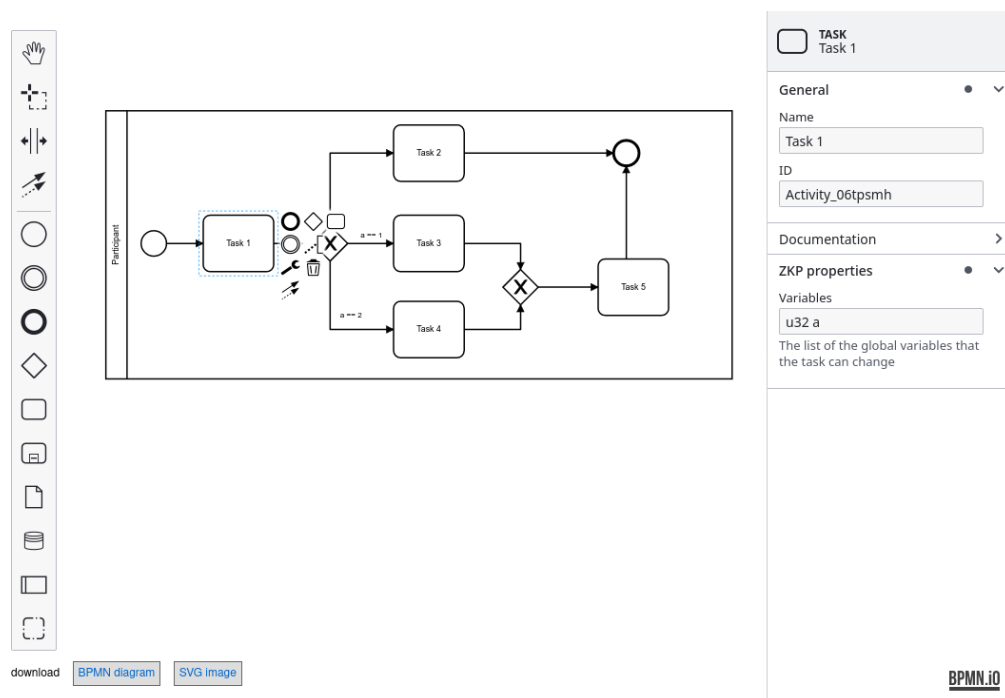


Figure 4.1: Screenshot of the modeller

4.2 zkWF implementation

This section describes how the zkWF program was implemented in ZoKrates.

4.2.1 ZKP framework

To implement zkWF programs, I chose Zokrates (see 2.6.2) because this seemed like the most advanced solution at the time of writing. It also makes it easy to generate a verifier smart contract which would be tedious to write manually. I used ZoKrates version 0.7.13 at the time. It is the latest version as of writing this.

4.2.2 Zokrates implementation

This section describes the structure and the implementation of a zkWF program in ZoKrates.

4.2.2.1 Files

The immediate implementation of the zkWF program is split into two template files: *root.zok.template* and *stateChange.zok.template*. These files cannot be used by themselves. They are meant to be used with the generator program, which adds constants and assertions based on the model.

There is also a *hash.zok.template* file, which is used to generate the hashes of the states. The template files can be found in the 6 appendix.

4.2.2.2 Checking the supplied hash

Generating proof for hashes is implemented in the Zokrates standard library. This makes it really easy to check if the supplied is the one that the current state generates.

4.2.2.3 Proving that a state update is valid

The array P is added to *stateChange.zok.template* as a constant. Since ZoKrates only has unsigned numbers, I implemented a *signed_field* struct along with functions for basic arithmetic operations.

The rest of the implementation follows the method described in section 3.3.6.2.

4.2.2.4 Authorisation

The authorisation process uses EdDSa key pairs. Cryptographic operations for EdDSa, like proving that a party owns a private key with a specific public key or proving that a signature is valid, are also implemented in the Zokrates standard library.

To generate these keys, I created a simple python script that loads these keys from a fixed set. These keys were used for testing and debugging.

4.2.2.5 Verification of an encrypted ciphertext

Unfortunately, ZoKrates does not provide any way to verify that a specific message was encrypted to a specific ciphertext. Implementing this is possible in theory, but it would be challenging to do it properly. Not only because it has to be secure, but it should also be possible to generate proofs for it reasonably on an average home computer.

4.3 Code generator

I implemented the code generator part of the tool in Kotlin. To load the BPMN models, I wrote a BPMN interpreter. This interpreter loads the whole model into the classes I wrote. These classes are shown in a UML class diagram in figure 4.2.

Each BPMN element (See section 3.2.1) has its class. They are derived from the abstract class Event.

Elements in the model that are not events (like Messages, Message flows) also have their classes. The generalisation of these elements and events is the Element abstract class.

The Model class is initialised with a BPMN file. It is responsible for parsing the model to the correct classes.

After serialising the model, the `generateZokratesCode()` function generates the zkWK program from the ZoKrates template files. First, the model is encoded in the way described in section 3.3.4. Then it generates the code for calculating the state hashes (See section 3.3.6.1), checking the variable write permissions, ensuring exclusive gateway paths, and verifying message sending (See section 3.3.6.2). Lastly, it saves the freshly generated code to files.

4.4 Zokrates wrapper

ZoKrates is written in rust. It can be used through a CLI interface or with a javascript library. Since most of the code I wrote is in Kotlin, I needed to find a way to interact with ZoKrates.

My approach to this problem is simple: Download the Zokrates CLI tool and execute the commands with a `ProcessBuilder` class. With this method, I can compile programs, do the trusted setup, compute witnesses, generate proofs, export verifier smart contracts, and verify proofs.

This makes it easy to integrate ZoKrates into a GUI application.

4.5 Smart contract implementation

To make my approach less dependent on one technology, I implemented the process manager smart contract for two distributed ledger systems: Ethereum and Hyperledger Fabric.

4.5.1 EVM

The process manager smart contract (see section 3.4.3) is implemented in Solidity version 0.8.0. It is derived from a Verifier smart contract generated by ZoKrates.

Hash The hash of the current state is stored in the smart contract as a struct. This struct contains eight uint-type variables, each holding 32 bytes of the hash. This struct is also used for verifying zero-knowledge proofs.

