KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in Computer Science and Engineering

Second cycle, 30 credits

Trust in your friends, on the ledger

Safer reproducible builds through decentralized distribution of .buildinfo files

JOHAN MORITZ

Stockholm, Sweden, 2022

Trust in your friends, on the ledger

Safer reproducible builds through decentralized distribution of .buildinfo files

JOHAN MORITZ

Degree Programme in Computer Science and Engineering
Date: March 31, 2022

Supervisor: Giuseppe Nebbione
Examiner: Mads Dam
School of Electrical Engineering and Computer Science
Host company: Subset AB
Swedish title: Detta är den svenska översättningen av titeln
Swedish subtitle: Detta är den svenska översättningen av undertiteln

Abstract

All theses at KTH are **required** to have an abstract in both *English* and *Swedish*.

Exchange students many want to include one or more abstracts in the language(s) used in their home institutions to avoid the need to write another thesis when returning to their home institution.

Keep in mind that most of your potential readers are only going to read your `title` and `abstract`. This is why it is important that the abstract give them enough information that they can decide is this document relevant to them or not. Otherwise the likely default choice is to ignore the rest of your document.
A abstract should stand on its own, i.e., no citations, cross references to the body of the document, acronyms must be spelled out, ...
Write this early and revise as necessary. This will help keep you focused on what you are trying to do.

Write an abstract that is about 250 and 350 words (1/2 A4-page) with the following components::

- What is the topic area? (optional) Introduces the subject area for the project.
- Short problem statement
- Why was this problem worth a Bachelor's/Master's thesis project? (*i.e.*, why is the problem both significant and of a suitable degree of difficulty for a Bachelor's/Master's thesis project? Why has no one else solved it yet?)
- How did you solve the problem? What was your method/insight?
- Results/Conclusions/Consequences/Impact: What are your key results/ conclusions? What will others do based upon your results? What can be done now that you have finished - that could not be done before your thesis project was completed?

The following are some notes about what can be included (in terms of LaTeX) in your abstract. Note that since this material is outside of the scontents environment, it is not saved as part of the abstract; hence, it does not end up on the metadata at the end of the thesis.

Choice of typeface with \textit, \textbf, and \texttt: *x*, **x**, and `x`

Text superscripts and subscripts with \textsubscript and \textsuperscript: A_x and A^x

Some useful symbols: \textregistered, \texttrademark, and \textcopyright. For example, copyright symbol: \textcopyright Maguire 2021, and some superscripts: ^{99m}Tc, A*, A^{\textregistered}, and A\texttrademark : ©Maguire 2021, and some superscripts: $^{\text{99m}}$Tc, A^*, A®, and A™. Another example: H\textsubscript{2}O: H_2O

Simple environment with begin and end: itemize and enumerate and within these \item

The following macros can be used: \eg, \Eg, \ie, \Ie, \etc, and \etal: *e.g.*, *E.g.*, *i.e.*, *I.e.*, *etc.*, and *et al.,*

The following macros for numbering with lower case roman numerals: \first, \second, \third, \fourth, \fifth, \sixth, \seventh, and \eighth: (i), (ii), (iii), (iv), (v), (vi), (vii), and $(viii)$.

Equations using \(xxxx \) or \[xxxx \] can be used in the abstract. For example: $(C_5O_2H_8)_n$ or

$$\int_a^b x^2\, dx$$

Even LaTeX comments can be handled, for example: % comment at end

Keywords

Keyword 1, Keyword 2, Keyword3

Choosing good keywords can help others to locate your paper, thesis, dissertation, …and related work.

Choose the most specific keyword from those used in your domain, see for example: the ACM Computing Classification System (`https://www.acm.org/publications/computing-classification-system/how-to-use`), the IEEE Taxonomy (`https://www.ieee.org/publications/services/thesaurus-thank-you.html`), PhySH (Physics Subject Headings) (`https://physh.aps.org/`), …or keyword selection tools such as the National Library of Medicine's Medical Subject Headings (MeSH) (`https:`

//www.nlm.nih.gov/mesh/authors.html) or Google's Keyword Tool (https://keywordtool.io/)

Mechanics:

- The first letter of a keyword should be set with a capital letter and proper names should be capitalized as usual.
- Spell out acronyms and abbreviations.
- Avoid "stop words" - as they generally carry little or no information.
- List your keywords separated by commas (",").

Since you should have both English and Swedish keywords - you might think of ordering them in corresponding order (*i.e.*, so that the nth word in each list correspond) - this makes it easier to mechanically find matching keywords.

Sammanfattning

Alla avhandlingar vid KTH **måste ha** ett abstrakt på både *engelska* och *svenska*.
Om du skriver din avhandling på svenska ska detta göras först (och placera det som det första abstraktet) - och du bör revidera det vid behov.

