UNIVERSITY OF TARTU
Institute of Computer Science
Cybersecurity Curriculum

Aivo Toots

Zero-Knowledge Proofs for Business Processes

Master's Thesis (24 ECTS)

Supervisor: Peeter Laud, PhD

Tartu 2020

Zero-Knowledge Proofs for Business Processes

Abstract:
Every day the amount of private and sensitive information shared in the Internet increases. This means that there is also a growing need for solutions to protect this information. There are various effective technologies used to protect or hide private and sensitive information, but even better approach would be to reduce the need for sharing this information through the Internet at all. In many cases, zero-knowledge proofs could be used, replacing the shared sensitive information with proofs based on this information. As zero-knowledge proofs have a lot of potential, but they are not widely used in practical applications yet, this thesis presents a tool that supports two goals – firstly, making systems more secure and privacy-preserving, and secondly, bringing zero-knowledge proofs more into practical applications. This paper describes a tool that allows to prove, based on a description of a system (a process) expressed as a business process model in BPMN notation, that the system (a process) has under certain conditions some stated properties, for example, that there is a flaw in it. For example, one may prove that a purchase process satisfying certain conditions allows one to receive the product without actually paying for it, without disclosing how this is achieved. For constructing and verifying the proofs, the tool uses the zkSNARK protocol provided by libsnark.

Keywords:
BPM, BPMN, zero-knowledge proof, zkSNARK, libsnark

CERCS: P175 Informatics, systems theory

Nullteadmustõestused äriprotsessidele

Lühikokkuvõte:
Iga päev liigub internetis järjest enam privaatset ja tundlikku informatsiooni, millest tulenevalt kasvab vajadus lahenduste järele, mis aitaksid seda informatsiooni kaitsta. Privaatse ja tundliku informatsiooni kaitsmiseks või peitmiseks on mitmeid efektiivseid lahendusi. Veel parem variant oleks aga üldse vähendada vajadust selliste vahendite järele. Nii mõnelgi juhul pakuks alternatiivi rakendused, mis kasutavad nullteadmustõestusi. Sellisel juhul saaks tundliku informatsiooni asemel vahendada läbi interneti hoopis sellel informatsiooni põhjal püstitatud tõestusi. Nullteadmustõestuste rakendamine pakub kasulikke võimalusi, aga paraku kasutatakse neid praktilistes rakendustes veel vähe. Käesoleva töö oluline eesmärk on propageerida nullteadmustõestuste rakendamist erinevates rakendustes ja seeläbi toetada süsteemide turvalisemaks ja enam privaatsust säilitavaks muutmist. Töö tulemusena valminud tööriist kasutab sisendina BPMN notatsioonis väljendatud süsteemi kirjeldust. Seades süsteemile kitsendusi, võimaldab tööriist tõestada, et sellel on teatud omadused, näiteks, et selles leidub viga. Tööriist võimaldab näiteks

tõestada, et teatud tingimustele vastavas ostuprotsessis on võimalik saada kaup kätte ilma selle eest maksmata, sealjuures avaldamata viisi, kuidas see saavutati. Nullteadmustõestuste koostamiseks ja verifitseerimiseks kasutab tööriist libsnark'i teeki.

Võtmesõnad:
BPM, BPMN, nullteadmustõestus, zkSNARK, libsnark

CERCS: P175 Informaatika, süsteemiteooria

Contents

1 Introduction

The amount of information shared across the world increases every day. More and more people share information about their lives in the Internet. On the one hand, people get more used to the possibilities of communication that the Internet provides and they share information more willingly than they would do it in direct face-to-face communication – the dangers are more difficult to notice in the Internet. On the other hand, as the number of new solutions promising to make lives easier increases and more and more of the everyday bureaucracy is moved into the Internet, the more the people are required to share the information about their lives in general, but also, to do that through public channels. This means that the need to protect the private or sensitive information becomes more relevant every day.

There are many technologies to protect and hide the private information, but even better approach would be to replace the need to share the sensitive information with alternatives that would not require sharing sensitive information through public channels at all. One possible alternative for many cases would be using zero-knowledge proofs. Using zero-knowledge proofs, it is possible to make true statements (and provide proofs of these) about the sensitive information to another party without revealing any new information. Provided the proof, another party could verify the alleged statements without receiving any sensitive information through the process. This way, instead of sharing private information through public communication channels, proofs could be shared instead. Unfortunately, without relevant background knowledge, zero-knowledge proofs are not too easy to comprehend and this means that they are not used very much in practice yet.

This thesis seeks to support the mission of reducing the need to share private information in different communication channels, and the goal of bringing zero-knowledge proofs more into use in every-day practical applications. This thesis does not address the need for practical applications using zero-knowledge proofs in every-day applications directly by providing an easy to learn useful tool for everybody to use, but provides a tool that would support designing such, and in general, more secure applications.

This thesis proposes a tool that allows to prove, based on a description of a system (or a process) expressed as a business process model (in BPMN notation), that the system (or a process) has under certain conditions some stated properties, for example, that there is a flaw in it. Based on these (true) statements, a proof could be generated and this could be later verified by another party without disclosing the actual flaw. This would support making different systems more secure and privacy-preserving, reducing the need to disclose sensitive information

through public channels. Implementing the proposed tool is the main goal of this thesis.

2 Background

The goal of this section is to provide a brief introduction into the concepts and technologies, through which the main goal of this thesis is achieved. In addition, introduction to few related works is provided. Following section covers BPMN language, zero-knowledge proofs and a library called libsnark – these form the basis of the implementation tool of this thesis.

Note that definitions in this section are not strictly formal as they get more and more complicated and detailed. For this thesis, all of these details are not relevant, but still, some details are highlighted to support the understanding of the following sections.

2.1 Business Process Model and Notation

Business Process Management (BPM) is a discipline about how work in organisations is being done and how to improve it – it seeks to improve the outcome of the work, and by not focusing on individuals, but rather on whole chains of events, activities and decisions (these chains are called processes) [DRMR13]. In order to express, and through that, improve the outcome of the work of different organisations, there are various ways to model organisation's processes. One widely used practice is to use Business Process Model And Notation (BPMN), a standard developed by the Object Management Group (OMG).

BPMN seeks to provide a notation that is easily understandable and usable for all business users, including analysts who draft the process models, technical users (developers, architects, etc.) who implement the technology that executes these processes, and business managers who manage and monitor these processes [OMG10, OMG11]. BPMN is a language that allows to create detailed models that show step by step how a process is expected to run. It is meant for a quite high-level, rather than too low, system level detailed processes [DDO08]. For the goal of this thesis, systems' descriptions with these characteristics are sufficient.

As BPMN is widely used, it has various different extensions to make it suitable for specific use cases. In terms of this thesis, the approaches that allow to make (business) processes more secure, are relevant. In the context of BPMN, this means extensions that allow to include security perspective more into the models. Examples of this include an extension to model security requirements in business processes [RFMP07], and an extension to visualise movements of private information as it is disclosed to participants of these processes – Privacy-Enhanced Business Process Model And Notation (PE-BPMN) [PMB17, PTMT19].

Under Privacy-Enhanced, it is meant that Privacy Enhancing Technologies are in use, which European Commission [PET07] defines from the perspective of using technologies at the design state of new systems as follows: *The use of PETs could help to design information and communication systems and services in such a manner that minimises the collection and use of personal data and supports making compliance with data protection rules easier. The use of PETs should provide that making breaches of certain data protection rules is more difficult and should also help to detect them.* Representatives of such technologies would be, for example, different encryption and secret sharing schemes.

PE-BPMN provides measures to specify privacy enhancing technologies to be used on process models. Currently, these advances are not yet considered in the implementation tool of this thesis (hopefully will be in the future), but they support the same overall goal – making systems more secure, from all kinds of perspectives.

Even without including extensions, BPMN notation includes considerable amount of different elements and symbols. Only a few of these elements are important for the current version of the resulting tool of this thesis (see Section 4.1 for details), but for overall background and to support better understanding of the context of these few elements, we will next describe the core elements of BPMN (see Figure 1):

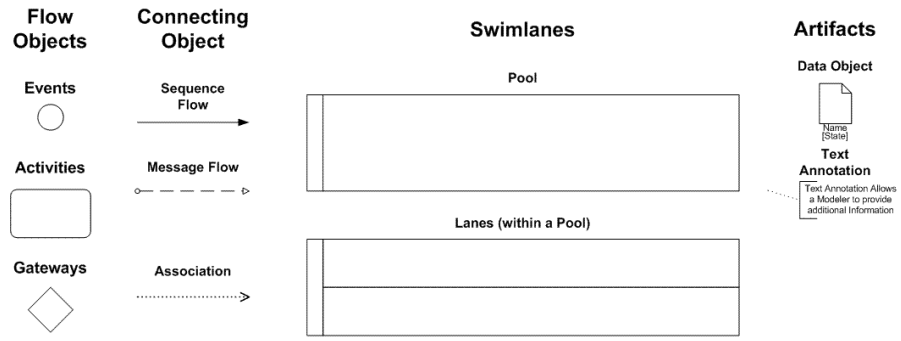

Figure 1. Set of BPMN core elements.
[Modified version of the image from
`http://www.omg.org/bpmn/Samples/Elements/Core_BPMN_Elements.htm`]

Swimlanes

Pools – the participants of the process – are somewhat the basis for all processes, as it is common that processes include various associated parties with different roles. Nevertheless, models without any swimlanes are also used. Lanes within pools are used to emphasise that specific parts of processes are carried out by certain roles. These elements do not affect the work of the implementation tool of this thesis, so they are not restricted to be included in input models.

Flow objects

Flow objects represent the decisions made in the process. In general, there are three groups of this type of elements:

1. Events illustrate different occurrences or situations in a process. There are various different event elements in BPMN notation, but in this thesis, only start (see Figure 4a) and end events (see Figure 4b) are considered – accordingly representing the beginning and ending of the process;

2. Activities describe the work that is being done in a process. The smallest atomic activities are represented by tasks (see Figure 4e);

3. Gateways combine and divide flows in a process – stating where flows are split into two or more branches and also, merging multiple flows into one. Again, there are many different gateways in BPMN notation, but in this thesis, only exclusive and parallel gateways are considered;

    - Exclusive gateways (see Figure 4c) divide the process flow into one or more mutually exclusive paths – based on a condition, only one path can be chosen;
    - Parallel gateways (see Figure 4d) represent two concurrent tasks in a business flow – there is no condition or event evaluated.

Connecting Objects

Connecting objects, such as sequence flows, are used to combine different flow elements, such as events, tasks, gateways, etc., into sequences. The direction of sequence flows fixes the order of activities being done in the process. Sequence flows are relevant for the implementation tool of this thesis – these are important to combine process steps into paths.

While (data) associations express information flow between activities in a specific direction, message flows express message sharing from one participant to another (in a specific direction). Data associations and message flows are not yet supported by the implementation tool of this thesis, but these can be included in input models as these do not directly affect the tool's work.

Artifacts

Artifacts can be used to include additional information about a process. This includes information relevant for certain activities – in this case, data objects, that are used – or just comments or general notes about the model.

In the context of this thesis, BPMN is a way of expressing and visualising processes – it has a rather supporting role. To use the implementation tool of this thesis, it is important to have a general understanding of how to model processes, but detailed specifications of BPMN are not that crucial – this is why only a general overview of the notation has been given.

2.2 Zero-knowledge proofs

Before describing what are zero-knowledge proofs, a short description of interactive proofs is given. It introduces the prover-verifier protocol and sets basis for the following description of zero-knowledge proofs.

2.2.1 Interactive proofs

An interactive proof system for a language L is a protocol between two algorithms – a prover P that is computationally unconstrained, and a verifier V with polynomial running time. The prover and the verifier are both given some instance x and the prover wants to prove that $x \in L$. In order to prove that, the prover and the verifier need to communicate with each other back and forth (this makes it interactive).

The prover should be able to prove a true statement and should not be able to prove false one. In case $x \in L$, it should be possible to prove true statement in a way that the verifier can verify the proof in polynomial time – an honest prover, following the protocol, should be able to convince the verifier (completeness). If $x \notin L$, it is not possible for the prover, even if it does not follow the protocol, to convince the verifier of the contrary (soundness) [GMR85]. While the prover can be untrusted, it is assumed that the verifier can be always trusted, that it is honest [Fei92].

Though it was previously mentioned that a prover is computationally unconstrained, there are cases, when it is not like that. For example, there are simple interactive proofs for all languages in the class **NP**. The prover generates the witness and hands it over to the verifier. Alternatively, we could think of a set-up, where the prover has somehow obtained the witness from elsewhere, as the proof that $x \in L$. In this case, the prover may also be polynomially bounded.

2.2.2 Zero-knowledge proofs

Compared to standard interactive proofs, zero-knowledge proofs have one extra condition – zero-knowledge – if the statement is true, verifier learns nothing else than the fact that it is true. An interactive proof system is called zero-knowledge if it succeeds in proving the given statements, but does not reveal anything else [Fei92]. This means that the prover convinces the verifier of the solution without actually giving the solution. Zero-knowledge proofs convince and yield nothing else than that the statement is valid.

Zero-knowledge proof is "for all practical purposes, *whatever* can be efficiently computed after interacting with a zero-knowledge prover and can be efficiently computed when just believing that the assertion it claims is indeed valid. (In *whatever* we mean not only the computation of functions but also the generation of probability distributions.)" [GMW91]. This means that it is possible to "imitate" interaction between an honest prover and the verifier by just believing that the statement being proved is true – there exists some sort of a simulator for any verifier (who has no access to the prover), which can reproduce the communication between the prover and the verifier.

The difficulty in zero-knowledge proofs is, that it is trivial to prove the possession of some information by just revealing it, but it is a challenge to prove having that information without revealing the information itself or anything additional.

In addition to zero-knowledge, there are three main ingredients that make zero-knowledge proofs different from more traditional proofs – this is stated in the article Non-interactive zero-knowledge by Blum et al. [BFM88]:

1. Interaction. The prover and the verifier communicate back and forth;
2. Hidden Randomization: There is unpredictability for the prover in what the verifier does (hidden coin tosses);
3. Computational Difficulty: There is computation difficulty of some other problem included into the proof.

As these ingredients make it difficult to implement zero-knowledge proofs, an easier to implement approach, non-interactive zero-knowledge proofs have been proposed. In article Quadratic Span Programs and Succinct NIZKs without PCPs [GGPR13] (page 21) is given a definition of Non-Interactive Zero Knowledge:

Definition 1. "Non-Interactive Zero Knowledge. Let R be a binary relation which consists of pairs (u, w), where u is a statement and w is a witness. Let f be a function such that $f(u, w) = 1$ iff $(u, w) \in R$. Let L be the language that consists of statements with valid witnesses for R. A non-interactive zero knowledge argument for the relation R, or the function f, consists of the triple of polynomial time algorithms (K, P, V):

- K takes a security parameter κ as well as the maximum size of a statement n and outputs a common reference string crs;
- P takes as input the CRS crs, a statement u and a witness w, and outputs an argument π;
- V takes as input the CRS crs, a statement u and its proof π, and either accepts or rejects the proof."