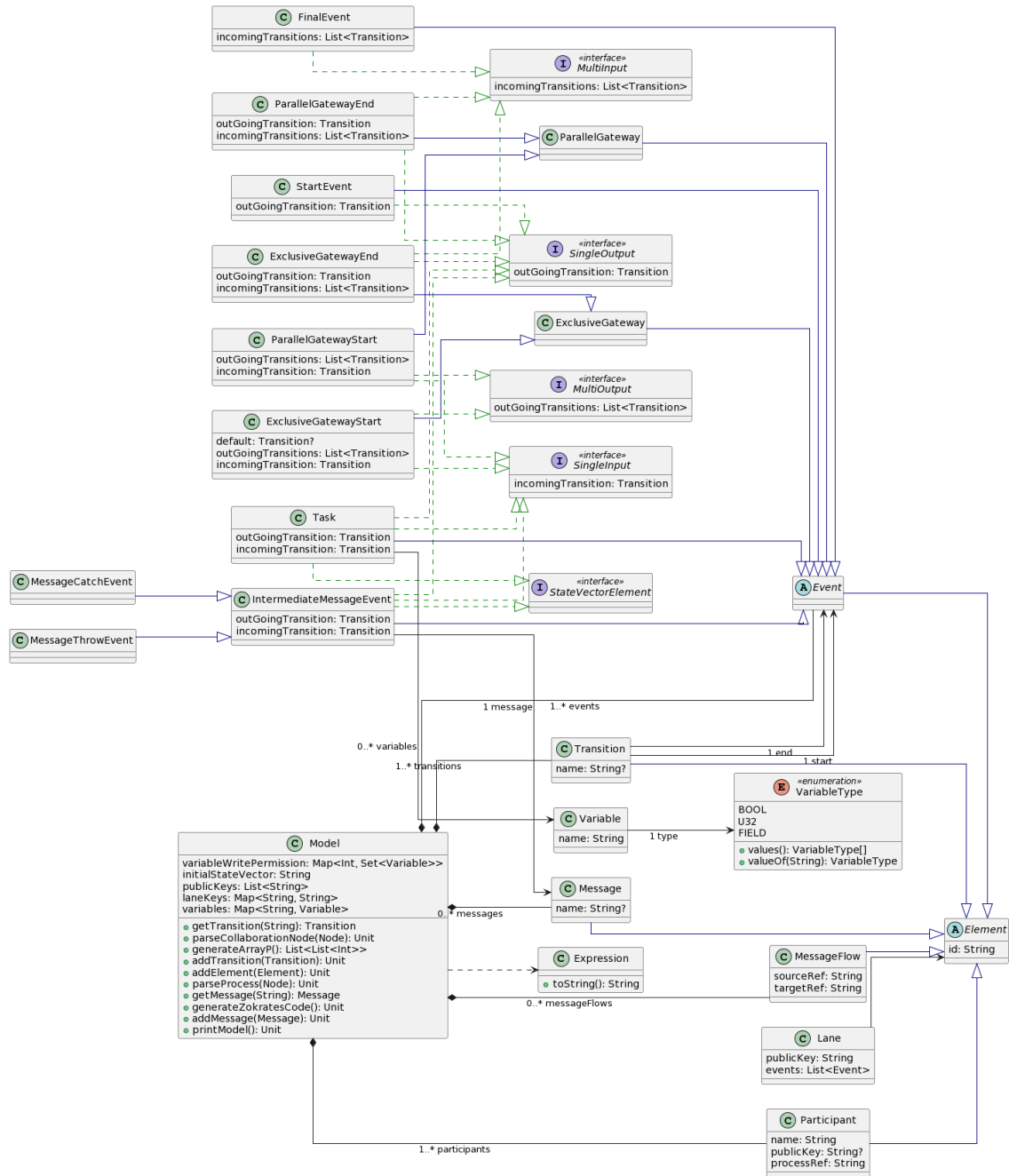


Figure 4.2: Class diagram of the BPMN elements

Signature The signature of the previous and current state hash sign by the last acting participant is stored as a struct in the smart contract. It has two fields: an array R with two `uint256` members and `uint256 S`. It represents an EdDSa signature.

Ciphertext The encrypted version of the current state is stored in the smart contract as a string.

4.5.2 Hyperledger Fabric

ZoKrates can only generate smart contract verifiers in Solidity. Fortunately, Hyperledger Fabric chaincodes run in standard docker containers and are written in traditional programming languages like Java. This means I can use the same approach as in section 4.4.

The rest of the smart contract implementation is analogue to the Solidity one.

4.6 WFGUI

I made a GUI application called WFGUI (WorkFlow GUI) to fully integrate my approach into a tool. I implemented it in Kotlin (as with the rest of the parts) with the TornadoFX library (Kotlin wrapper for JavaFX).

The GUI itself is separated into three different tabs. One for modelling, one for testing, and the last for deploying and operating.

4.6.1 Modeler tab

The modeling tab integrates the modeller described in section 4.1. It uses the web view feature of JavaFX. Figure 4.3 shows the modeller in the GUI application.

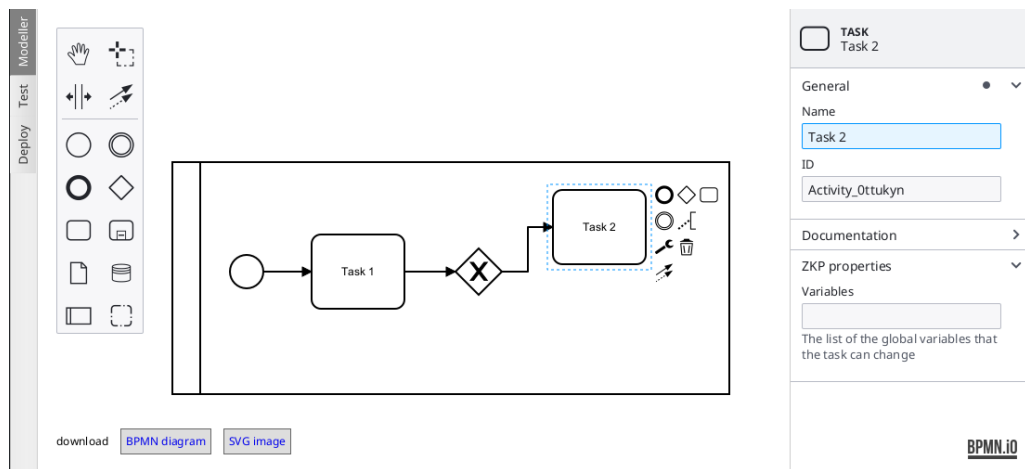


Figure 4.3: Modeler in WFGUI

4.6.2 Testing tab

The testing tab serves two purposes. First, it can verify that a model can be used with my program. It can generate and compile the Zokrates code from a model with a click of a button. It can also do the trusted setup, and with a form, it can generate proofs (after the setup). On this page, the user can also load a test case set, which can test different scenarios in a batch. A preview of this page is shown in figure 4.4.

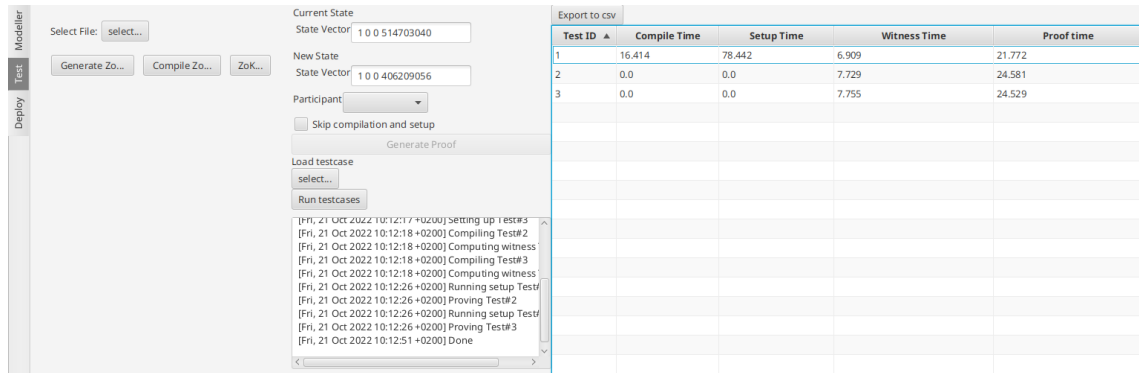


Figure 4.4: Testing tab in WFGUI

4.6.3 Deployment and operation tab

This is the most critical tab out of all. It is responsible for generating the Zokrates code, doing the trusted setup, creating, deploying the smart contract, and updating the state on the blockchain.

The user first needs to open a BPMN model from the Files menu, while the tool automatically generates the ZoKrates code in the background. The user can then decide to deploy a new smart contract (after the trusted setup), or they can input an address of a previously deployed smart contract. Either way, the program will monitor the current state of the process on the blockchain and visualise it for the user.

The user can generate new zero-knowledge proofs in the sidebar, or they can use pre-generated proofs from a drop-down menu. Before stepping the process execution, the user can also choose which Ethereum address they want to use for calling the smart contract. This is useful because participants can use different pseudo-anonymous identities for each state change. Figure 4.5 shows an example screenshot of this page.

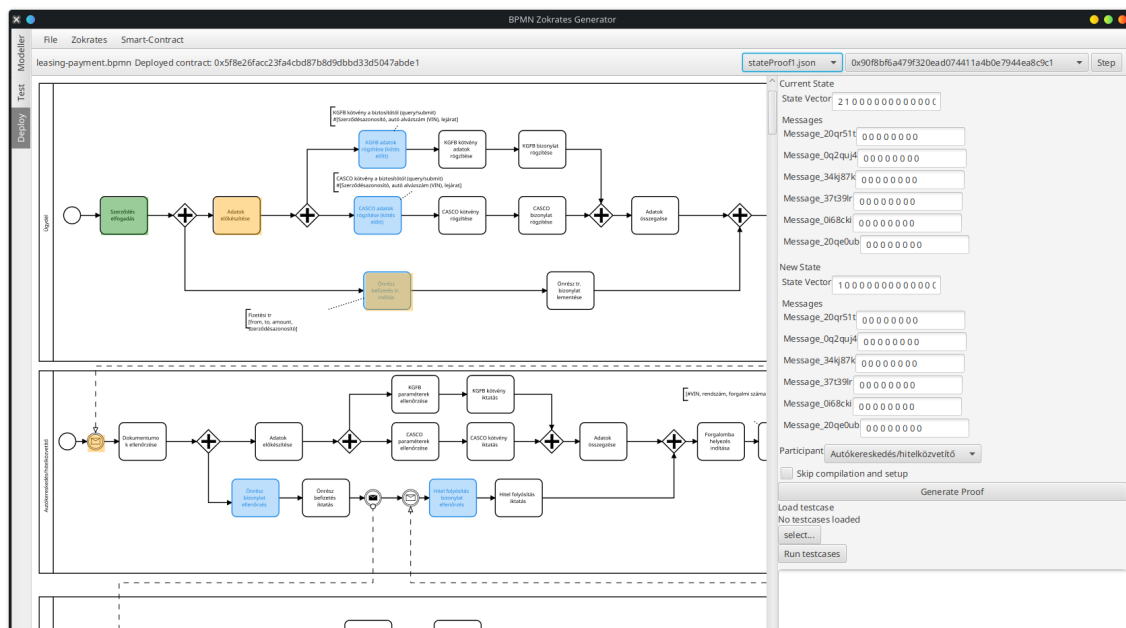


Figure 4.5: Deployment and Operation tab in WFGUI

4.6.4 Video presentation

A video demo presentation is available by clicking figure 4.6 or at this link¹.