The general idea of the defined protocol is visualised in Figure 2. To put it into the context of the application tool of this thesis, the transition (R) is the implementation application of this thesis itself – it takes inputs x (in the definition, u) (instance) and w (witness), which is basically information needed to prove the statements described in Sections 4.2 and 4.3.

Generator, prover and verifier (in the definition K, P and V) are three polynomial time algorithms. The generator uses the instance x to produce common reference string CRS (in the definition, crs), which in the implementation tool of this thesis is a combination of three keys – a proving key, a verification key and a processed verification key. The prover uses the instance x, the witness w and the proving key to generate a proof π. The verifier uses two verification keys and the proof π to verify the proof. If the verification is successful ($(u, w) \in R$), an accept (1) is returned, and if not, a reject (0) is returned.

Definition 1 already describes the main idea of the protocol used in the application tool of this thesis, but actually there is a more strict protocol in use – Zero-Knowledge Succinct Non-interactive ARgument of Knowledge (zkSNARK). While non-interactive zero-knowledge satisfies following properties: completeness, soundness, zero-knowledge and non-interactivity, the protocol used in the implementation tool of this thesis satisfies in addition to the listed four, the two following

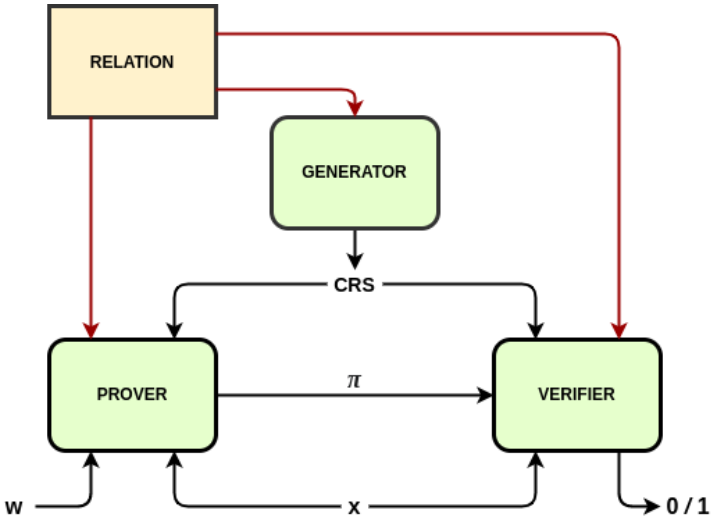

Figure 2. What all parties see and produce.

properties: succinctness and proof of knowledge [BCI+13]. Succinctness means that the proof is short and easily verifiable, and proof of knowledge means that the prover proves that a statement is true and also knows *why* it is true [libb].

Also, regarding the difference between zero-knowledge proofs and arguments of knowledge, it is said in the article Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting [BCC+16] that "In general proofs can only have computational zero-knowledge, while arguments may have perfect zero-knowledge". In terms of soundness, for a proof, it holds against a computationally unbounded prover, but for an argument, it holds against a polynomially bounded prover [NOV06].

2.3 Libsnark

In general, libsnark is a C++ library that provides a programming framework for creating zkSNARK applications. Based on the repository page of libsnark at `https://github.com/scipr-lab/libsnark`, the library provides:

- a C++ implementation of seven general-purpose proving systems – for example, preprocessing SNARK-s for languages of arithmetic circuits and Boolean circuits, also, a preprocessing zkSNARK for the NP-complete language "R1CS" (Rank-1 Constraint Systems). R1CS is a language similar to arithmetic circuit satisfiability – this language is relevant for this thesis;

- two libraries – gadgetlib1[1] and gadgetlib2[2], which are used for constructing R1CS instances using modular "gadget" classes. While the gadgetlib1 is a low-level library, covering all features of the preprocessing zkSNARK for R1CS, the gadgetlib2 is a library for constructing systems of polynomial equations and R1CS instances. In the application tool of this thesis, gadgetlib2 is used. The gadgetlib2 library provides fewer useful gadgets, but is better documented. In the application tool of this thesis, the following gadgets are used (see Section 5.2.4 for more details):
  - *EqualsConst_Gadget*, for checking if a variable has a value of a certain constant. It returns 1 (true) if the constant is equal to a variable;
  - *LooseMUX_Gadget*, for reading from one array, using indices (index) from another array (variable). It returns 1 (true) if the index is in the bounds of the array and the reading is successful, and returns 0 (false) if the index is not in the bounds of the array;
  - *Comparison_Gadget*, for comparing values of two variables. It returns information about whether the value of one variable is less than the value of the other OR whether the value of one variable is less or equal than the value of the other variable. There are two values returned – 1 or 0 for the "less than" condition and 1 or 0 for the "less or equal than" condition;
  - *InnerProduct_Gadget*, for multiplying two vectors together, with the result being a scalar. It takes in two vectors and returns a scalar;
  - *AND_Gadget*, for computing AND of two (or more) variables. It takes in two or more variables (for example, the (boolean) values returned by other gadgets) and returns 1 (true) if they are all equal to 1 (true);
  - *OR_Gadget*, for computing OR of two (or more) variables. It takes in two or more variables and returns 1 (true) if they are all equal to 1 (true). Similary to the *AND_Gadget*, the inputs can be the (boolean) values returned by other gadgets.
- examples of applications that use different proving systems, some of which were mentioned above. Unfortunately, there are not too many of these examples and they tend to be in a simple form, where, for example, generator, prover and verifier algorithms are put into one executable, though they should be run separately.

[1] `https://github.com/scipr-lab/libsnark/tree/master/libsnark/gadgetlib1`
[2] `https://github.com/scipr-lab/libsnark/blob/master/libsnark/gadgetlib2`

Based on the libsnark's description [libb], libsnark uses the preprocessing zk-SNARK (ppzkSNARK) scheme, which means that using the library requires following these four steps:

1. Expressing the statements to be proved as R1CS (or any other language supported);
2. Using libsnark's generator algorithm to create public parameters (CRS) – this includes the proving key, the verification key and the processed verification key;
3. Using libsnark's prover algorithm to create proofs of true statements about the satisfiability of the R1CS;
4. Using libsnark's verifier algorithm to verify the proof of initial statements.

Details about how the implementation tool of this thesis follows these four steps are described in Section 5.

2.4 Related work

There has been a considerable amount of research done on business processes and zero-knowledge proofs separately, but not about these two combined. As business process management is more widely covered, this section focuses on zero-knowledge related works – more precisely, applications using libsnark, because this is one of the most mature libraries for creating zero-knowledge proof applications, and because it is used in the implementation tool of this thesis. Nevertheless, there are not too many applications using libsnark.

One, maybe the most widely known application that uses libsnark is a Zcash[3]. "Zcash is an implementation of the Decentralized Anonymous Payment scheme Zerocash [BSCG+14], with security fixes and improvements to performance and functionality. It bridges the existing transparent payment scheme used by Bitcoin[4] with a shielded payment scheme secured by zk-SNARK" [HBHW]. This means that using Zcash, it is possible to make direct transactions between users that does not disclose the amount, origin or destination.

Differently from, for example, Bitcoin, blockchains of Zcash contain pseudo-random values, ciphertexts and zero-knowledge proofs – transaction information

[3] `https://z.cash/`

[4] `https://bitcoin.org/en/`

contains proofs that can be verified by everybody interested. This is a nice development for protecting private information in blockchains, as for example, in European Union, General Data Protection Regulation (GDPR)[5] requires that it must be possible to take back the private information given out to be processed. This is not typically possible for most of the blockchains, but the approach of Zcash prevents this kind of problem – it prevents adding private information into blockchains.

Another example of an application using libsnark is a BlockMaze – "an efficient privacy-preserving account-model blockchain based on zk-SNARKs" [GWY+19]. Similarly to the Zcash, it hides transaction amounts, origins and destination. BlockMaze seeks to protect account-model blockchains like, for example, Ethereum. In fact, the implementation of BlockMaze[6] has been built on top of libsnark and Go-Ethereum[7] blockchain.

Also, there is one not directly blockchain related application built on top of libsnark library: ZKlaims. ZKlaims is "a system that allows users to present attribute-based credentials in a privacy preserving way" [SKSB19]. Attribute-based credentials are a scheme of authentication mechanism that provides a way to authenticate different attributes about an object in a privacy preserving manner, without disclosing any other information about that object [PPO]. Using ZKlaims, it is possible to prove statements related to credentials issued by third parties, without verifier seeing the contents of these credentials. The authors of ZKlaims claim that it could be exchanged via fully decentralised services, for example, peer-to-peer networks based on distributed hash tables (DHTs) or blockchains [SKSB19].

As there are not too many applications built on top of libsnark or other similar systems yet, this thesis seeks to contribute to bringing zero-knowledge proofs more into practice.

[5] `https://gdpr-info.eu/`
[6] `https://github.com/Agzs/BlockMaze`
[7] `https://github.com/ethereum/go-ethereum`

3 Problem

In general, this section describes the motivation behind this thesis. This includes three topics – a problem statement to describe what this thesis seeks to achieve; a motivation stating a simple use case where the implementation tool of this thesis could be used, together with a list of reasons why this topic is fascinating for the author of this thesis, and last, but not least, approach describing how the idea is formed into an implementation tool.

3.1 Problem statement

Every day, different systems use more and more sensitive or private information, and users of these systems expect that the information is handled very carefully and disclosed to only relevant parties. For example, in European Union, this is strictly regulated by the GDPR. Following these rules is required by the law and in theory users should not need to worry, but in practice, sensitive or private information is leaked through unsecure systems every day.

Using zero-knowledge proofs in different systems would reduce the need to share sensitive information over different unsecure or secure communication channels. Reducing the need to move sensitive information in or between systems, in general, makes these systems more secure.

Today, there are already some applications using zero-knowledge proofs, but these are still rather exceptional. Most of these applications are related to making cryptocurrencies (and blockchains in the bottom) more secure and privacy preserving – few examples are mentioned in Section 2.4. Also, there are various tools to express, plan and simulate systems using business process modelling. Yet, possibilities and potential of combining zero-knowledge protocols and business processes – this has not been researched much.

Proposing an approach and developing a tool to connect BPMN models with zero-knowledge proving systems (with R1CS sequences in between), seeks to carry forward the research in both, business processes and zero-knowledge proofs, by bringing in new ideas and showing that there is a value in combining these two. A goal is to bring zero-knowledge proofs more into practical use in different technologies and through that – make different systems more secure and privacy preserving.

3.2 Motivation

The main motivation for this thesis comes from a certain use case: there is a security flaw in some system and a person, a white or grey hat hacker for example, knows the flaw and wishes to prove that, but does not wish to reveal specific details of the flaw to the owner of the system.

In practice, there are mainly two ways how system owners react to declarations of finding flaws in their systems. One option is to acknowledge and praise these persons responsible for finding flaws – there are also monetary prizes (bug bounties) in many cases. Another option is that persons responsible for finding flaws are accused of attacking systems, threatened with court cases, and in some cases, prosecuted.

To make real life systems more secure, specialists (white hat hackers) need to work on real systems and it would help if there were ways to first prove that one has found a flaw and if the system appreciates it, only then reveal it, without being afraid of being prosecuted. This thesis offers one possible approach to support proving existence of flaws in different systems without actually revealing the flaws. This approach is described in Section 3.3.

The motivation for choosing this particular approach derives from three following aspects.

Firstly, the author of this thesis has worked multiple years on developing systems (PLEAK[8]) that seek to detect privacy leakages from systems analysing business process models describing these systems. This means that the author of this thesis has interest in detecting flaws in systems and experience in using BPMN notation models for this purpose.

Secondly, the author of this thesis wishes to research modern cryptography achievements and support bringing these solutions into practice.

Thirdly, the author of this thesis works as a programmer at a research and development intensive ICT company Cybernetica AS[9], on DARPA[10]'s project PROVENANCE under SIEVE program[11]. One main goal of this project is to

[8] `https://pleak.io`

[9] `https://cyber.ee/`

[10] `https://www.darpa.mil/`

[11] `https://www.darpa.mil/program/securing-information-for-encrypted-verification-and-evaluation`

develop ways to bring zero-knowledge proofs more into real life practice – the SIEVE program "seeks to advance the state of the art in ZK proofs to enable complex, DoD-relevant applications. SIEVE will use ZK proofs to enable the verification of capabilities relevant to the DoD without revealing the sensitive details associated with those capabilities" [SIE].

3.3 Approach

Supporting the goals described in Section 3.1, this thesis offers an approach that offers a way to prove flaws (without actually revealing them) in systems using BPMN notation models to describe these systems and zero-knowledge proofs to prove the existence of flaws under certain circumstances.

In more detail, the approach is to use BPMN notation models to describe systems and different processes, and use regular expressions to describe unwanted behaviours of the processes – to specify a set of traces made up of the elements of the process, that are considered bad. For example, fixing that all steps of the process are allowed to be taken except certain few, or that certain steps must be taken before others. This helps to represent real life limitations in processes that sometimes could not be derived from the model by just analysing the structure of it.

A flaw in a system or a business process is demonstrated by a run (a path or a trace) of a business process that belongs to the language of unwanted runs. More concretely, a certain flaw in the system is expressed as a string (that is matched by the regular expression) that describes a certain path (a trace) through the process. This approach shows that business processes and regular expressions intersect in the way how they "act" – regular expressions could outline unwanted situations in business processes.

To prove that a certain run (a trace) belongs to the business process semantics and is accepted by a nondeterministic finite automaton (described by the regular expression), zero-knowledge proofs are used. The steps that are required to prove that a using certain path (certain steps in a specific order) under restrictions (determined by the regular expression) to pass through the process are described in Section 4.

4 Design

For the tool implemented in this thesis to be able to serve its purpose, the inputs provided by the user need to follow certain requirements. Following subsections describe the initial requirements for the inputs, and also the principal steps of the proofs under the implementation tool of this thesis. There are examples provided next to the two proofs to make it easier to understand, why each step is necessary and how the steps support the overall goal.

4.1 Prerequisites

In order to understand what is expected to be provided as inputs to implementation tool of this thesis, following paragraphs define what is here meant by an input business process and what are the requirements for it. Also, the requirements for the input nondeterministic finite automaton are listed. Finally, it is described how the business process and automaton are used together in the proofs under the implementation tool of this thesis.

For an introduction, an illustration of a business process in the form of a BPMN model is provided in Figure 3 (note that in terms of this thesis, the content of this model is not relevant, it is just used as a simple example). The definition and requirements of a business process in terms of the application tool of this thesis are following:

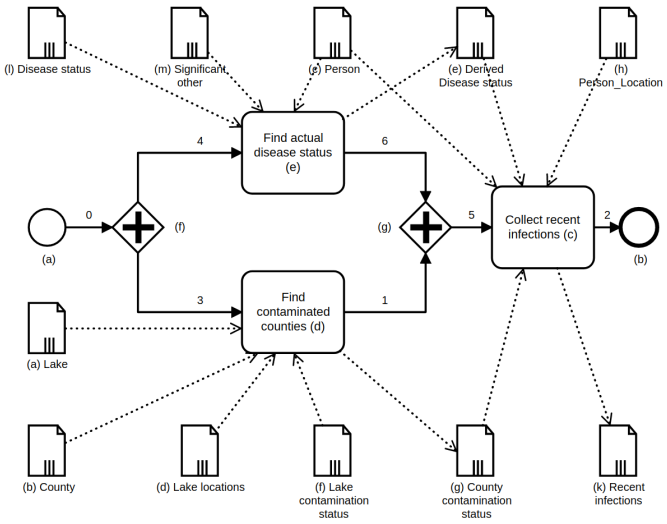

Figure 3. An example of BPMN model.
[Modified version of the model from
`https://pleak.io/app/#/view/aPCvSrCZIgwB-vtYo2hO`]

A business process consists of *Start event*, *End event*, *Exclusive gateway*, *Parallel gateway* and $Task$ elements (see Figure 4 (a, b, c, d, e)), connected by $Sequenceflow$ elements (see Figure 5).