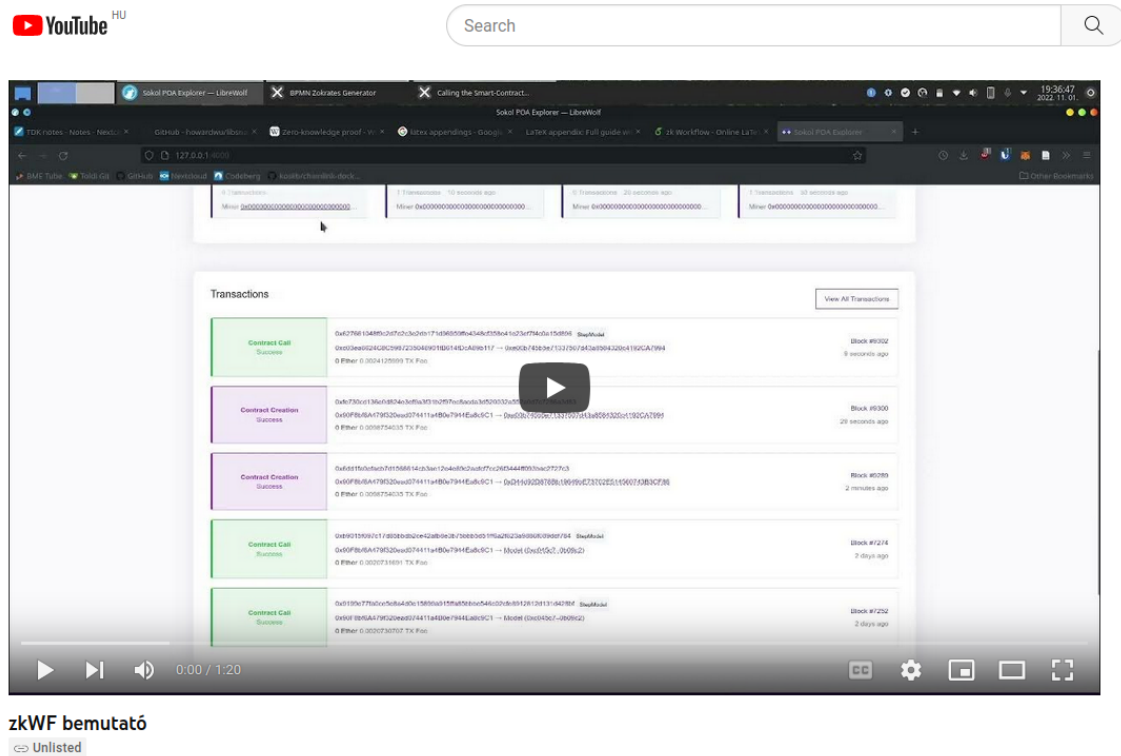


Figure 4.6: Demo presentation video

¹<https://youtu.be/MH3GrW4Jtwg>

Chapter 5

Results

In this chapter, I provide quantitative performance figures to demonstrate the feasibility of the zkWF approach.

5.1 Simple & corner cases

This section reports on the performance figures for the earlier defined simple test cases.

5.1.1 Hardware used for testing

To run the test cases, I used my home desktop PC with the following specs:

- CPU: AMD Ryzen 7 2700 Processor
- RAM: 16 GB DDR4 memory

5.1.2 Software used for testing

To run each test scenario, I used my testing framework (section 3.8.1.3). To measure the gas required to run on an Ethereum-compatible blockchain, I set up a private test network using geth version 1.10.25.

5.1.3 Comparing test cases

Table 5.1 compares the test cases based on their sizes. These aspects ruffly measure the complexity of each model. In the following sections, I want to find how the complexity of a model is linked to compilation time, setup time(the time it takes to do the trusted setup phase), proof creation time (the amount of time that is needed to produce a proof), and gas usage on Ethereum.

Test case	Vertices	Edges	Executable	Size of P	Test 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

Table 5.1: Compression of the test cases based on their sizes

5.1.4 Results

I ran all tests sequentially, and all test cases were successful. Table 5.2 show the timing costs of the off-chain computations. The compilation and the setup phase are only executed once. The table also shows the average proof creation time, the sum of computing the witness and generating the proof. This shows that deploying a smart contract and stepping the execution can be done in just a few minutes.

Test case	Compilation time	Setup time	Average Proof creation
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

Table 5.2: Off-chain computation timing results

Table 5.3 shows how the smart contract performs if deployed on an Ethereum-compatible blockchain. The table demonstrates how much gas is used to deploy the smart contract (one-time fee). It also describes how much gas is required to update the current state on the blockchain.

Test case	Deployment costs	Average gas cost
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

Table 5.3: Gas usage on Ethereum

5.2 Complex test

This section presents performance figures for the earlier introduced representative leasing collaboration model.

5.2.1 Size of the model

Table 5.4 shows the size of my representative model. If we compare it to the previous models (see table 5.1), it is about 5-6 times larger in size.

Test case	Vertices	Edges	Executable	Size of P	Test Scenarios
Representative	68	69	50	54	52

Table 5.4: size of the representative model

5.2.2 Results

The ZoKrates code was compiled in 81.02 seconds, and the setup time was done in 187.33 seconds.

Table 5.5 demonstrates the sizes of the files resulting after the compilation and setup phase.

File	Size
Flattened code (out)	3.0 GB
Proving key (proving.key)	95 MB
Verification key (verification.key)	4.4 kB

Table 5.5: Resulted file sizes

5.2.2.1 Model execution

All of the tasks successfully prove that the whole model can be executed with this approach.

5.2.3 Proving time

The distribution of the witness computation times can be seen in figure 5.1. Figure 5.2 shows the distribution of proof generation times. Both of these are reasonably close to a normal distribution. On average, the witness computation took 32.04 seconds, while the proof generation took 90.43 seconds on average.

These calculation times are not drastically higher than with the smaller models. This model is 5-6 times larger, but proofs only took 2.29 times more time. This proves that this tool is still practical even with larger models.

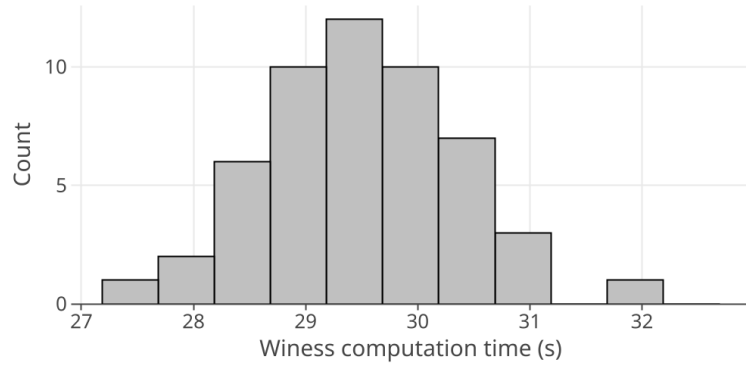


Figure 5.1: Distribution of the witness computation time for the complex test

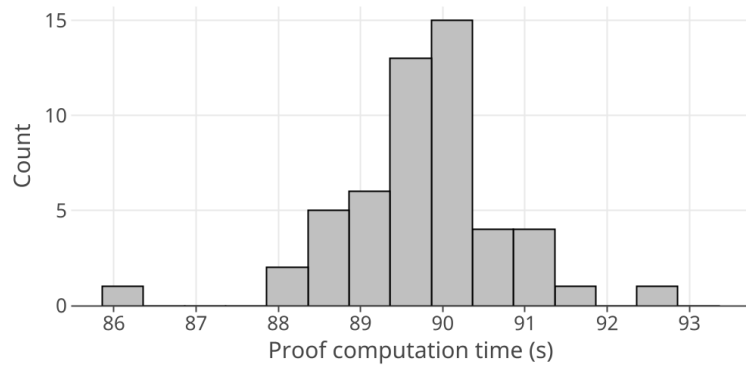


Figure 5.2: Distribution of the proof computation time for the complex test

5.2.4 Gas usage

The cost of deployment was costed 2,408,635 gas. I also measured the gas usage in each test scenario on an Ethereum test network. The average step costs 548,898 gas.

It is essential to point out that even though the model is 5-6 times larger than the smaller models shown in table 5.1, the gas usage is only 11.90 per cent higher than the lowest in previous steps.

This was expected, since the hashes, the signatures, and the proofs have a fixed length. This means the only thing that drives gas usage higher in larger models is the encrypted version of the current state.

5.3 Comparison between existing solutions

It is hard to compare this method to other approaches since this is the first one that uses zero-knowledge proofs to hide the current state of the execution process. In theory, I could compare it to Aivo et al. [2], but it uses a vastly different approach and tools, resulting in an unfair comparison.

However, the Ethereum smart contracts' gas usage can be compared to existing techniques. The deployment cost is on par with or less than the existing solutions.

On the other hand, the cost of updating the state is significantly higher compared to previous approaches like Chorchain [10] or Caterpillar [22]. Chorchain [10] uses about

92,905 gas on average for each message, while Caterpillar [22] requires similar amounts of gas on average.

Unfortunately, this is due to the cost of verifying proof on-chain. Overcoming this issue would be rather tricky since the verification smart contract ZoKrates generates uses the pre-compiled smart contract built into Ethereum. In particular, a zkSNARK proof is verified on-chain with about 500,000 gas units on average¹. In theory, this scales with the number of public inputs used for generating proofs, so reducing them would also decrease the gas expense.

¹See <https://ethereum.org/en/zero-knowledge-proofs/#proof-verification-costs>

Chapter 6

Conclusion

I designed a protocol that allows the users to orchestrate business processes on multiple blockchain systems without storing the current state and the logic of the execution on-chain.

The protocol's main component is a program written in ZoKrates. This program generates zero-knowledge proofs for making steps in the business process.

This protocol also contains a process manager smart contract that stores data on the blockchain that is only meaningful for the participants. To update these, the participant has to upload a valid zero-knowledge proof generated by my program.

I fully integrated this protocol into a tool. This tool includes a modeller, a participant-side SDK, a wallet manager, and a simple visual interface.

I tested my approach with several test models and scenarios. These tests seem to show that this approach is practical but has limitations. I see gas usage as the most significant flaw this approach has. In the future, I want to work on reducing the limitations of this approach. I want to remove some of the restrictions mentioned in section 3.6.1. This will make creating models for this tool more accessible. Some mechanics such as loops are also planned.

I also want to prove the practical feasibility of this business process orchestration approach and rigorously prove its correctness. There are different ways to do this; some of these were mentioned in section 3.8.2.

I also plan to fully integrate Hyperledger Fabric into my current tool to make deployment and operation more accessible on other distributed ledger systems.

I would also love to reduce the gas usage of this approach on Ethereum to make transactions cheaper. This would be possible by generating proofs for making several steps in a batch.

Bibliography

- [1] ABID, Amal ; CHEIKHROUHOUS, Saoussen ; JMAIEL, Mohamed: Modelling and Executing Time-Aware Processes in Trustless Blockchain Environment. In: KALLEL, Slim (Hrsg.) ; CUPPENS, Frédéric (Hrsg.) ; CUPPENS-BOULAHIA, Nora (Hrsg.) ; HADJ KACEM, Ahmed (Hrsg.): *Risks and Security of Internet and Systems*. Cham : Springer International Publishing, 2020 (Lecture Notes in Computer Science), S. 325–341. – ISBN 978-3-030-41568-6
- [2] AIVO, Toots ; PEETER, Laud: Zero-Knowledge Proofs for Business Processes.
- [3] BABKIN, Eduard ; KOMLEVA, Nataliya: Model-Driven Liaison of Organization Modeling Approaches and Blockchain Platforms. In: AVEIRO, David (Hrsg.) ; GUIZZARDI, Giancarlo (Hrsg.) ; BORBINHA, José (Hrsg.): *Advances in Enterprise Engineering XIII*. Cham : Springer International Publishing, 2020 (Lecture Notes in Business Information Processing), S. 167–186. – ISBN 978-3-030-37933-9
- [4] BEN-SASSON, Eli ; BENTOV, Iddo ; HORESH, Yinon ; RIABZEV, Michael: Scalable, transparent, and post-quantum secure computational integrity.
- [5] BEN-SASSON, Eli ; BENTOV, Iddo ; HORESH, Yinon ; RIABZEV, Michael: Scalable, transparent, and post-quantum secure computational integrity.
- [6] BOUBETA-PUIG, Juan ; ROSA-BILBAO, Jesús ; MENDLING, Jan: CEPchain: A graphical model-driven solution for integrating complex event processing and blockchain. In: *Expert Systems with Applications* 184 (2021), Dec, S. 115578. – ISSN 0957-4174
- [7] CHINOSI, Michele ; TROMBETTA, Alberto: BPMN: An introduction to the standard. In: *Computer Standards Interfaces* 34 (2012), Nr. 1, S. 124–134. – URL <https://www.sciencedirect.com/science/article/pii/S0920548911000766>. – ISSN 0920-5489
- [8] CORRADINI, Flavio ; MARCELLETTI, Alessandro ; MORICHETTA, Andrea ; POLINI, Andrea ; RE, Barbara ; SCALA, Emanuele ; TIEZZI, Francesco: Model-driven engineering for multi-party business processes on multiple blockchains. In: *Blockchain: Research and Applications* 2 (2021), Sep, Nr. 3, S. 100018. – ISSN 2096-7209
- [9] CORRADINI, Flavio ; MARCELLETTI, Alessandro ; MORICHETTA, Andrea ; POLINI, Andrea ; RE, Barbara ; TIEZZI, Francesco: 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)*, May 2022, S. 25–32
- [10] CORRADINI, Flavio ; MARCELLETTI, Alessandro ; MORICHETTA, Andrea ; RE, Barbara ; TIEZZI, Francesco: ChorChain: A Model-Driven Framework for Choreography-Based Systems Using Blockchain.