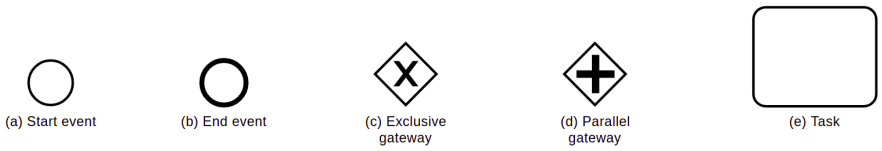

Figure 4. Vertex elements relevant in this thesis.

Given $n \in \mathbb{N}$, we write $[n]$ to mean a set $\{1, \ldots, n\}$.

Definition 2. We let the business process to be a tuple $(V, E, \sigma, \tau, \lambda, \overrightarrow{p}, \overleftarrow{p})$, where

- V is the set of *vertices*, and E the set of *edges* of the process;
- $\sigma : E \to V$ and $\tau : E \to V$ give the *starting* and *ending* vertex of an edge;
- $\lambda : V \to \{task, excl, paral, start, end\}$ gives the *type* of a vertex;

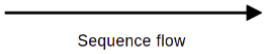

Figure 5. Edge element relevant in this thesis.

- $\overrightarrow{\mathsf{p}_v} : \sigma^{-1}(v) \to [|\sigma^{-1}(v)|]$ and $\overleftarrow{\mathsf{p}_v} : \tau^{-1}(v) \to [|\tau^{-1}(v)|]$ are defined for those $v \in V$, where $\lambda(v) = \text{excl}$, and they fix an ordering on the incoming and outgoing edges of the vertex v.

In this work, we restrict the business processes so, that they satisfy the following constraints on the numbers of incoming and outgoing edges of the vertices:

- start event: 0 incoming and 1 outgoing;
- end event: 1 incoming and 0 outgoing;
- task: 1 incoming and 1 outgoing;
- parallel gateway: 1 incoming and 2 outgoing OR 2 incoming and 1 outgoing;
- exclusive gateway: at least 1 incoming and at least 1 outgoing.

Let $E \to \mathbb{N}$ be a *state* of a business process. The structural operational semantics (small-step semantics) of a business process is a binary relation on such states, defining one *step* of a business process. This means that the binary relation tells from which state it is possible to get to another state with one step. A state is considered as a number of tokens on each edge. In the beginning of the business process, there is 1 token on the outgoing edge of a start event, the other edges have none.

Considering that every step is related to the changes on token numbers on the edges of connected vertices, let Σ be the set of all possible token number changes on the edges of each vertex of a business process.

$$\Sigma = \{v \in V \mid \lambda(v) \neq \text{excl}\} \cup \{(i, v, j) \mid v \in \lambda^{-1}(\text{excl}), i \in [|\sigma^{-1}(v)|], j \in [|\tau^{-1}(v)|]\} \ .$$

Let there be a public procedure to list the elements of set Σ. This procedure equates the elements of set Σ with numbers $1, 2, \ldots, |\Sigma|$.

The trace semantics of a business process is a language $S \subset \Sigma^*$. A trace $t \in \Sigma^*$ is an element of S if and only if changes on token numbers on the edges of connected vertices are possible in the order specified in t. The t can be a an actual trace (be an element of S) only if the number of tokens on the edges of connected vertices

does not become negative.

Note that while processing the trace t, on every step, moving from one vertex to another, token number changes on the edge connecting two vertices. For example, assuming that the first element in a trace t is of type start, moving to another element, the token number on the outgoing edge of the start element is increased by one. Taking next step, the token number on this edge (outgoing edge of the start) is decreased by one and increased by one on the outgoing edge(s) of the next element – it can be understood as taking a token and passing it on.

In general, on every step, the number of tokens on incoming edges of an element (vertex) is decreased by one and on outgoing edges increased by one. The numbers of possible (accepted) incoming and outgoing connections for each element were listed before. Nevertheless, there is one exception – exclusive gateways. For each exclusive gateway, no matter how many incoming or outgoing edges it has, the number of tokens changes only on one incoming edge (decreased by one) and on one outgoing edge (increased by one) at a time. Also, when moving through parallel gateways, it can be understood as the one token is split into two tokens or two tokens are merged backed together into one – note that when token is split, there will be $1 + 1$ tokens, not $0.5 + 0.5$ tokens, and when merged, $1 + 1$ tokens becomes 1 token.

Continuing with defining the input automaton, let A be a nondeterministic finite automaton with:

- the finite set Q of states that we equate with a set $\{1, 2, \ldots, |Q|\}$,
- the set $I \subseteq Q$ of initial states,
- the set $F \subseteq Q$ of accepting states,
- the transition relation $\delta \subseteq Q \times \Sigma \times Q$.

A accepts a language $L(A)$, defined by a regular expression, in a standard way, for example, as described in a book Introduction to Compiler Design [Mog17] (page 2).

Using zero-knowledge, we want to prove following statements. Given some language $L \subseteq \Sigma^*$ accepted by a NFA, we prove statements in the form of $L \cap S \neq \emptyset$.

In order to prove a statement $L \cap S \neq \emptyset$, we present string $[\![w]\!]$ as a witness of the proof and show that $[\![w]\!] \in L$ and $[\![w]\!] \in S$. We write $[\![\cdot]\!]$ to mean that w is private. By private we mean that w is known only to the prover. String $[\![w]\!]$ is given

as a sequence of its characters: $[\![w_1]\!], [\![w_2]\!], \ldots, [\![w_{|w|}]\!]$, which are all elements of set Σ (numbers from set $[|\Sigma]\!]$). We assume that the length of sequence $[\![w]\!]$ is public.

Showing how $[\![w]\!]$ belongs to languages S and L is done in Sections 4.2 and 4.3.

4.2 A trace (witness) belongs to the business process semantics

In order to prove that a trace belongs to the business process semantics, a business process (input as .bpmn model from the user) is required. This business process (.bpmn model) must be parsed to check if all requirements are met (see Section 4.1) and to find all vertices (steps) and edges from the model.

Given that the number of incoming and outgoing edges of vertices is limited, the number of tokens changes on maximum three edges for each step of the business process.

Let P be an array, indexed by the elements of Σ (numbers $1, 2, \ldots, |\Sigma|$). Each element of P is a triple of the elements of set $\mathcal{N}$, where

$$\mathcal{N} = (\{\text{incr}, \text{decr}\} \times E) \cup \{\text{nothing}\} \ .$$

An element $P[i]$ shows how and on which edges token numbers are changed when we "take" step i. The order of the values in a triple is not important.

Given private information:

- string $[\![w]\!]$ (a trace),

and public information:

- array P,
- the length of string $|w|$,
- the number of steps on the model,
- the number of edges on the model,

the proof consists of five following steps, which can be, and are implemented (see Section 5) using the zkSNARK construction through the libsnark library:

Step 1:

Read from the array $[\![P]\!]$ using indices $[\![w_1]\!], \ldots, [\![w_{|w|}]\!]$. This reading gives $|w|$ private values that all belong to set $\mathcal{N}^3$. We consider these values as $3|w|$ private values $[\![L_1^\#]\!], \ldots, [\![L_{3|w|}^\#]\!]$ from set $\mathcal{N}$. This means that reading by index $[\![w_1]\!]$, we get values $[\![L_1^\#]\!], [\![L_2^\#]\!], [\![L_3^\#]\!]$, reading by index $[\![w_2]\!]$, we get values $[\![L_4^\#]\!], [\![L_5^\#]\!], [\![L_6^\#]\!]$ etc.

This step gives us (groups of) token changes on edges in timely order. We can see for a moment in time (a step), on which edges, the token number is increased, and on which, decreased.

Step 2:

Define L, an array of pairs, with $3|w|+|E|$ elements. For the first $3|w|$ elements, let the left component of $[\![L_i]\!]$ be $[\![L_i^\#]\!]$ and the right component of $[\![L_i]\!]$ be i. For the last $|E|$ elements, let the left component of $[\![L_{3|w|+i}]\!]$ be (init, i) and the right component of $[\![L_{3|w|+i}]\!]$ be 0.

In this step, we start to group the token changes by edges. "init" values are added (for each edge) to mark the beginning of the change in the number of tokens for an edge. The right component of some $[\![L_i]\!]$ still denotes the time of change.

Step 3:

Sort array $[\![L]\!]$ based on the following order:

$$\begin{aligned} ((\_, e_1), i_1) &\leq ((\_, e_2), i_2) && \text{if } e_1 < e_2 \text{ OR } e_1 = e_2 \wedge i_1 \leq i_2 \\ (\text{nothing}, i_1) &\leq (\text{nothing}, i_2) && \text{if } i_1 \leq i_2 \\ \_ &\leq (\text{nothing}, \_) \ . \end{aligned}$$

In this step token changes are sorted in a way that all changes for one edge are grouped and preceded by "init" value (denoting the beginning of the change in the number of tokens for an edge).

Step 4:

Remove unnecessary information from an array $[\![L]\!]$: let $[\![L']\!]$ be an array (with

the same length as the array $[\![L]\!]$), where each element belongs to set $\mathbb{Z} \cup \{\text{init}\}$:

$$L'_i = \begin{cases} 1, & \text{if } L_i = ((\text{incr}, \_), \_) \\ -1, & \text{if } L_i = ((\text{decr}, \_), \_) \\ 0, & \text{if } L_i = (\text{nothing}, \_) \\ \text{init}, & \text{if } L_i = ((\text{init}, \_), \_) \end{cases}$$

Step 5:

Find the prefixsum of the array $[\![L']\!]$, where addition of integers is defined as usual. In addition, let $x + \text{init} = \text{init}$ and $\text{init} + x = x$ for each x. Let the array $[\![M]\!]$ be the prefixsum of $[\![L']\!]$. Verify that there are no negative values in the array $[\![M]\!]$.

In this step, by finding prefixsums based on these conditions, we calculate (prefix sums) the sum of token changes for each edge – this means that if token number on one edge is increased (+1) and then decreased (-1), the prefixsum is 0. In case the sum is negative, there have been (at least) two non-consecutive (on the model) steps made or some step is missing. For example, when taking a step from one task (the number of tokens on the incoming edge is decreased and the number of tokens on the outgoing edge is increased – like moving the token from one edge to another) to another task, the sum of token changes for the edge between these two tasks should be 0 – in case it is not, an incorrect (not allowed) step has been taken. For another example, in case of exclusive or parallel gateway steps when there are multiple incoming and/or outgoing connections, but one or more required steps have not been taken, the sum of token changes becomes invalid.

In case following all previous steps has been successful, we have proved that a trace (witness) belongs to the business process semantics.

4.2.1 Example

In order to make the previously listed steps more understandable, based on the example model in Figure 3, let the input information be following:

- let the trace $[\![w]\!]$ be a string "a,f,e,d,g,c,b" (note that all steps of the model are included in the trace, this is just used for a simple example);
- let the array P be listed in Table 2.
- and let the indices $[\![w_1]\!], \ldots, [\![w_{|w|}]\!]$, based on trace $[\![w]\!]$ and array P, be listed in Table 1.

Table 1. Indices $[\![w_1]\!], \ldots, [\![w_{|w|}]\!]$ based on the trace $[\![w]\!]$.

0	5	4	3	6	2	1

Table 2. P – a set of vectors of token changes on the edges of vertices on the model.

1	0	0	-1	0	-1
-1	2	0	-1	0	-1
-1	5	1	2	0	-1
-1	3	1	1	0	-1
-1	4	1	6	0	-1
-1	0	1	3	1	4
-1	1	-1	6	1	5

In Table 2 are listed the token changes for the edges of vertices on the example model (see Figure 3). One line of a table contains information about maximum three edges (three pairs) as for each vertex of the model, token numbers can change on maximum three connected edges. The number of possible incoming and outgoing connections (edges) for each type of element (vertex) was listed before (see Section 4.1). In case the number of incoming and outgoing connections combined for an element is less than three, "placeholders" are added to always have three pairs – these "empty" token changes are marked with pairs (0,-1). This means that, as in the first row of the table, there are pairs (1,0),(0,-1),(0,-1), the last two pairs are "empty".

With the exception of the "empty" pair, the first component of a pair shows whether the number of tokens is increased by one (1) or decreased by one (-1). The second component is an identifier on an edge. Knowing that only for elements of type start and end, as there are no other connections than the one incoming or outgoing connection, we can say that the first row of the table describes token number changes on start element (as it has only one outgoing edge and on outgoing edges token numbers are increased). How the numbers of tokens on the edges of model elements change was described earlier in Section 4.1.

Continuing, the second row describes token changes on the edges of an element of type end, as two of the pairs are again "empty" and the first component of the first pair describes decrease (which describes an incoming connection). Continuing the same logic, we can see for each row, whether it describes token changes on start, end, task, excl or paral type of element. For easier understanding, all edges on the example model (see Figure 3) are numbered as their identifiers appear in the pairs in Table 2. Note that these identifiers do not appear in the table in any relevant order,

as the values represent internal identifiers of these elements in the client application.

Based on the input information (indices $[\![w_1]\!], \ldots, [\![w_{|w|}]\!]$ and set P), the explanations and results of proof steps are following:

Step 1:

The result of reading from the array $[\![P]\!]$ using indices $[\![w_1]\!], \ldots, [\![w_{|w|}]\!]$ is listed in Table 3. These are token changes on edges of vertices (steps) in timely order. This means that we can see for a moment in time (a step), on which edges the token number is increased, and on which, decreased.

Table 3. Elements read from P using indices $[\![w_1]\!], \ldots, [\![w_{|w|}]\!]$ from the trace $[\![w]\!]$.

1	0	0	-1	0	-1
-1	0	1	3	1	4
-1	4	1	6	0	-1
-1	3	1	1	0	-1
-1	1	-1	6	1	5
-1	5	1	2	0	-1
-1	2	0	-1	0	-1

Step 2:

In this step, we start to group the token changes by edges. The six values in a row are divided into pairs and these pairs are numbered. Also, "init" (here represented with numbers 999999999) values are added (for each edge) to mark the beginning of the change in the number of tokens for an edge. The result of this step, L, is listed in Table 4.

Table 4. The set L.

1	0	1
0	-1	2
0	-1	3
-1	0	4
1	3	5
1	4	6
-1	4	7
1	6	8
0	-1	9
-1	3	10
1	1	11
0	-1	12
-1	1	13
-1	6	14
1	5	15
-1	5	16
1	2	17
0	-1	18
-1	2	19
0	-1	20
0	-1	21
999999999	0	0
999999999	1	0
999999999	2	0
999999999	3	0
999999999	4	0
999999999	5	0
999999999	6	0

Step 3:

As already stated before, in this step the token changes are sorted in a way that all changes for one edge are grouped and preceded by "init" (999999999) value (denoting the beginning of the change in the number of tokens for an edge). Here we can see that there are seven "init" values, as there are seven edges in the model, and token changes for edges are following these "init" values. The result of this step is listed in Table 5.