- [11] EBERHARDT, Jacob ; TAI, Stefan: 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 (Smart-Data)*, Jul 2018, S. 1084–1091
- [12] GARAMVÖLGYI, Péter ; KOCSIS, Imre ; GEHL, Benjámín ; KLENIK, Attila: Towards Model-Driven Engineering of Smart Contracts for Cyber-Physical Systems. In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, Jun 2018, S. 134–139. – ISSN 2325-6664
- [13] GOLDREICH, Oded ; MICALI, Silvio ; WIGDERSON, Avi: Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. In: *Journal of the ACM* 38 (1991), Jul, Nr. 3, S. 690–728. – ISSN 0004-5411, 1557-735X
- [14] GROTH, Jens: On the Size of Pairing-based Non-interactive Arguments. (2016). – URL <https://eprint.iacr.org/2016/260>. – Report Number: 260
- [15] GROTH, Jens ; MALLER, Mary: Snarky Signatures: Minimal Signatures of Knowledge from Simulation-Extractable SNARKs. (2017). – URL <https://eprint.iacr.org/2017/540>. – Report Number: 540
- [16] GROUP, Object M.: Business Process Model and Notation (BPMN), Version 2.0.
- [17] HAMDAQA, Mohammad ; METZ, Lucas Alberto P. ; QASSE, Ilham: iContractML: A Domain-Specific Language for Modeling and Deploying Smart Contracts onto Multiple Blockchain Platforms. In: *Proceedings of the 12th System Analysis and Modelling Conference*. New York, NY, USA : Association for Computing Machinery, Oct 2020 (SAM '20), S. 34–43. – URL <https://doi.org/10.1145/3419804.3421454>. – ISBN 978-1-4503-8140-6
- [18] HOPWOOD, Daira ; BOWE, Sean ; HORNBY, Taylor ; WILCOX, Nathan: Zcash Protocol Specification, Version 2022.3.7 [NU5].
- [19] HORNÁČKOVÁ, Barbora ; SKOTNICA, Marek ; PERGL, Robert: Exploring a Role of Blockchain Smart Contracts in Enterprise Engineering. In: AVEIRO, David (Hrsg.) ; GUIZZARDI, Giancarlo (Hrsg.) ; GUERREIRO, Sérgio (Hrsg.) ; GUÉDRIA, Wided (Hrsg.): *Advances in Enterprise Engineering XII*. Cham : Springer International Publishing, 2019 (Lecture Notes in Business Information Processing), S. 113–127. – ISBN 978-3-030-06097-8
- [20] LADLEIF, Jan ; WESKE, Mathias ; WEBER, Ingo: Modeling and Enforcing Blockchain-Based Choreographies. In: HILDEBRANDT, Thomas (Hrsg.) ; DONGEN, Boudewijn F. van (Hrsg.) ; RÖGLINGER, Maximilian (Hrsg.) ; MENDLING, Jan (Hrsg.): *Business Process Management*. Cham : Springer International Publishing, 2019 (Lecture Notes in Computer Science), S. 69–85. – ISBN 978-3-030-26619-6
- [21] LU, Qinghua ; BINH TRAN, An ; WEBER, Ingo ; O'CONNOR, Hugo ; RIMBA, Paul ; XU, Xiwei ; STAPLES, Mark ; ZHU, Liming ; JEFFERY, Ross: Integrated model-driven engineering of blockchain applications for business processes and asset management. In: *Software: Practice and Experience* 51 (2021), Nr. 5, S. 1059–1079. – ISSN 1097-024X
- [22] LÓPEZ-PINTADO, Orlenys ; GARCÍA-BAÑUELOS, Luciano ; DUMAS, Marlon ; WEBER, Ingo ; PONOMAREV, Alexander: Caterpillar: A business process execution engine on