Table 5. A sorted version of set L.

999999999	0	0
1	0	1
-1	0	4
999999999	1	0
1	1	11
-1	1	13
999999999	2	0
1	2	17
-1	2	19
999999999	3	0
1	3	5
-1	3	10
999999999	4	0
1	4	6
-1	4	7
999999999	5	0
1	5	15
-1	5	16
999999999	6	0
1	6	8
-1	6	14
0	-1	2
0	-1	3
0	-1	9
0	-1	12
0	-1	18
0	-1	20
0	-1	21

Step 4:

In this step we remove unnecessary information, as at this point only the token changes are relevant. In this part of the proof, we do not need to know any more on which edges specifically these changes are taking place. The result of this step is listed in Table 6.

Table 6. Unnecessary information removed from the set L, only token changes left.

999999999
1
-1
999999999
1
-1
999999999
1
-1
999999999
1
-1
999999999
1
-1
999999999
1
-1
999999999
1
-1
0
0
0
0
0
0
0

Step 5:

In this step, by finding prefixsums (based on conditions stated before, in Section 4.2 (step 5)), we calculate (prefixsums) the sum of token changes for each edge – this means that if token number on one edge is increased (+1) and then decreased (-1), the prefix sum is 0. If there were negative values, something would have been wrong. As we can see in the result Table 7, there are only non-negative values in the list and that means that all (required) steps were made consecutively and we have proved that the trace (witness) belongs to the business process semantics.

Table 7. Prefixsums of the set listed in Table 6.

999999999
1
0
999999999
1
0
999999999
1
0
999999999
1
0
999999999
1
0
999999999
1
0
999999999
1
0
0
0
0
0
0
0
0
0

How this proof has been implemented in the result tool of this thesis is explained in Section 5.2.4.

4.3 A trace (witness) is accepted by an automaton

In order to prove that a trace is accepted by an automaton, a regular expression string (input from the user) is required. This string must be converted into a nondeterministic finite automaton (A) where ε-transitions are removed (ε-transitions are not supported by this tool).

$\delta \subseteq Q \times \Sigma \times Q$ being the transition relation of a nondeterministic finite automaton A, let Δ be an array listing the elements of δ, where every element is a triple of state, character, state (three numbers). We write Δ_i^1, Δ_i^2, Δ_i^3 to mean three components of an element on position i in Δ.

To find how $[\![w]\!]$ (a trace) is accepted by an automaton A, prover needs to find indices $[\![k_1]\!], \ldots, [\![k_{|w|}]\!] \in [|\delta|]$ in such a way that when processing a character of $[\![w]\!]$ on position i, a transition $[\![\Delta[k_i]]\!]$ is used. This means that there must exist (and the prover has to find it) such a sequence of transitions (with indices $[\![k_1]\!], \ldots, [\![k_{|w|}]\!] \in [|\delta|]$) that all characters of the trace are consecutively read by the automaton, leading the automaton to an accepting state.

Given private information:

- string $[\![w]\!]$,
- indices $[\![k_1]\!], \ldots, [\![k_{|w|}]\!]$,

and public information:

- the length of string $|w|$,
- the number of transitions of an automaton A,
- $|Q|$, the number of states of an automaton A,
- I, initial states of an automaton A,
- Δ, the transition relation of an automaton A (triples),
- F, accepting states of the automaton A,

the proof consists of five following steps:

Step 1:

Reading from the array $[\![\Delta]\!]$ (see Table 9) using indices $[\![k_1]\!], \ldots, [\![k_{|w|}]\!]$. This reading gives us values $[\![K_1]\!], \ldots, [\![K_{|w|}]\!]$ – each value being of the same type (a triple) as the elements of $[\![\Delta]\!]$.

Step 2:

Verify that $[\![K_1^1]\!]$ belongs to set I.

Step 3:

Verify that $[\![K_{|w|}^3]\!]$ belongs to set F, where the superscript 1 denotes the first component of the triple.

Step 4:

Verify that for each $i \in [|w| - 1]$, $[\![K_i^3]\!] = [\![K_{i+1}^1]\!]$.

Step 5:

Verify that for each $i \in [|w]\!]$, $[\![K_i^2]\!] = [\![w_i]\!]$.

In case all previous steps have been successful, we have proved that a trace (witness) is accepted by an automaton.

4.3.1 Example

In order to make the previously listed steps more understandable, based on the example model in Figure 3, let the input information be following:

- let the trace $[\![w]\!]$ be string "a,f,e,d,g,c,b" (note that all steps of the model are included in the trace, this is just used for a simple example);
- let the automaton A be constructed from a regular expression "(a|b|c|d|e|f|g)*" (note that in the regular expression all steps of the business process are allowed, this is just used for a simple example; also, this is not the smallest automaton for this regular expression, but it serves its purpose) and visualised in Figure 6;

- let the transition relation Δ of A be listed in Table 9 (Note that here the first and the last value of triples express states of an automaton A. In general, the middle value of a transition expresses accepted character for a transition, but here in this list the values have been replaced with identifiers of transitions, as the indices $[\![k_1]\!], \ldots, [\![k_{|w|}]\!]$ are already in accordance with the characters needed to be accepted by transitions. These indices already represent the suitable transitions accepting characters specified in the trace $[\![w]\!]$ (see Section 5.1.4 for more information);
- let the initial states of A be $I = \{0\}$;
- let the accepting states of A be $F = \{0, 1, ..., 7\}$;
- and let the indices $[\![k_1]\!], \ldots, [\![k_{|w|}]\!]$, based on trace $[\![w]\!]$ and transition relation Δ, be listed in Table 8.

Based on the input information, the result of Step 1 is listed in Table 10 and it is easy to verify that it meets the requirements of Steps 2, 3, 4 and 5 and we have proved that the trace (witness) is accepted by an automaton. How this proof has been implemented in the result tool of this thesis is explained in Section 5.2.4.

Table 8. Indices $[\![k_1]\!], \ldots, [\![k_{|w|}]\!]$ based on the trace $[\![w]\!]$.

0	12	46	38	34	51	22

Table 9. Δ – transitions of a deterministic finite automaton A (divided into four columns).

0	0	1	2	14	1	4	28	1	6	42	1
0	1	2	2	15	2	4	29	2	6	43	2
0	2	3	2	16	3	4	30	3	6	44	3
0	3	4	2	17	4	4	31	4	6	45	4
0	4	5	2	18	5	4	32	5	6	46	5
0	5	6	2	19	6	4	33	6	6	47	6
0	6	7	2	20	7	4	34	7	6	48	7
1	7	1	3	21	1	5	35	1	7	49	1
1	8	2	3	22	2	5	36	2	7	50	2
1	9	3	3	23	3	5	37	3	7	51	3
1	10	4	3	24	4	5	38	4	7	52	4
1	11	5	3	25	5	5	39	5	7	53	5
1	12	6	3	26	6	5	40	6	7	54	6
1	13	7	3	27	7	5	41	7	7	55	7

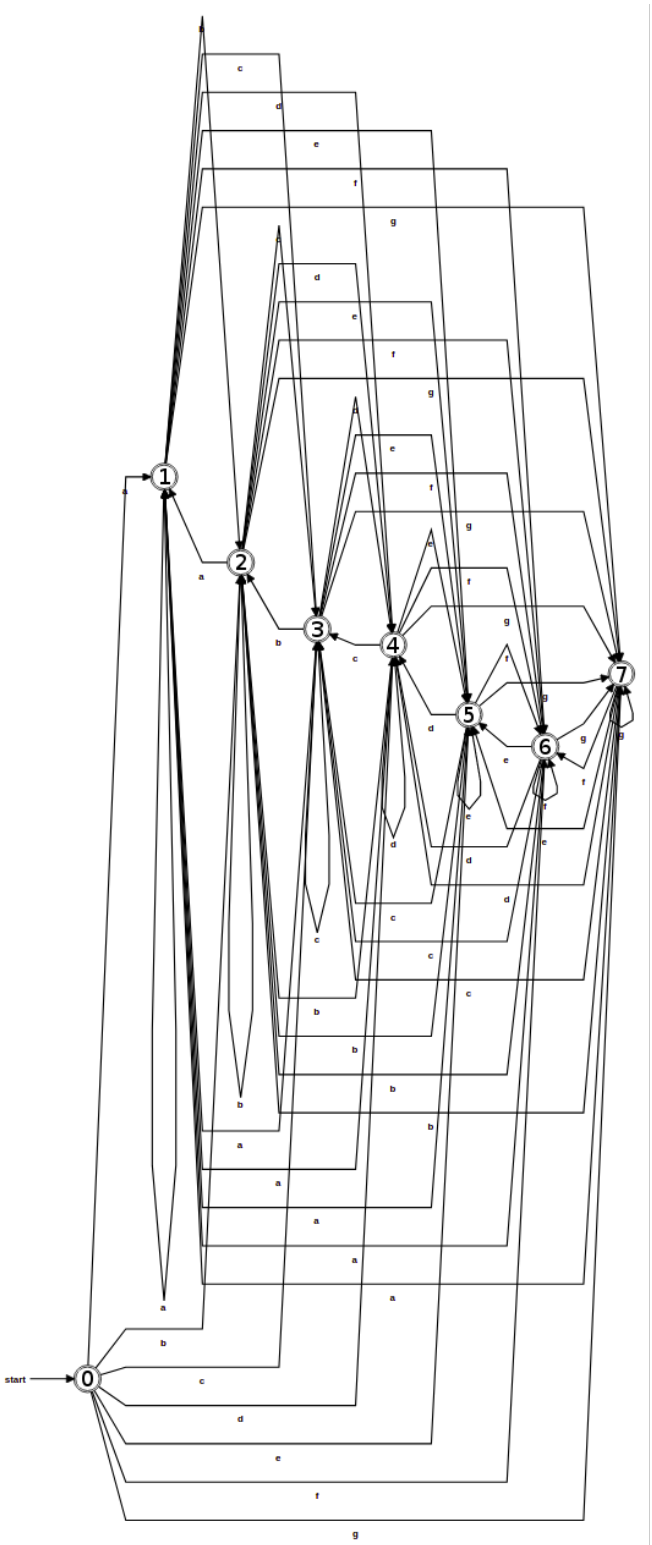

Figure 6. A deterministic finite automaton constructed from a regular expression "(a|b|c|d|e|f|g)*".

Table 10. K – elements read from transitions Δ using indices $[\![k_1]\!], \ldots, [\![k_{|w|}]\!]$ from the trace $[\![w]\!]$.

0	0	1
1	12	6
6	46	5
5	38	4
4	34	7
7	51	3
3	22	2

This section has explained the logic behind the proposed implementation tool of this thesis and the two principal proofs under it. The following section describes how this logic and proofs have been implemented in practice.

5 Implementation

This section describes how the main idea behind this thesis is brought into practice. It provides details about how the implementation tool of this thesis is built up – which are the components and technologies in use, including why and how are these used.

The overall structure of the tool implemented in this thesis is illustrated in Figure 7. There are two main applications – the client application and the server application, which both contain a Node.js server that provides a basis for the inner applications and allows the two main applications to communicate.

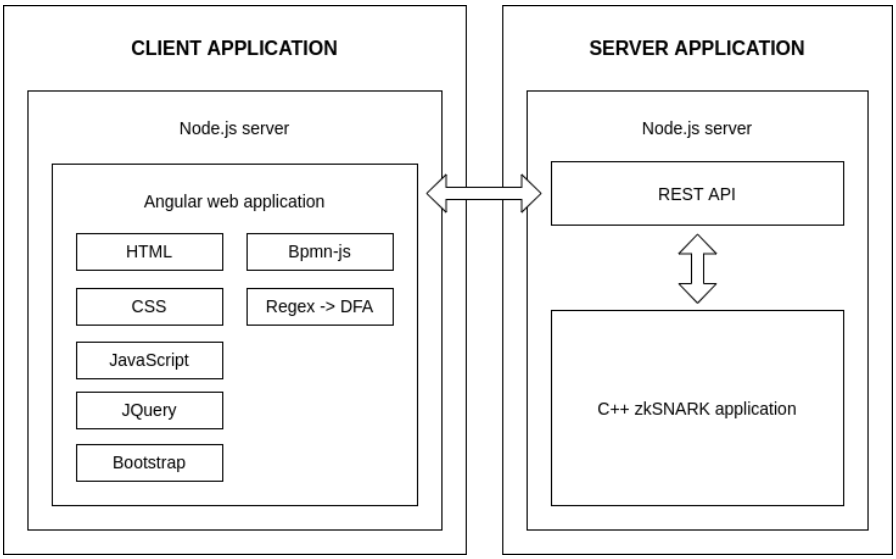

Figure 7. The structure of the implementation tool of this thesis.

The client application serves a graphical user-interface that makes the whole tool easier to use. The server application provides a gateway between client and server applications and contains the generator-prover-verifier zkSNARK application that allows to prepare, prove and verify the two principal proofs (see Section 4) under the tool implemented in this thesis.

The source code of both applications is available as an attachment to this thesis document, see Appendix I for details.

5.1 Client application

Following subsections and paragraphs describe the user-facing web application of the tool implemented in this thesis. The client (web) application provides a graphical user interface for easier use. It is described how to use this interface, what are the possibilities, how and why the input information is processed, and what the user cannot see – what is happening in the background.

5.1.1 Technologies used

The client application (user interface) of the tool implemented in this thesis is an Angular[12] web-application built on a standard @angular-cli (version 7.1.4)[13] project. Graphical user-interface is based on HTML[14] and JavaScript[15] using Bootstrap (version 3.4.1)[16] open-source toolkit and JQuery (version 3.5.1)[17] JavaScript library in addition. Bootstrap provides styled responsive templates for forms, buttons etc. JQuery supports making forms interactive.

The visualization of XML[18]-format BPMN models is served by an open-source BPMN 2.0 rendering toolkit Bpmn-js (version 6.4.1)[19] provided by Camunda Services GmbH[20].

A regular expression input parsing is done using few functions from the CyberZHG's toolbox[21] (direct functions and sources of these are referenced in the source code (see Appendix I) of the client application). These functions provide a deterministic finite automaton.

The front-end Angular application is served to the user using JavaScript runtime Node.js[22].

[12] `https://angular.io/`
[13] `https://cli.angular.io/`
[14] `https://www.w3.org/html/`
[15] `https://www.javascript.com/`
[16] `https://getbootstrap.com/docs/3.4/`
[17] `https://jquery.com/`
[18] `https://www.w3.org/XML/`
[19] `https://bpmn.io/toolkit/bpmn-js/`
[20] `https://camunda.com/`
[21] `https://github.com/CyberZHG/toolbox`
[22] `https://nodejs.org/en/`

5.1.2 Reasons for using selected technologies

For the technologies to be used in the client application, different candidates were considered. The reasoning behind why these technologies were chosen is following:

Angular, HTML, CSS, JavaScript, JQuery, Bootstrap and Node.js

These technologies were selected because of two main reasons:

1. The author of this thesis is already familiar with these technologies and based on the previous experience, decision was made that these are suitable;
2. It is possible that the application tool of this thesis will be integrated into the PLEAK toolset. PLEAK uses the same technologies, so the integration would be smoother than with other possible technologies.

Bpmn-js

There are four reasons why the Bpmn-js was chosen:

1. Bpmn-js is a well maintained and regularly improved open-source tool with great community behind it;
2. It is easy to use it in Angular projects;
3. The author of this thesis has multiple years of experience in using it, so it is familiar;
4. There are no good alternatives with comparable functionality.

CyberZHG's toolbox

The first condition for choosing a tool that would parse regular expressions and construct automatons based on it was that it must run on JavaScript. The hope behind that condition was that it would require less work in the server application. Unfortunately, it turned out that all found suitable candidates have their own problems.

The first choice was Automata.js[23] and it got integrated into the tool as it seemed to serve its purpose. It worked well for the small example model (see Figure 3), but when the medium-sized model (see Figure 16) was used, the constructed automaton had flaws in it – there were states missing. For example, if a regular expression was

[23] `https://github.com/hokein/automata.js/`

in a form (a|b|c|d|...)* then there was always missing a state accepting the character on index 3 (d). Also in order to construct an automaton for the medium-sized and big example models (see Figure 17), a large amount of memory was required – it froze the web browser for a while. Due to these issues, another tool had to be found.

The second choice (that is currently in use) was to use some functions from CyberZHG's toolbox ,as the full toolbox was not needed and it would have been not too easy to integrate it into the implementation tool of this thesis in its original form.

These functions serve their purpose very well – automatons from acceptable regular expressions are built quickly and there are no apparent mistakes in these. Unfortunately, the regular expression parser reads characters one by one so it is possible to use only one-character short names to connect the regular expression with a trace and a business process model (see Section 5.1.4 for more details). Fortunately, UTF-8 is supported, so there is a long list of suitable characters for short names.

Hoping to keep the server application simpler, a wrong choice was made, as it turned out that there are no good options for JavaScript regular expression parsers known that would construct suitable automatons from these. The rest of the chosen technologies serve their purpose well.

5.1.3 User roles

The overall purpose of the tool implemented in this thesis is to prove that it is possible to follow a certain described path under specific limitations (restrictions) in a business process. As this proof must be also verifiable, the user interface provides a choice of three roles to generate the keys for the proving process, run the proving process and later verify the proof. These roles are named generator, prover and verifier.

In general, the generator prepares proving and verification processes and creates proving and verification keys for prover and verifier accordingly (keys can be downloaded after successful generator process). The prover runs the proving process (using proving key from generator) and provides a proof if it was successful (proof file can be downloaded after successful proving process). The verifier verifies the proof (from prover) using two different verification keys (from generator).

Although all three roles are related to the same proving-verification process, the requirements for input information are different for each role, dependent on the amount of information available. General overview of the required input information is provided in Section 5.1.4, detailed lists of required information for each role is

provided in Section 5.1.5.

5.1.4 Overview of the information gathered and processed

The client side part of the tool implemented in this thesis plays mainly a supportive, user input formatting role, in addition to providing graphical user-interface. The input information required for the tool to serve its purpose is asked from the user in three forms:

1. XML-format BPMN (.bpmn) model[24] through open file dialog;
2. numerical and textual inputs through text field inputs;
3. key- and proof files through file upload inputs.

Details about how the front-end application handles and formats the input data (BPMN model and numerical and textual inputs) for later use in the back-end part of the application are described in following paragraphs.

Business process

After user of the tool has opened (loaded in) a BPMN (.bpmn) model, it is parsed by the Bpmn-js renderer which provides a JavaScript object that describes the full model and has functions to read different details, trigger events on the model and manipulate it. These functionalities are used to gather and prepare information that is required for the tool to serve its purpose.

The front-end application gathers information from the model about the following vertex-type (step) elements:

- Task elements,
- Start event elements,
- End event elements,
- Parallel gateways,
- Exclusive gateways.

[24]Suitable models can be created in Bpmn.io: `https://bpmn.io/` or Pleak.io: `https://pleak.io/`.

Previously listed elements are considered as steps that are taken when following different paths through the process. There are functions implemented to get JavaScript objects describing all of these elements.

In addition, information about edge-type elements is gathered – more precisely, about Sequence flow elements.

As described in Section 4.1, there are different requirements (such as a number of incoming and outgoing sequence flows for an exclusive gateway) for the elements in the business process for it to be usable in the tool. To check the details about the business process and its elements, there are following supporting functions implemented:

- A function to get a type of an element (Task, Start event, End event, Exclusive gateway, Parallel gateway, Sequence flow);
- A function to get all vertex-type elements;
- A function to get all edge-type elements (currently only sequence flows);
- A function to get all incoming connections of elements – this covers all different types of edge-type elements, including sequence flows;
- A function to get all outgoing connections of elements – this covers all different types of edge-type elements, including sequence flows;
- A function to get all vertex-type elements that are connected to the input of a given edge-type element (Sequence flow);
- A function to get all vertex-type elements that are connected to the output of a given edge-type element (Sequence flow).

In addition, there are functions to check if all model elements meet the requirements listed in Section 4.1. There are the following checks to guarantee that the input model is suitable for the tool:

1. Is there at least one start event?
2. Is there at least one end event?
3. Do start events and end events have correct number of incoming and outgoing connections?
4. Do all task elements have exactly one incoming and one outgoing connection?

5. In case exclusive gateways are used, is there exactly one incoming and one or more outgoing connections OR one or more incoming and exactly one outgoing connection for each exclusive gateway? Note that here (in the actual implementation) the requirements for exclusive gateways are stricter than in the BPMN model requirements listed in Section 4.1, because it was easier to implement it this way.

6. In case parallel gateways are used, are there exactly one incoming and two outgoing OR two incoming and one outgoing connections for each parallel gateway?

In case one or more of these checks fail, the user is alerted with related error message dialogs.

In case all checks succeed, using previously listed functions, a set of all (accepted) vertex (step) elements of the model is created. This is described as set V in Section 4.1. The set V is converted into a set P (described in Section 4.2), containing triples of pairs that describe for each vertex whether on each connected edge (by edge ID) the number of tokens is increased (denoted by a string "incr") or decreased (denoted by a string "decr"). If a vertex has altogether less than three incoming or outgoing connections, a string "nothing" is used instead of a pair to mark "empty" token changes. Also, the triple is not ordered.

For all vertices there is at least one triple describing token changes on the edges connected to it. As exclusive gateways in principle describe that (based on some condition) one or another (or another, etc.) incoming or outgoing edge is chosen, these gateways have multiple different possibilities for token changes (in case there are more than one incoming or outgoing connection) on its edges. This means that for exclusive gateways there can be multiple triples, as tokens are considered to change only on one incoming and one outgoing edge at the same time. How to fix which edges should be considered when describing a path through the process (a trace), is described later in the same section.

An simple example of a triple describing token changes on the connected edges for one vertex could be [("decr", 5), ("incr", 2) , "nothing"], where for the first pair, on edge with an ID 5, the number of tokens is decreased (by one).

To make the set P easier to use in the back-end application, it is further formatted into a set, where each element is a set of six numbers – a triple of pairs is merged into one set. String "incr" is replaced with an integer 1, string "decr" with an integer "-1" and string "nothing" with a pair of integers (0, -1). The

previously mentioned example becomes following: [-1, 5, 1, 2, 0, -1] - see Table 2 for an example of a full set P based on the example model in Figure 3.

Detailed requirements for the business process and its elements are described in the Section 4.1.

Regular expression and nondeterministic finite automaton

To set limitations, whether taking certain steps or following certain paths in the business process is allowed or disallowed, the user of the tool implemented in this thesis is required to insert a regular expression string. This string is parsed by using few functions from the CyberZHG's toolbox project. Due to the limitations of the CyberZHG's toolbox project, not all regular expressions are accepted. Currently, only regular expressions supported by the following grammar (where r, s, t are regular expressions and a is a character) are supported:

$$r ::= (s) \mid st \mid s|t \mid s* \mid s+ \mid s? \mid \varepsilon.$$

See the page[25] of the original project for more details.

The functions from CyberZHG's toolbox provide a JavaScript object describing deterministic finite automaton (DFA). This DFA contains information about the number of states, initial state, accepted states and possible transitions.

Possible transitions are formatted into a list of triplets (Δ, as described in Section 4.3), containing state ID, accepted character and following state ID. State ID-s are here automaton-specific, while accepted characters are the characters from the trace, separated by commas. Using these transitions, the tool finds out how the automaton accepts the input string (trace) and creates a new trace that contains consecutively identifiers of the transitions that accept (consecutively) each character of the trace. After that, characters in triples are replaced with transition ID-s. Finally, when the list of transitions and the new trace are sent to the server application, there are no characters in these structures any more, only integers – see Table 9 for an example of set Δ based on the example model in Figure 3.

Trace

To describe a certain path (being proved by the tool implementation of this thesis to exist) on the model, the user is required to insert a trace, a string containing short names (short name is expected to be a one-character UTF-8 identifier

[25] `https://cyberzhg.github.io/toolbox/regex2nfa`

that can be fixed in element's name between parentheses) of vertex (step) elements (separated by commas) in a specific order.

In case the trace contains an exclusive gateway element and there are more than one incoming or outgoing connections, user needs to specify which incoming or outgoing connection (edge) should be considered. This specification can be done by naming or numbering these incoming or outgoing connections (logic is the same as for naming vertex elements – the short name should be fixed between parentheses – except that it can be longer that one character) on the model and then specifying selected connection in the trace by adding the short name of the connection next to the short name of the exclusive gateway into parentheses. For example, if an exclusive gateway with a short name "x" has two outgoing connections, with short names 1 and 2, and connection 1 is chosen, it should be written as "x(1)" in the trace.

The inserted trace string is split into elements' short names and compared against the set of all model vertex element names. In case there is a short name of an edge in parentheses next to the element's name in the trace, the short name of an edge is considered when calculating token changes on certain edges. To be precise, the short names are compared to the elements' names in the set of all model vertex element names to connect these short names to real elements, but in order to consider certain paths (fixed in parentheses next to step names), correct elements from set P have to be found. The short name of the element from trace must match the name of the element in set P, but also the short name of the edge from trace must match an element that describes token changes on this edge. A new ordered set is created, where element names are replaced with corresponding ID-s of the elements from the set P (set of vectors of token changes for each edge connected to certain vertex elements of the model).

Other information

In addition to the information mentioned in previous paragraphs, to serve its purpose, the back-end part of the tool implemented in this thesis requires also the number of edge elements on the model.

In case not all of the information mentioned in previous paragraphs is available, it could be replaced by the numbers of model vertex (step) elements, model edge elements, automaton transitions and automaton transitions. Also, the length of the trace is required. These numbers can replace the information mentioned in previous paragraphs when using the generator user role, prover and verifier roles require more input information. The list of the user roles is provided in Section 5.1.3 and

lists of the required input information for each case is described in Section 5.1.5.

5.1.5 Graphical user-interface

In general, the graphical user-interface (a part of the front-end application) of the tool implemented in this thesis provides a user-friendly, easy to use way to interact with the zero-knowledge proving system, the main part of the back-end application. Communication between the front-end and back-end application is done using REST[26] API (a part of the back-end application).

The graphical user-interface can be divided into two parts – the left side for the BPMN model input and visualization, and the right side for the user role choice and role-specific inputs. User role can be selected in the upper right corner of the panel.

Instructions on how to use the graphical user-interface and required input information for each role is described in following three paragraphs.

Generator

Running the tool as a generator, there are two principal options:

1. to run the generator process knowing all information about the business process (having access to the BPMN model), knowing input regular expression and trace strings (see Figure 8). In this case, user is required to load in BPMN (.bpmn) model (using "Open model" button (see Figure 9) and insert regular expression and trace strings;

[26] `https://restfulapi.net/`

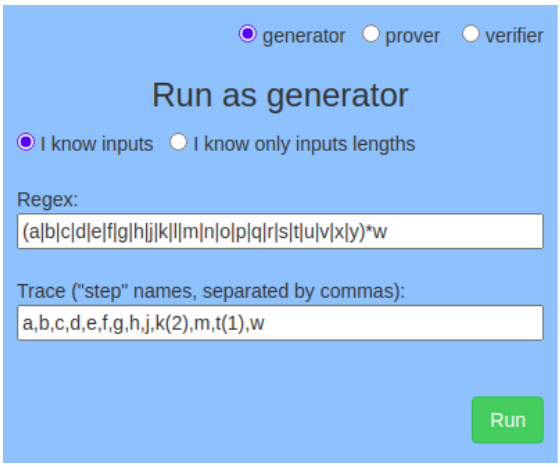

Figure 8. Inputs for the generator role, knowing all necessary information about the business process, regular expression and trace (pre-filled values are based on the medium-sized example model in Figure 16).

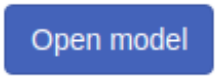

Figure 9. Active "Open model" button, to load in the BPMN (.bpmn) model.

2. to run the generator process without having access to the model, but knowing certain details about the business process and knowing details about the nondeterministic finite automaton (constructed from the regular expression input) (see Figure 10). In this case, user cannot load in a BPMN model ("Open model" button is inactive – see Figure 11), but is required to insert numerical details about the business process – the number of steps (accepted vertex elements) and the number of edges (sequence flows). In addition, information about the NFA is required – numbers of transitions and states. Also, trace length is needed. These values are calculated and provided to the user when running the application in prover role.

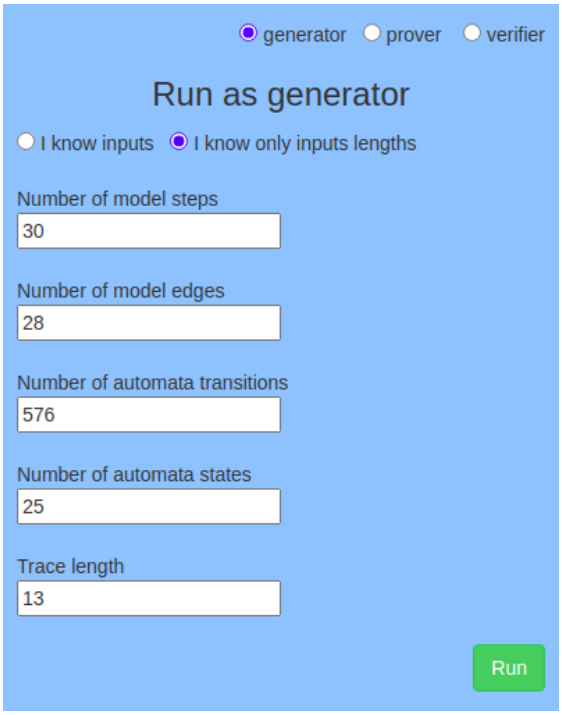

Figure 10. Inputs for the generator role, not knowing all details about the business process, but knowing some numerical values about the business process, DFA and trace.

Figure 11. Inactive "Open model" button – in case running as generator and not having access to the BPMN (.bpmn) model.