- the Ethereum blockchain. In: *Software: Practice and Experience* 49 (2019), Nr. 7, S. 1162–1193. – ISSN 1097-024X
- [23] MARCHESI, Michele ; MARCHESI, Lodovica ; TONELLI, Roberto: An Agile Software Engineering Method to Design Blockchain Applications. In: *Proceedings of the 14th Central and Eastern European Software Engineering Conference Russia on ZZZ - CEE-SECR '18*. Moscow, Russian Federation : ACM Press, 2018, S. 1–8. – URL <http://dl.acm.org/citation.cfm?doid=3290621.3290627>. – ISBN 978-1-4503-6176-7
- [24] MAVRIDOU, Anastasia ; LASZKA, Aron: Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach. In: MEIKLEJOHN, Sarah (Hrsg.) ; SAKO, Kazue (Hrsg.): *Financial Cryptography and Data Security*. Berlin, Heidelberg : Springer, 2018 (Lecture Notes in Computer Science), S. 523–540. – ISBN 978-3-662-58387-6
- [25] MENDLING, Jan ; WEBER, Ingo ; AALST, Wil Van D. ; BROCKE, Jan V. ; CABANILLAS, Cristina ; DANIEL, Florian ; DEBOIS, Søren ; CICCIO, Claudio D. ; DUMAS, Marlon ; DUSTDAR, Schahram ; GAL, Avigdor ; GARCÍA-BAÑUELOS, Luciano ; GOVERNATORI, Guido ; HULL, Richard ; ROSA, Marcello L. ; LEOPOLD, Henrik ; LEYMANN, Frank ; RECKER, Jan ; REICHERT, Manfred ; REIJERS, Hajo A. ; RINDERLE-MA, Stefanie ; SOLTI, Andreas ; ROSEMANN, Michael ; SCHULTE, Stefan ; SINGH, Munindar P. ; SLAATS, Tijs ; STAPLES, Mark ; WEBER, Barbara ; WEIDLICH, Matthias ; WESKE, Mathias ; XU, Xiwei ; ZHU, Liming: Blockchains for Business Process Management - Challenges and Opportunities. In: *ACM Trans. Manage. Inf. Syst.* 9 (2018), feb, Nr. 1. – URL <https://doi.org/10.1145/3183367>. – ISSN 2158-656X
- [26] MERCENNE, Lucie ; BROUSMICHE, Kei-Leo ; HAMIDA, Elyes B.: Blockchain Studio: A Role-Based Business Workflows Management System. In: *2018 IEEE 9th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, Nov 2018, S. 1215–1220
- [27] POURMIRZA, Shaya ; PETERS, Sander ; DIJKMAN, Remco ; GREFEN, Paul: A systematic literature review on the architecture of business process management systems. In: *Information Systems* 66 (2017), S. 43–58. – URL <https://www.sciencedirect.com/science/article/pii/S0306437917300248>. – ISSN 0306-4379
- [28] ROCHA, Henrique ; DUCASSE, Stéphane: Preliminary Steps Towards Modeling Blockchain Oriented Software. In: *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, May 2018, S. 52–57
- [29] SILVA, Diogo ; GUERREIRO, Sérgio ; SOUSA, Pedro: Decentralized Enforcement of Business Process Control Using Blockchain. In: AVEIRO, David (Hrsg.) ; GUIZZARDI, Giancarlo (Hrsg.) ; GUERREIRO, Sérgio (Hrsg.) ; GUÉDRIA, Wided (Hrsg.): *Advances in Enterprise Engineering XII*. Cham : Springer International Publishing, 2019 (Lecture Notes in Business Information Processing), S. 69–87. – ISBN 978-3-030-06097-8
- [30] VAN DER AALST, Wil M. ; LA ROSA, Marcello ; SANTORO, Flávia M.: Business process management. In: *Business & Information Systems Engineering* 58 (2016), Nr. 1, S. 1–6
- [31] WEBER, Ingo ; XU, Xiwei ; RIVERET, Régis ; GOVERNATORI, Guido ; PONOMAREV, Alexander ; MENDLING, Jan: Untrusted Business Process Monitoring and Execution

Using Blockchain. In: LA ROSA, Marcello (Hrsg.) ; LOOS, Peter (Hrsg.) ; PASTOR, Oscar (Hrsg.): *Business Process Management*. Cham : Springer International Publishing, 2016 (Lecture Notes in Computer Science), S. 329–347. – ISBN 978-3-319-45348-4

[32] ZKPROOF: ZKProof Community Reference.

ZoKrates template files

A.1 State vector checking

```
import "hashes/sha256/512bitPacked" as sha256packed
import "hashes/sha256/512bit" as sha256h
from "ecc/babyjubjubParams" import BabyJubJubParams
import "signatures/verifyEddsa.zok" as verifyEddsa
import "ecc/babyjubjubParams.zok" as context
import "utils/casts/u32_to_field.zok" as u32_to_field
import "utils/casts/field_to_u32.zok" as field_to_u32

struct signed_field {
  field value
  bool positive
}

def signed_field_add(signed_field a,signed_field b) -> signed_field:
field value = if a.positive == b.positive then a.value + b.value else if a.value > b.value then a.
  value - b.value else b.value - a.value fi fi
bool positive = if a.positive == b.positive then a.positive else if a.value > b.value && a.positive
  || b.value > a.value && b.positive then true else false fi fi
return signed_field{value: value, positive: positive}

def signed_field_sub(signed_field a,signed_field b) -> signed_field:
signed_field temp = signed_field{value: b.value,positive: !b.positive}
return signed_field_add(a,temp)

def signed_field_graterThan(signed_field a,signed_field b) -> bool:
signed_field temp = signed_field_sub(a,b)
return temp.positive

def signed_field_lessThan(signed_field a,signed_field b) -> bool:
return signed_field_graterThan(b,a)

def signed_field_graterThanZero(signed_field a) -> bool:
return a.value > 0 && a.positive

def signed_field_lessThanZero(signed_field a) -> bool:
return a.value > 0 && !a.positive

def signed_field_equal(signed_field a,signed_field b) -> bool:
return a.value == b.value && b.positive == a.positive

def signed_field_create(field v,bool p) -> signed_field:
return signed_field{value: v,positive: p}

def isNothing(signed_field[2] a) -> bool:
return a[0].value == 0 && a[1].value == 1 && !a[1].positive

def Li_graterThan(signed_field[3] a,signed_field[3] b) -> bool :
return if signed_field_lessThan(a[1],b[1]) && !isNothing(a[0..2]) && !isNothing(b[0..2])||
  signed_field_equal(a[1],b[1]) && signed_field_lessThan(a[2],b[2]) && !isNothing(a[0..2]) && !
  isNothing(b[0..2]) || isNothing(a[0..2]) && isNothing(b[0..2]) && signed_field_lessThan(a[2],b
  [2]) || !isNothing(a[0..2]) && isNothing(b[0..2]) then true else false fi

const signed_field signed_field_zero = signed_field{value: 0, positive:true}
const signed_field signed_field_one = signed_field{value: 1, positive:true}
```

```

const signed_field signed_field_negative_one = signed_field{value: 1, positive:false}
const signed_field[2] nothing = [signed_field{value: 0, positive:true},signed_field{value: 1,
positive:false}]

[[ !!!REPLACE THIS WITH CONSTANTS!!! ]]

def main(private u32[len_V] s_curr,private u32[len_V] s_next) -> u32[4]:
  signed_field[3][2] changes = [nothing;3]
  u32[4] chres = [0;4]
  u32 change_count = 0
  u32 pos = 0
  for u32 i in 0..len_V do
    assert( s_curr[i] <= 2 && s_next[i] <= 2)
  endfor

  for u32 i in 0..len_V do
    u32 change_id = if change_count <= 2 then change_count else 0 fi
    changes[change_id] = if s_curr[i] == s_next[i] then\
    changes[change_id] \
    else if s_curr[i] == 1 && s_next[i] == 2 then\
    [signed_field{value: 1,positive: false},signed_field{value:u32_to_field(i),positive:true}]\
    else if s_curr[i] == 0 && s_next[i] == 1 then \
    [signed_field{value: 1,positive: true},signed_field{value: u32_to_field(i),positive:true}] \
    else if s_curr[i] == 0 && s_next[i] == 2 then \
    [signed_field{value: 1,positive: false},signed_field{value: u32_to_field(i),positive:true}] \
    else \
    [signed_field{value: 1,positive: false},signed_field{value: 1,positive:false}] \
    fi fi fi fi
    pos = if s_curr[i] != s_next[i] && !changes[change_id][0].positive then i else pos fi
    chres[change_id] = if s_curr[i] != s_next[i] then i else chres[change_id] fi
    change_count = if s_curr[i] == s_next[i] then change_count else change_count + 1 fi
  endfor
  chres[3] = pos

  assert(change_count != 1 || pos != 0) // New tokens cannot be created by calling the start event
  again...
  assert(change_count <= 3)
  bool result = if change_count == 0 then true else false fi
  for u32 i in 0..len_w do
    bool good = true
    for u32 j in 0..3 do
      bool pair_found = false
      for u32 k in 0..3 do
        pair_found = if signed_field_equal(p[i][j][0],changes[k][0]) && signed_field_equal(p[i][j]
)[1],changes[k][1]) then true else pair_found fi
      endfor
      good = if pair_found then good else false fi
    endfor
    result = if good then true else result fi
  endfor

  for u32 i in 0..len_w do
    bool good = if change_count == 1 then true else false fi
    u32 minusCount = 0
    bool contains = false
    u32 other = 0
    for u32 j in 0..3 do
      minusCount = if signed_field_equal(p[i][j][0],signed_field_create(1,false)) then minusCount + 1
      else minusCount fi
      contains = if signed_field_equal(p[i][j][0],changes[0][0]) && signed_field_equal(p[i][j][1],
changes[0][1]) then true else contains fi
      other = if signed_field_equal(p[i][j][0],changes[0][0]) && !signed_field_equal(p[i][j][1],
changes[0][1]) then field_to_u32(p[i][j][1].value) else other fi
    endfor
    good = if minusCount == 2 && contains && s_next[other] != 2 then good else false fi
    result = if good then true else result fi
  endfor

  assert(result)
  return chres