After inserting all required details and clicking "Run" button, there are two possible outcomes:

- In case the generator process fails, a red "Generator process unsuccessful!" message appears;
- In case the generator process is successful, a green "Generator process successful!" message is shown and it is possible to download keys required for proving and verification processes (see Figure 12).

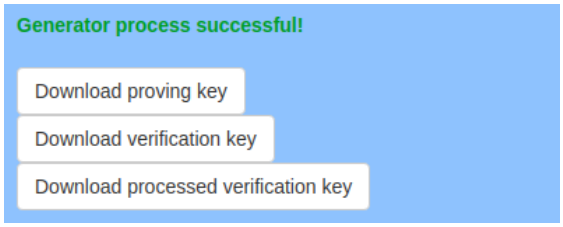

Figure 12. Generator process has been finished successfully, keys for proving and verification processes can be downloaded.

After downloading these three keys, it is possible to continue with the proving process.

Prover

In order to run the tool as a prover, the user needs to have access to the same amount of information as a generator (in case having access to the BPMN (.bpmn) model). The user is required to load in BPMN (.bpmn) model (using "Open model" button (see Figure 9) and insert regular expression and trace strings. In addition, proving key provided by generator is required. All required inputs can be seen in Figure 13.

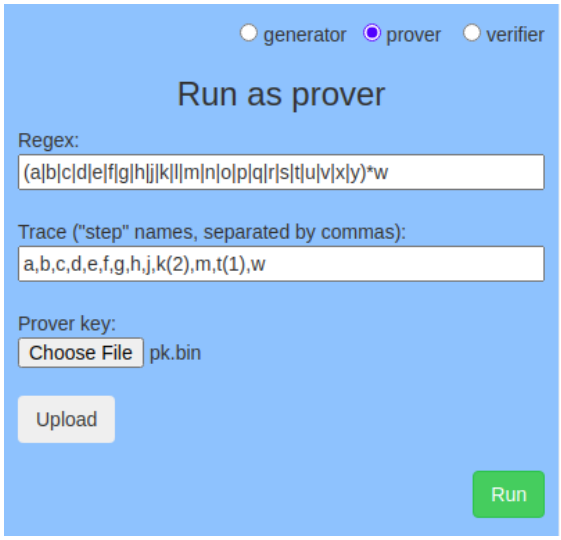

Figure 13. Inputs for the prover role.

Before clicking the "Run" button and starting the proving process, proving key needs to be uploaded to the server – this means that the user needs to select the proving key (*pk.bin*) file from his / her computer and click "Upload" button. There are two possible outcomes after clicking the upload button:

- In case the upload process fails, a red message "Upload failed!" appears;
- In case the upload process is successful, a green message "Prover key uploaded!" with a remark "Click 'Run' to run proving process" is shown.

In addition to two possible upload-related messages, information required (see Figure 10) in the generator process (when running the process without using BPMN (.bpmn) model as an input) is provided. This contains the numbers of model steps, model edges, automaton (DFA) transitions and automaton states, also, length of the trace.

In case the proving process fails, a red "Proving unsuccessful!" message appears. In case it is successful, a green "Proving successful!" message is shown and it is possible to download the proof file (*proof.bin*). This proof can be later verified in the verification process.

Verifier

To use the tool as a verifier (to verify a proof), the user is required to load in a BPMN (.bpmn) model (using "Open model" button (see Figure 9) and insert a regular expression. Instead of a trace, the length of the trace string is required. In addition, the proof provided by prover and two verification keys – verification key (*vk.bin*) and processed verification key (*pvk.bin*) – are required. All required inputs can be seen in Figure 14.

Figure 14. Inputs for the verifier role.

Before starting the verification process, the proof and verification keys have to be uploaded to the server. There are two possible outcomes after selecting required files and clicking the upload button:

- In case the upload process fails, a red message "Upload failed!" appears;
- In case the upload process is successful, a green message "Proof, verification key and processed verification key uploaded!" with a remark "Click 'Run' to run verification process" is shown.

In case the verification process fails, a red "Verification unsuccessful!" message appears. In case it is successful, a green "Verification successful!" message is shown and the proof about initial statements has been verified.

5.2 Server application

Following subsections and paragraphs describe the part where the actual goal of this thesis is achieved – the generator-prover-verifier zkSNARK application that allows to prove whether it is possible to take a certain path in a business process under stated conditions. It is explained what technologies are used and why, how the application is built up and how the proving process takes place in practice.

5.2.1 Technologies used

The server application of the tool implemented in this thesis consists of two main components – a Node.js server and a generator-prover-verifier C++ application.

In general, the Node.js server provides a REST API for the client application to communicate with the generator-prover-verifier application. In addition, it provides functions for preparing compilation files and API endpoints for file upload, cleaning directories, preparing compiling and executing generator-prover-verifier application.

The generator-prover-verifier application is a set of zkSNARK applications compiled from one C++ source code, but using different inputs for generator, prover and verifier roles. The application uses a slightly modified version of libsnark-tutorial[27] project as a basis that provides a CMake[28] compilation framework. The libsnark-tutorial project contains libsnark library that "implements zkSNARK schemes, which are a cryptographic method for proving / verifying, in zero knowledge, the integrity of computations" [libb]. Libsnark provides a Groth16 [Gro16] zkSNARK protocol that is comprised of setup (generator), proving and verification phases.

5.2.2 Reasons for using selected technologies

Compared to the client application, the conditions for the technologies to be used in the server application were slightly different. The reasoning behind why these technologies were chosen is the following:

[27] `https://github.com/howardwu/libsnark-tutorial`
[28] `https://cmake.org/`

Node.js

As the developing of the client application of the application tool of this thesis began first, it was a logical choice to use the same server technology in the server application as well. Also, the author of this thesis hoped to get more experience in using Node.js in this way.

Libsnark and C++

The libsnark library was chosen because it is the most mature and widely known tool for building zero-knowledge proof applications. Also, DIZK[29], a Java library for distributed zero knowledge proof systems, was considered, but libsnark seemed to be more suitable for our needs. As the libsnark is a C++ library, it was a logical choice to build the server application using the libsnark also in C++.

Libsnark-tutorial project was selected as a basis for the generator-prover-verifier application as it already contained tools and files for building libsnark example applications – this made taking libsnark into use a lot easier.

All technologies selected for the server application seem to serve their purpose well as there have been no unsolvable problems so far.

5.2.3 Node.js server

The Node.js server part of the server application provides a gateway between the client application and the generator-prover-verifier application that is running in the background of the server application. It makes it possible for these two applications to communicate. Following paragraphs describe the REST API that is used for the communication, and generator-prover-verifier application that does the heavy part – preparing proving and verification keys, running the proving and verifying processes, that are the vital parts of achieving the goal of this thesis.

Compilation of the generator-prover-verifier application

The Node.js server contains three functions to prepare the compilation of generator-prover-verifier application based on the inputs from the client application:

- Firstly, there is a function to overwrite the default CMakeLists.txt file (in *./libsnark-tutorial/src* directory) of libsnark-tutorial project. The libsnark-

[29]https://github.com/scipr-lab/dizk

tutorial project creates one executable for all three (generator, prover, verification) phases described in example application's program file. In this thesis, running these three phases (roles) is divided into three executables – one for each role. This function is used, so there is no need to manually change the CMakeLists.txt file in the libsnark-tutorial project submodule;

- Secondly, there is a function to write the input data from client application into the source file of the generator-prover-verifier application. Each time a generator, proving or verification process is started, this function is used to make sure that the application uses the latest input data;
- Thirdly, there is a function to compile (using the latest input data from client server) and execute generator-prover-verifier application. This function is used each time a generator, proving or verification process is started.

These three functions are called through POST requests to the REST API endpoints – these endpoints are listed in Section 5.2.3).

REST API

The Node.js server in the server application provides following endpoints to make generator-prover-verifier application usable for the user:

#POST /cleanUploadDirectory

Removes all files in the upload directory (*./libsnark-tutorial/upload*), where the user of the client application has uploaded key and proof files when running the application as a prover or a verifier. This endpoint is used each time before uploading new files to the server.

#POST /cleanDownloadDirectory

Removes all files in the download directory (*./libsnark-tutorial/download*), where all the keys and proof files generated by generator and proving processes are stored and can be downloaded by the user of the application.

#POST /proverFileUpload

Uploads a proving key file used in prover process into *.libsnark-tutorial/upload* directory and renames it to *pk.bin*.

#POST /verifierProofFileUpload

Uploads a proof file used in verification process into *.libsnark-tutorial/upload* directory and renames it to *proof.bin*.

#POST /verifierVKFileUpload

Uploads a verification key file used in verification process into *.libsnark-tutorial/upload* directory and renames it to *vk.bin*.

#POST /verifierPVKFileUpload

Uploads a processed verification key file used in verification process into *.libsnark-tutorial/upload* directory and renames it to *pvk.bin*.

#POST /exec

Prepares compilation files for the generator-prover-verifier application, writes input data from the client application into the application's source file (*./template.cpp*), compiles and executes one of the three executables (generator, prover or verifier in *./libsnark-tutorial/build/src* directory), based on the user role provided as one input from the client server.

5.2.4 zkSNARK (generator-prover-verifier) application

The generator-prover-verifier application in the tool implemented in this thesis is a zkSNARK application that uses Groth16 [Gro16] zkSNARK protocol provided by libsnark. The protocol includes three phases – generator (setup), prover and verification. In setup phase, public parameters (used by prover and verifier) are constructed. In the proving phase, using public parameters and public and private inputs, succinct proof is generated. In verification phase, the proof is verified, using verification key, public input and the proof.

In more detail, used scheme is a preprocessing zkSNARK (ppzkSNARK)[30]. As explained in the description of the libsnark library [libb], this means that first the size / circuit / system representing proved NP statements are decided and then a generator algorithm is used to create corresponding public parameters. The public parameters include the (long) proving key, the (short) verification key and the (short) processed verification key. After that, the prover uses the proving key and

[30] `https://github.com/scipr-lab/libsnark/tree/master/libsnark/zk_proof_systems/ppzksnark`

input data to generate a proof, which, together with verification keys, is used by the verifier to verify the statements of the proof.

In this application, the ppzkSNARK uses bn128 elliptic curves – this is used by default in libsnark-tutorial project. As stated in libff library [liba], bn128 is "an instantiation based on a Barreto-Naehrig curve, providing 128 bits of security. The underlying curve implementation is [ate-pairing], which has incorporated our patch that changes the BN curve to one suitable for SNARK applications".

After generator, libsnark's prover algorithm creates proofs of true statements about the satisfiability of the R1CS and libsnark's verifier algorithm checks proofs for declared statements.

In the zkSNARK application of this thesis, different statements to be proved as R1CS are constructed and linked together with libsnark in multiple steps of two proof constructions (see Section 5.2.4). Currently, these two constructions are not used as two separate proofs to be proved by libsnark's prover algorithm, but instead, are considered as parts of one aggregated proof. The primary input for this aggregated proof consists of specific inputs of both smaller proofs.

Different statements to be proved as R1CS are constructed using libsnark's protoboard, which stores information and allows to add constraints to this information – through gadgets (in this case, from libsnark's gadgetlib2, a library to construct R1CS instances) and directly, using specific functions.

The first information stored into the protoboard is considered by libsnark as the primary input for the proof – the amount is fixed by setting value to the $primary\_input\_size$ property of the R1CS constraint system of the protoboard. This information is available for both – prover and verifier. The rest of the stored information is considered as an auxiliary input – this is available only for the prover. Generator knows dimensions of input vectors and matrices, but not the contents of these.

Following are provided details about the implementation of two proofs in this generator-prover-verifier application.

A trace (witness) belongs to the business process semantics

The primary input (available for prover and verifier) for this proof (based on the Section 4.2) is a set of vectors of token changes (set P) for each edge connected to certain vertex elements of the model. Elements of the set P are vectors of six integers (see Section 5.1.4 for details). The private input (available only for the prover) is the trace (*trace*, also a vector of integers) and public input is the number of edges ($numberOfEdges$) in the model.

As stated in Section 4.2 (step 1), the first step of this proof is reading values from P using values from vector *trace*. This is done in function $ParallelRead$ using gadget $LooseMUX\_Gadget$ (from gadgetlib2). The function is divided into three parts:

1. As the gadget $LooseMUX\_Gadget$ does not allow to read vectors from a set, but requires numerical values to be read, the set P is transposed from rows into columns. This is done by generator, prover and verifier;

2. To read values from *trace* and from each column of P, there is a separate gadget used. This means that six gadgets are used for each row. Here, prover generates witness for each gadget and generator, prover and verifier all add constraints to state that all these reading processes must be successful;

3. The result of the previous step is transposed back into rows. Specific values here are known (and returned by the function) only for the prover. For generator and verifier, the values are all zeros.

An example of an output of function $ParallelRead$ based on the example model in Figure 3 is listed in Table 3.

The second step (see Section 4.2 (step 2)) is covered by a function $CreateL$. Here integer 999999999 is used instead of string "init". All value assignments from the result of the previous function into the result of this function (set L) are done only by the prover. Fixing constraints, stating which values from the input set must be equal to which values in the output set of this function, are done by generator, prover and verifier.

An example of an output of function $CreateL$ is listed in Table 4.

The third step of the proof requires sorting the set L. This is covered by a function *Sort*, which is divided into four parts:

1. The set L (set of triples) is sorted – vector R (vector of triples) is produced. The sorting based on conditions stated in Section 4.2 (step 3)) is done only by the prover. For generator and verifier, the dimensions of the vector R are known, but all values in each triple are zeros;

2. A a permutation matrix is created while sorting L. After that, a function $AssertPermutation$ is used to check if the permutation matrix is correct. All values of the matrix are enforced to be boolean and constraints are added to state that sums of the values of each column and each row must be one – each row and each column of the matrix contains exactly one value of one, the rest are zeros. This is done by generator, prover and verifier;

3. The set L (set of triples) and vector R (vector of triples) are both divided into three columns. A function $MatVecMult$ is used on each column of L to make sure that the result of multiplying the column with permutation matrix is the same as each corresponding column of vector R. Function $MatVecMult$ uses gadget $InnerProduct\_Gadget$ for this purpose. Generator, prover and verifier generate constraints for this gadget, but only prover generates a witness;

4. A function $AssertSorted$ is used to check if L is sorted correctly. In this function, thirteen gadgets are used to check if all sorting requirements are met. These gadgets are based on the graph in Figure 15, illustrating the thirteen condition and their combination nodes (derived from the sorting conditions in Section 4.2 (step 3)) and how they are related. These gadgets include $Comparison\_Gadget$, $EqualsConst\_Gadget$, AND_Gadget and OR_Gadget. Generator, prover and verifier all generate constraints for these gadgets, but only prover generates witnesses.