```

Listing A.1: ZoKrate templates code to prove that a state vector change is valid

A.2 Hasing ZoKrates code template

```
import "hashes/sha256/sha256.zok" as sha256h
import "utils/casts/bool_256_to_u32_8.zok" as bool_to_u32
import "utils/pack/u32/nonStrictUnpack256.zok" as field_to_u32

[[ !!!REPLACE THIS WITH CONSTANTS!!! ]]

struct variables {
  [[ !!!REPLACE THIS WITH VARIABLES!!! ]]
}

struct message_hashes {
  [[ !!!REPLACE THIS WITH MESSAGES!!! ]]
}

const u32 hash_in_len = 8

def main(u32[len_V] s_n,u32 random,variables v,message_hashes msg) -> u32[hash_in_len]:
  u32[8] hash = [[ !!! REPLACE THIS WITH HASH FUNCTION !!! ] ]
  return hash
```

Listing A.2: The ZoKrates template code for hashing

A.3 The root ZoKrates code

```
import "hashes/sha256/sha256.zok" as sha256h
import "utils/casts/bool_256_to_u32_8.zok" as bool_to_u32
import "utils/pack/u32/nonStrictUnpack256.zok" as field_to_u32
import "ecc/babyjubjubParams.code" as context
from "ecc/babyjubjubParams" import BabyJubJubParams
import "signatures/verifyEddsa.zok" as verifyEddsa
import "ecc/proofOfOwnership.zok" as proofOfOwnership
import "./stateChange.zok" as stateChange

[[ !!!REPLACE THIS WITH CONSTANTS!!! ]]

struct variables {
  [[ !!!REPLACE THIS WITH VARIABLES!!! ]]
}

struct message_hashes {
  [[ !!!REPLACE THIS WITH MESSAGES!!! ]]
}

const u32 hash_in_len = 8

def sha256State(u32[len_V] s_n,u32 random,variables v,message_hashes msg) -> u32[hash_in_len]:
  u32[8] hash = [[ !!! REPLACE THIS WITH HASH FUNCTION !!! ] ]
  return hash

def main(public u32[8] h_s_curr,private u32[len_V] s_curr,private u32 r_curr,private variables
  v_curr,private message_hashes msg_curr,private u32[len_V] s_next,private u32 r_next,private
  variables v_next,private message_hashes msg_next,field[2] R,field S,private field[2] A,private
  field sk) -> u32[8]:
  assert(r_curr != r_next)
  u32[8] h_curr = sha256State(s_curr,r_curr,v_curr,msg_curr)
  assert(h_s_curr == h_curr)
```

```

//bool b = if s_curr == [0, 0, 1] then s_next == [0, 0, 0] || s_next == [0, 0, 1] else if s_curr
== [1, 0, 0] then s_next == [0, 1, 0] || s_next == [1, 0, 0] else if s_curr == [0, 1, 0] then
s_next == [0,0,1] || s_next == [0, 1, 0] else false fi fi fi
u32[4] changes = stateChange(s_curr,s_next)
u32 state = changes[3]
[[ !!! REPLACE THIS WITH HASH VARIABLE ASSERTION !!!]]
// Itt a trace már biztosan helyes
BabyJubJubParams context = context()
field[2] pk = keys[state]
assert(pk != [0,0])
assert(A == pk || changes == [0;4])
u32[8] result = sha256State(s_next,r_next,v_next,msg_next)
bool isVerified = proofOfOwnership(A,sk,context) && verifyEddsa(R, S, A, h_curr, result, context)
assert(isVerified)
return result

```

Listing A.3: The root ZoKrates template code of this paper