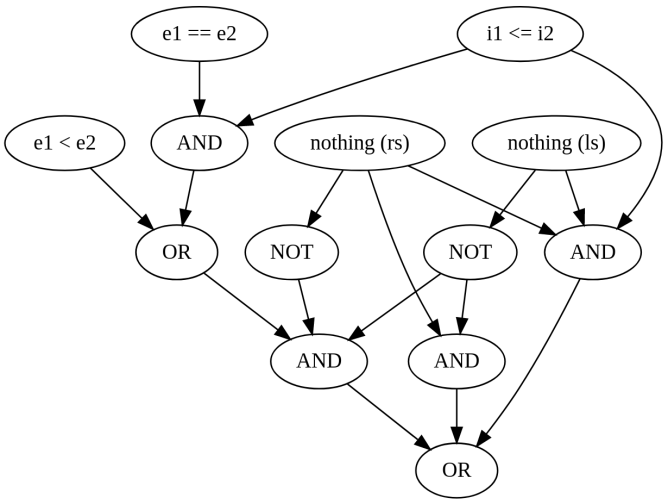

Figure 15. Sorting conditions used in the proof "A trace (witness) belongs to the business process semantics".

An example of an output of function *Sort* is listed in Table 5.

The fourth step of the proof (see Section 4.2 (step 4)) is covered by a function *RewriteL*. Everything in this function is done by generator, prover and verifier.

An example of an output of function *RewriteL* is listed in Table 6.

The fifth step of the proof contains finding prefixsum of the result set of the previous function and verifying that there are no negative values in the result. This is covered by the function $PrefixSum$. This function is divided into three parts:

1. The first value of the result set is assigned only by the prover. Constraint to state that the first value of the result set must be equal to the first value of the result set of the previous function is added by generator, prover and verifier;

2. Based on conditions in Section 4.2 (step 5), prover calculates prefixsums. Also, prover checks if any of the calculated values is negative;

3. Two *EqualsConst_Gadget*-s and one *OR_Gadget* are used to compare values in order to calculate prefixsums, following the same conditions as used in previous step. Constraints for these gadgets are generated by generator, prover and verifier, only prover generates witnesses. In addition, a rank 1 constraint is added (by generator, prover and verifier) to state, which value

must be used when calculating the prefix sum (in case one of the values is 999999999).

An example of an output of function $PrefixSum$ is listed in Table 7.

In case running previously listed steps raises no errors, program continues with the proof described in the next section.

A trace (witness) is accepted by an automaton

The primary input (available for prover and verifier) for this proof is a set of transitions (*transitions*) and a vector of accept states (*acceptStates*) of the DFA constructed from the regular expression input. The private input (available only for the prover) is the trace (*trace*) and public input is the initial state of the previously mentioned DFA.

As stated in Section 4.3 (step 1), the first step of this proof is reading values from set *transitions* using values from vector *trace*. This is done in function $ParallelRead2$ in the same way as described in the first step of the proof in Section 5.2.4 – the only difference is that instead of six gadgets, three are used (see Section 5.1.4 for details about the dimensions and contents of set *transitions*).

An example of an output of function $ParallelRead2$ is listed in Table 10.

The second, third, fourth and fifth steps of the proof are covered by function $CheckK$. This function is divided into three parts:

1. $EqualsConst\_Gadget$ gadget is used to check if the first state (the first value of the first triple of the result vector) of the result vector of the previous function (K) equals to the initial state of the DFA. Constraint is generated for this gadget by generator, prover and verifier. Witness is generated only by the prover;

2. $LooseMUX\_Gadget$ and $EqualsConst\_Gadget$ gadgets are used to check if the final state (the last value of the last triple of vector K) of vector K is in the vector of accept states (*acceptStates*) of the DFA. *acceptStates* is a boolean vector containing ones and zeros. The length of the vector is the number of states in DFA. Ones in the vector denote whether a state in this position is an accept state. Checking if the the last value of the last triple of vector K is in accept states means comparing whether the value in accept states in the position of the value of the last state (last value of the last triple)

is one. Constraints for both gadgets are generated by generator, prover and verifier, witness is generated only by the prover;

3. *EqualsConst_Gadget* is used to check if the first value minus second value equals to zero – these cover the checks required in Section 4.3 (steps 4 and 5). Here, constraints are generated by generator, prover and verifier, but witness is generated only by the prover.

In case running previously listed steps raises no errors, program continues with libsnark's generator, proving or verifier algorithms, based on the user role being used.

6 Results

The main result of this thesis is the implemented generator-prover-verifier zkSNARK application and the proof of the concept that it is possible to implement the previously stated idea – to combine business processes with zero-knowledge proofs and through that receive new value. As Sections 4 and 5 already describe the design principles and implementation of the tool thoroughly, this section describes the three example models that were designed together with the tool and also benchmarking results of the tool using these three models. These models and benchmarking results give a little overview of what this tool is capable of.

6.1 Example models

Together with the implementation tool of this thesis, three models were designed (see Appendix I). Firstly, these models were needed to test if the tool implemented in this thesis works at all. Secondly, these models are provided in order to show how the tool is able to handle models with different sizes.

The first (smallest) model

The first model, used as an example in Section 4.1 existed already before, it just happens that it meets all the requirements for an input BPMN model of this tool. In terms of this thesis, the meaning of the contents of the model is not relevant.

The input regular expression used with the model is following:

*(a|b|c|d|e|f|g)**

As this model is used only as a simple example to explain step-by-step how the proofs are built up, there are no limitations added, all steps are allowed.

The trace used with the model is following:

a,f,e,d,g,c,b

The second (medium-sized) model

The second example model (see Figure 16) was specially designed for this tool by Pille Pullonen. Firstly, it has been designed to illustrate a close to real-life process in which it is possible to show a flaw. Secondly, it has been designed to be big enough so processing it would require work that is already not that trivial or

comfortable to do on paper, at least compared to the smaller model.

In terms of this thesis, this model has already a meaning. The model describes a process of ordering a product. It begins from the point where an order for some product comes in and is going to be processed, it ends with cancelling the order or sending out the product. In between, there are different activities, including a step where it is being checked whether the payment has cleared. Based on this, we want to show that it is possible to achieve, that the product is being sent out without actually paying for it. For this, we set a constraint that all steps are allowed to be taken, except the one requiring to wait until the payment is done ("Wait until the payment is done (i)") and the final (goal) step is the "Send out the product (w)" step. The regular expression is the following:

*(a/b/c/d/e/f/g/h/j/k/l/m/n/o/p/q/r/s/t/u/v/x/y)*w*

A trace describing such a path in the model where paying is not required, but the product is sent out, is following:

a,b,c,d,e,f,g,h,j,k(2),m,t(1),w

In this trace, we have fixed certain paths to be taken in decision points (exclusive gateways). The names (numbers) of these paths are marked in parentheses following the short name (one-character identifier between parentheses) of a step. It can be seen in Figure 16 that, for example, after step with the short name "k", we have chosen to continue to the step with the short name "m" – here the number 2 in parentheses after "k" lets the application know that on step "k", token changes should be considered for the incoming edge and the outgoing edge with the short name "2".

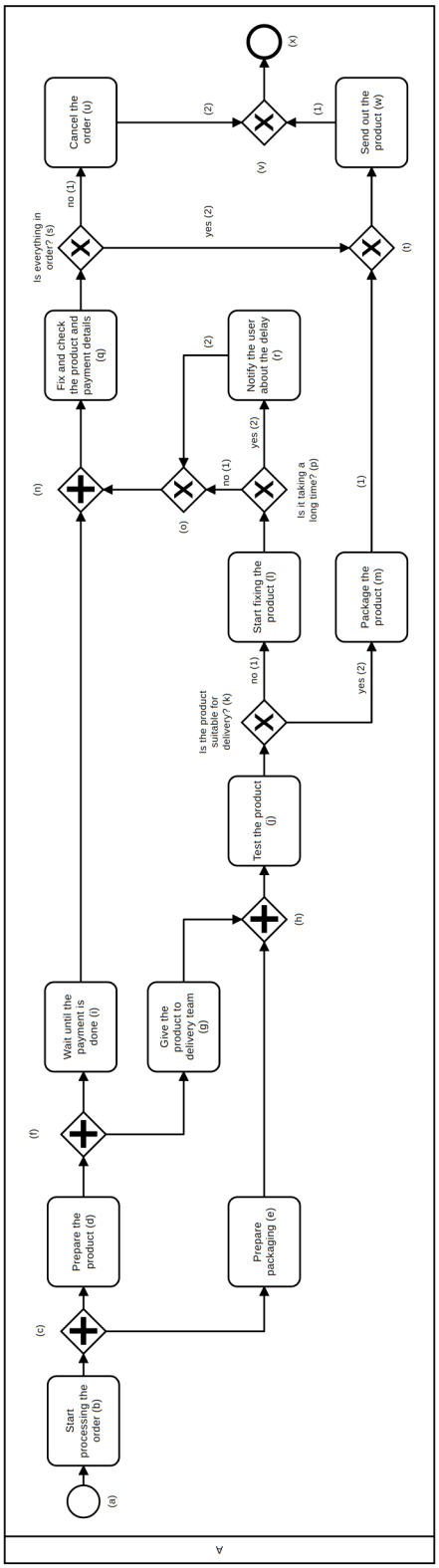

Figure 16. Medium-sized example model.

The third (large) model

The third example model (see Figure 17) was also specially designed for this tool, by Peeter Laud, – it is based on the model[31] by Sara Belluccini [BND+20]. This example model has been designed to test if the tool can handle that size of a model and the set of transitions of the automaton constructed from the regular expression setting limitations to this model. In terms of this thesis, the contents of this model is also not too relevant, we just want to show that it is possible to take two bad steps (reach to points named "First bad place to reach (1)" and "Second bad place to reach (I)") on the model.

The input regular expression used with the model is following (split onto two lines):

*(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|õ|ä|ö|ü|A|B|C|D|E|F|G|H|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|Õ|Ä|Ö|Ü|0|2|3|4|5|6|7|8|9|@|€|ç|©|®)*1I*

The trace used is following:

a,c,b,d,e,g,h,i,m,r,z,s(2),ä,C,k,o,v,ü(2),l,q,x,A,E,H,J,N,R,W(2),y,S,Y,3,p,w,1,I

Similarly to the the trace of the medium-sized example model, we have fixed here paths to be taken when going through exclusive gateways (namely the exclusive gateways with short names "s", "ü" and "W").

[31] `https://pleak.io/pe-bpmn-editor/viewer/XSKNlq2-dwJV_jQAem2r/`

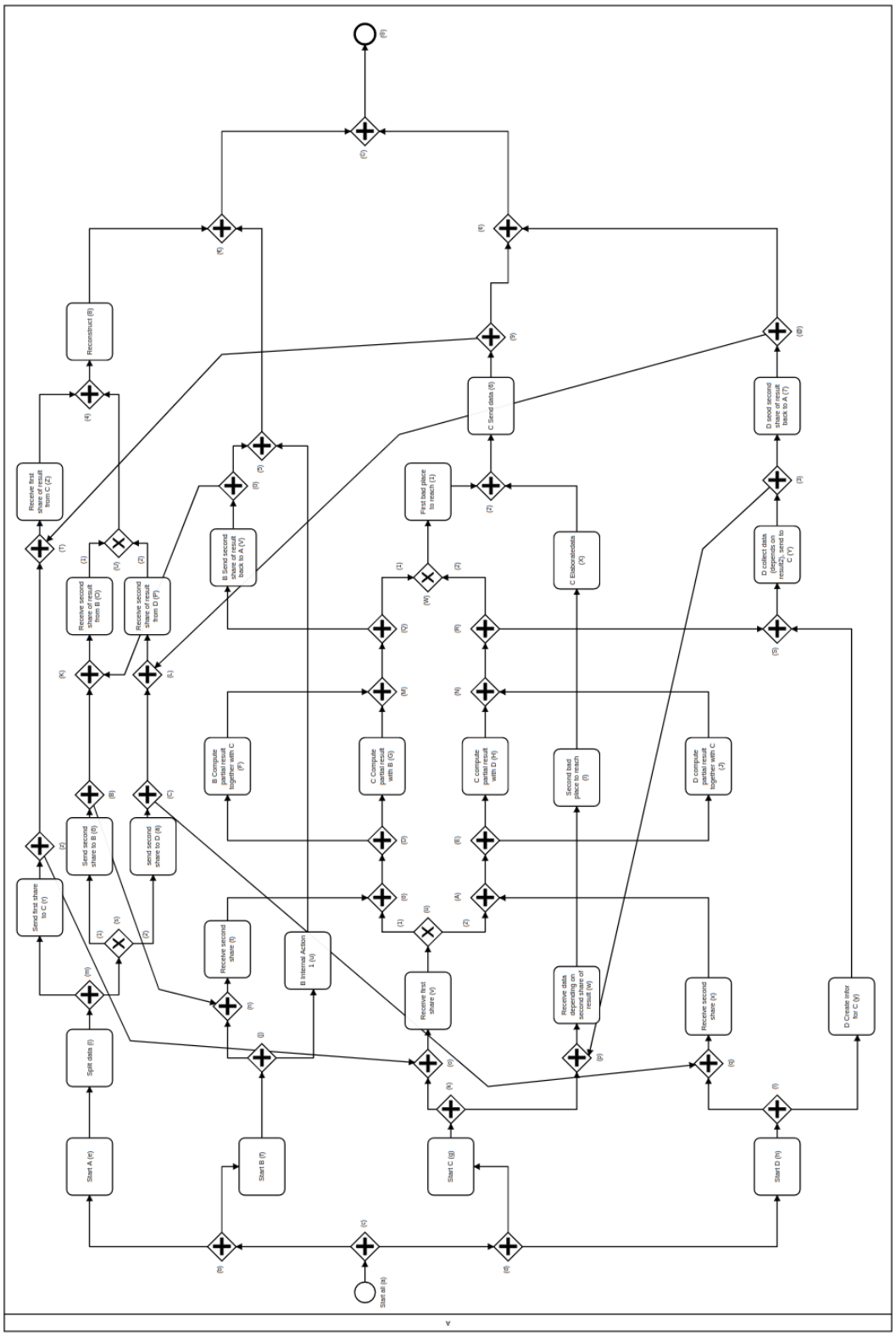

Figure 17. Big example model.

For all three models, using described input information, it is possible to show that the trace belongs to the semantics of the business process and that trace is accepted by the automaton constructed from the regular expression – meaning that the path described by the trace is a part of the business process and it has no conflict with the limitations stated by the automaton. With these inputs, all three steps – generator process, proving process and verification process – are successful.

6.2 Benchmarking

In order to provide information whether the execution times of different parts of the tool implemented in this thesis are reasonable and whether the execution times are consistent for multiple consecutive executions, here are benchmarking results for six different activities – three in the client application and three in the server application. In Table 11 some details about the input models and the automatons used with the models are provided. Also, Table 12 lists information about the keys used in benchmarking activities.

Table 11. Details about the example models and automatons used with the models.

Model	**Vertices**	**Edges**	**Elements in P**	**Length of trace**	**States in A**	**Transitions in A**
Small	7	7	7	7	8	56
Medium	24	28	30	13	25	576
Big	71	90	75	36	76	5477

Table 12. Key sizes for all three example models (PK – proving key, VK – verification key, PVK – processed verification key).

Model	**PK**	**VK**	**PVK**
Small	2.7 MB	8.8 kB	110 kB
Medium	11.5 MB	76.7 kB	177,9 kB
Big	105.4 MB	684.6 kB	785.8 kB

The size of proof files for all three models is 312 bytes.

For benchmarking, execution times were measured ten times for six different activities – the activities were the following:

In client application:

- A = preparing all inputs for the generator process to be sent to the server application;
- B = parsing regular expression, constructing an automaton and preparing all inputs for the proving process to be sent to the server application;
- C = parsing regular expression, constructing an automaton and preparing all inputs for the verification process to be sent to the server application.

In server application:

- D = compiling the executable and running generator process;
- E = compiling the executable and running proving process;
- F = compiling the executable and running verification process.

Table 13. Measurements of activities A, B, C, D, E and F using the small example model (milliseconds).

#	**A**	**B**	**C**	**D**	**E**	**F**
1	1	27	3	9680	8324	5862
2	0	33	3	9535	8372	5787
3	0	52	3	9536	8349	5780
4	1	21	3	9528	8372	5698
5	1	31	3	9450	8245	5723
6	1	36	3	9427	8305	5730
7	1	31	4	9652	8325	5876
8	0	40	4	9512	8367	5805
9	1	25	2	9546	8403	5740
10	1	38	3	9652	8392	5719
Avg.	1	33	3	9552	8345	5772

Table 14. Measurements of activities A, B, C, D, E and F using the medium-sized example model (milliseconds).

#	**A**	**B**	**C**	**D**	**E**	**F**
1	4	72	32	14110	11218	6621
2	3	69	34	14130	11261	6744
3	3	53	38	14060	11169	6723
4	3	84	36	14246	11179	6755
5	4	51	35	14166	11225	6712
6	3	56	35	14200	11203	6727
7	2	64	32	14061	11195	6839
8	3	72	32	14056	11206	6773
9	4	51	33	14194	11184	6740
10	3	64	32	13991	11188	6728
Avg.	3	64	34	14121	11203	6736

Table 15. Measurements of activities A, B, C, D, E and F using the big example model (milliseconds).

#	**A**	**B**	**C**	**D**	**E**	**F**
1	15	1074	606	66471	51705	31757
2	16	1164	599	66160	51771	31957
3	13	1058	592	66605	51645	31593
4	14	1015	604	66321	51508	31833
5	19	1019	612	66415	51804	31547
6	15	1006	602	66274	51493	31891
7	17	1112	608	66079	51263	32013
8	13	1064	603	65811	51688	31985
9	15	1017	596	66445	52019	31407
10	14	1057	598	66437	51977	31911
Avg.	15	1059	602	66302	51687	31789

To summarise the measurement results in Tables 13, 14 and 15, the average execution times for activities A, B and C for all three example models are illustrated in Figure 18 and for the activities D, E and F in Figure 19.

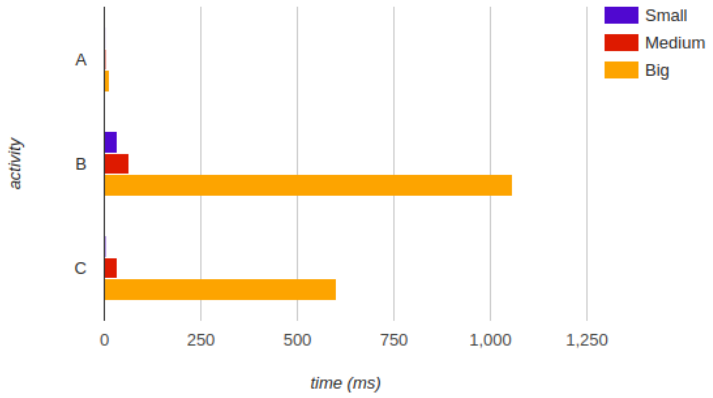

Figure 18. Average execution times for activities A, B and C.

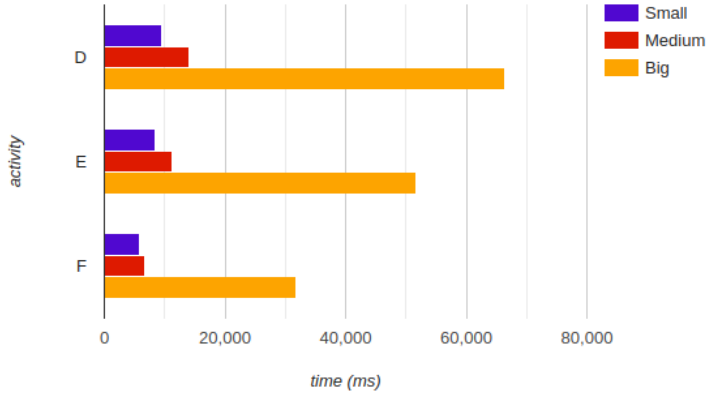

Figure 19. Average execution times for activities D, E and F.

As the client application runs in the user's web browser, it is difficult to draw any conclusions about the execution times of the activity A, B and C (current measurements of the client side application were done using a computer with 16GB of RAM and CPU Intel® Core™ i5-6300U CPU @ 2.40GHz × 4). Yet, as expected, we can see that the bigger the model, the longer are the execution times.

It is also difficult to draw conclusions whether the execution times for activities D, E and F in server application are reasonable, but as expected, we can see that similarly to the client application activities, the bigger the model, the longer the execution times. Also, we can see that verification processes take reasonably less

time than proving processes as the proving key is much longer than verification keys (see Table 12). Measurements of the activities running in the server application were done in a virtual machine on OpenNebula[32] platform with 12GB of RAM and 1 CPU with 2 VCPU-s.

[32] https://opennebula.io/

7 Discussion

This section concludes the work and describes what has been achieved with this thesis. Also, known limitations of the result tool of this thesis are listed, and for these, solutions as a future work are suggested. In addition, some new ideas of how the tool could be improved even more are provided.

7.1 Conclusion

The main goal of this thesis was to implement a prototype that would consist of two main parts – firstly, a user interface that would allow the user of the tool to load in a business process (BPMN) model and state certain information about this model, and secondly, a zkSNARK application that would, in case the statements about the business process model were true, provide a proof of these true statements and provide a way to verify the proof.

As a result of this thesis, the author of this work has implemented a web application that meets the initial requirements. The application provides the user of the tool a web-based graphical user interface. Using this interface it is possible to load in a BPMN model, insert a description of a certain path in this model and add some constraints to the model using regular expressions. Based on these inputs, the tool verifies if the provided path belongs to the business process semantics and is accepted by an automaton constructed from the regular expression. If these verifications succeed, the tool provides a proof of these statements. Provided with a proof, another user of this tool can verify the proof.

In addition to the tool, three example BPMN models of different size were designed. Firstly, these models are used as examples of how the input to the tool should look like. Secondly, these models provide a brief insight into what this tool is capable of.

Also, as a result, it has been shown that there is a way to connect business processes with zero-knowledge proofs and receive a value from it. This work could introduce new ideas for the researchers working on topics related to zero-knowledge or business processes.

Finally, the tool implemented in this thesis contributes to the goal of bringing zero-knowledge proofs more into use in practical applications.

Details regarding limitations of the the result tool of this thesis and future work are described in the next section.

7.2 Future work

In the current version of the tool implemented in this thesis, there are four main known limitations:

Firstly, the tool supports currently only six different BPMN elements in a business processes. This means that many of the already created business process models have to be redesigned to meet these requirements. Note that this does not mean that using other elements is not allowed, it just means that these other elements are not taken into account. Currently, the tool is not yet modular enough so introducing a new BPMN element requires an individual approach.

Secondly, the regular expression parser supports only a short list of grammars and as it processes the regular expression one character at a time, it sets another limitation for the tool implemented in this thesis – unique identifiers of model elements (short names) can be only one character long, fortunately, UTF-8 is supported, making the list of possible characters not too short.

Thirdly, the generator-prover-verifier application is currently recompiled with new input information each time when running a generator, proving or verification processes. Due to this, the execution time of these processes is longer.

Fourthly, in case something goes wrong, the graphical user interface does not always give enough feedback to the user about what went wrong and how to solve the problem, currently only main use cases are covered with feedback.

In order to improve the tool implemented in this thesis, there are six ideas for the future work:

Firstly, introducing new BPMN elements would help to make the tool more useful and easier to use. For example, introducing message flow and data object elements would allow to specify different participants in the business process and would allow to take information flow into account when describing statements to be proved.

Secondly, replacing the current regular expression parser with another more advanced tool would allow to support new grammars and remove the one-character limitation for the short names of elements in business process, making the tool much easier to use.

Thirdly, improving the graphical user interface in a way that the user of the tool would not have to insert the path (trace) input as a text, but could select the elements (vertices and edges) just by clicking on them, would improve the

user experience considerably, especially in cases the input BPMN contains many elements.

Fourthly, integrating the tool into PLEAK would allow to introduce the tool to a wider audience and through this, receive constructive feedback to improve the tool even more. Also, wider audience would not just bring new ideas, but would help to find issues in the code of the tool more easily, including the missing feedback messages in cases of errors.

Fifthly, adding a support for PE-BPMN models would make it possible to set constraints for the business process regarding different privacy-preserving technologies. Also, it would allow to consider in the proof statements the movements and disclosure of private (encrypted) information.

Finally, replacing different components with more optimal alternatives would reduce the execution times of different processes of the tool. For example, replacing the current solution for parallel reading with Waksman permutation networks [Wak68] would increase the speed of the process.

8 Acknowledgement

Firstly, I would like to thank my supervisor Dr. Peeter Laud for providing a fascinating and challenging idea for the thesis and guiding me through the full process. Thanks for being patient and providing support when needed.

Secondly, I would like to thank Pille Pullonen for designing a medium-sized example model for the tool, Sara Bellucini who's model we could use as a basis for the big example model for the tool, Martin Pettai for the help with libsnark and Tarmo Oja for suggesting solutions to different technical issues.

Finally, I would also like to thank Cybernetica AS for providing the opportunity and resources for the development environment of the tool implemented in this thesis.

References

[BCC+16] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient Zero-Knowledge Arguments for Arithmetic Circuits in the Discrete Log Setting. In *Advances in Cryptology – EUROCRYPT 2016*, pages 327–357. Springer, Berlin, Heidelberg, 2016.

[BCI+13] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Omer Paneth, and Rafail Ostrovsky. Succinct Non-interactive Arguments via Linear Interactive Proofs. In Amit Sahai, editor, *Theory of Cryptography*, pages 315–333, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[BFM88] Manuel Blum, Paul Feldman, and Silvio Micali. Non-Interactive Zero-Knowledge and Its Applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 103–112, New York, NY, USA, 1988. Association for Computing Machinery.

[BND+20] Sara Belluccini, Rocco De Nicola, Marlon Dumas, Pille Pullonen, Barbara Re, and Francesco Tiezzi. Verification of Privacy-Enhanced Collaborations. In *Proceedings of the 8th International Conference on Formal Methods in Software Engineering (FORMALISE)*, Seoul, South Korea, July 2020. IEEE Computer Society. to appear.

[BSCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[DDO08] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information and Software Technology*, 50(12):1281 – 1294, 2008.

[DRMR13] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer, Berlin, Heidelberg, second edition, 2013.

[Fei92] Joan Feigenbaum. Overview of interactive proof systems and zero-knowledge. *IEEE–Contemporary Cryptology*, pages 423–440, 1992.

[GGPR13] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic Span Programs and Succinct NIZKs without PCPs. In

Advances in Cryptology – EUROCRYPT 2013, pages 626–645, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. Association for Computing Machinery.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs That Yield Nothing but Their Validity or All Languages in NP Have Zero-Knowledge Proof Systems. *J. ACM*, 38(3):690–728, July 1991.

[Gro16] Jens Groth. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326. Springer, Berlin, Heidelberg, 2016.

[GWY+19] Zhangshuang Guan, Zhiguo Wan, Yang Yang, Yan Zhou, and Butian Huang. BlockMaze: An Efficient Privacy-Preserving Account-Model Blockchain Based on zk-SNARKs. *IACR Cryptol. ePrint Arch.*, 2019:1354, 2019.

[HBHW] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash Protocol Specification, version 2020.1.11. URL: `https://github.com/zcash/zips/raw/master/protocol/protocol.pdf`.

[liba] libff: a C++ library for Finite Fields and Elliptic Curves. URL: `https://github.com/scipr-lab/libff`.

[libb] libsnark: a C++ library for zkSNARK proofs. URL: `https://github.com/scipr-lab/libsnark`.

[Mog17] Torben Ægidius Mogensen. *Introduction to Compiler Design*. Springer International Publishing, 2017.

[NOV06] Minh-Huyen Nguyen, Shien Jin Ong, and Salil Vadhan. Statistical Zero-Knowledge Arguments for NP from Any One-Way Function. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 3–14, 2006.

[OMG10] OMG. BPMN 2.0 by Example: non-normative OMG document with BPMN 2.0 examples. URL: `https://www.omg.org/cgi-bin/doc?dtc/10-06-02.pdf`, June 2010.

[OMG11] OMG. Business Process Model and Notation (BPMN) Version 2.0. URL: https://www.omg.org/spec/BPMN/2.0/, January 2011.

[PET07] European Union. Press release: Privacy Enhancing Technologies (PETs). URL: http://europa.eu/rapid/press-release_MEMO-07-159_en.htm, May 2007.

[PMB17] Pille Pullonen, Raimundas Matulevičius, and Dan Bogdanov. PE-BPMN: Privacy-Enhanced Business Process Model and Notation. In *Proceedings of the 15th International Conference on Business Process Management (BPM)*, pages 40–56. Springer, 2017.

[PPO] Attribute based credentials – Privacypatterns.org. URL: https://privacypatterns.org/patterns/Attribute-based-credentials.

[PTMT19] Pille Pullonen, Jake Tom, Raimundas Matulevičius, and Aivo Toots. Privacy-Enhanced BPMN: A Multi-Level Approach to Information Disclosure Analysis. *Software and Systems Modeling*, 18:3235–3264, 2019.

[RFMP07] Alfonso Rodriguez, Eduardo Fernández-Medina, and Mario Piattini. A BPMN Extension for the Modeling of Security Requirements in Business Processes. *IEICE Transactions on Information and Systems*, E90D, March 2007.

[SIE] Defense Advanced Research Projects Agency (DARPA). Securing Information for Encrypted Verification and Evaluation (SIEVE) program Introduction. URL: https://www.darpa.mil/program/securing-information-for-encrypted-verification-and-evaluation.

[SKSB19] Martin Schanzenbach, Thomas Kilian, Julian Schütte, and Christian Banse. ZKlaims: Privacy-preserving Attribute-based Credentials using Non-interactive Zero-knowledge Techniques. In *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications - Volume 2: SECRYPT*, pages 325–332. INSTICC, SciTePress, 2019.

[Wak68] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, January 1968.

Appendix

I. Attachments

There are two directories and three files attached to this document:

- *bpmnzk-client/* – the source code of the client application of the implemented tool of this thesis. Installation instructions are in the *README.md* file (*./bpmnzk-client/README.md*);
- *bpmnzk-server/* – the source code of the server application of the implemented tool of this thesis. Installation instructions are in the *README.md* file (*./bpmnzk-client/README.md*);
- *example_ small.bpmn* – the small example model;
- *example_ medium.bpmn* – the medium-sized example model;
- *example_ big.bpmn* – the big example model.

II. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Aivo Toots**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Zero-Knowledge Proofs for Business Processes,

   supervised by Peeter Laud.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Aivo Toots
04.08.2020