If you are writing your thesis in English, you can leave this until the draft version that goes to your opponent for the written opposition. In this way you can provide the English and Swedish abstract/summary information that can be used in the announcement for your oral presentation.
If you are writing your thesis in English, then this section can be a summary targeted at a more general reader. However, if you are writing your thesis in Swedish, then the reverse is true – your abstract should be for your target audience, while an English summary can be written targeted at a more general audience.
This means that the English abstract and Swedish sammnfattning or Swedish abstract and English summary need not be literal translations of each other.

The abstract in the language used for the thesis should be the first abstract, while the Summary/Sammanfattning in the other language can follow

Nyckelord

Nyckelord 1, Nyckelord 2, Nyckelord 3

Nyckelord som beskriver innehållet i uppsatsrapporten

Note that you may need to augment the set of language used in polyglossia or babel (see the file kththesis.cls). The following languages include those languages that were used in theses at KTH in 2018-2019, except for one in Chinese.
Remove those versions that you do not need.
If adding a new language, when specifying the language for the abstract use the three letter ISO 639-2 Code – specifically the "B" (bibliographic) variant of these codes (note that this is the same language code used in DiVA).

Use the relevant language for abstracts for your home university.

Acknowledgments

It is nice to acknowledge the people that have helped you. It is also necessary to acknowledge any special permissions that you have gotten – for example getting permission from the copyright owner to reproduce a figure. In this case you should acknowledge them and this permission here and in the figure's caption.
Note: If you do **not** have the copyright owner's permission, then you **cannot** use any copyrighted figures/tables/…. Unless stated otherwise all figures/tables/…are generally copyrighted.

I would like to thank xxxx for having yyyy.

Stockholm, March 2022
Johan Moritz

Contents

List of Figures

List of Tables

Listings

If you have listings in your thesis. If not, then remove this preface page.

List of acronyms and abbreviations

API	Application Programming Interface
DLT	Distributed Ledger Technology
DNS	Domain Name System
IPFS	InterPlanetary File System
RDBMS	Relational Database Management System
TLA	Temporal Logic of Actions

The list of acronyms and abbreviations should be in alphabetical order based on the spelling of the acronym or abbreviation.

Chapter 1

Introduction

This chapter describes the specific problem that this thesis addresses, the context of the problem, the goals of this thesis project, and outlines the structure of the thesis.

1.1 Background

Discussions on how to verify the lack of malicious code in binaries go at least as far back as to Ken Thompson's Turing award lecture [1] where he discusses the issues of trusting code created by others. In recent years, several attacks on popular packages within the Free Open Source Software (FOSS) have been executed [2] where trusted repositories have injected malicious code in their released binaries. These attacks question how much trust in such dependencies is appropriate. In an attempt to raise the level of trust and security in Free Open Source Software (FOSS), the reproducible builds projects [3] was started within the Debian community. Its goal was to mitigate the risk that a package is tampered with by ensuring that its builds are deterministic and therefore should be bit-by-bit identical over multiple rebuilds. Any user of a reproducible package can verify that it has indeed been built from its source code and was not manipulated after the fact simply by rebuilding it from the package's .buildinfo file. These metadata files for reproducible builds include hashes of the produced build artifacts and a description of the build environment to enable user-side verification. .buildinfo files are by this notion the crucial link to ensure reproducibility, which also means that a great deal of trust is assumed when using them. Current measures for validating .buildinfo files and their corresponding packages involve package repository managers and volunteers running rebuilderd [4] instances that test the reproducibility of every .buildinfo file added to the relevant package archive. This setup allows users to audit the separate build logs, thus confirming the validity of a particular package. However, because this would be a manual process and the different instances do not coordinate their work, it relies on the user judging on a case-by-case basis whether to trust a package or not.

Validating the aggregated results from many package builds could potentially be done through Distributed Ledger Technologies (DLTs). DLTs were popularized by Bitcoin [5, 6] for crypto-currencies but has wide ranging applications in trust related domains. A distributed ledger is a log of transaction held by many different nodes on a network. Transactions are validated, ordered and added to the network by a consensus algorithm to ensure that no single or small group of nodes can act maliciously. The log itself is commonly a tree or graph of hashes which allows proving that a particular transaction has happened in an efficient manner. Because of their distributed nature, DLTs are however hard to test and verify. One way to go about this without loosing accuracy **Precision?** is by modeling the system with formal specification tools that can validate the properties of the system design. Even though such a model is not a true representation of the system itself,

This project seeks to reduce some of the above mentioned burden from the user while increasing their trust in the software they use by investigating possible decentralized solutions for distributing and proving the correctness of .buildinfo files.

1.2 Problem

Equivalences between human readable source code and binaries are hard to prove **(cite)**. Likewise is it if the comparison is between source and a hash of a binary. A more easily proved variation of this problem is whether multiple binaries have been built from the same, potentially unknown, source. If the binaries are identical and we trust that the builder is not forging their results, we can be confident in that the binaries were all built the same way. The proof, though, is only as strong as our trust in the builder; an actor which could be compromised without us knowing. With multiple builders, we reduce this risk and our trust can increase likewise.

Distributing the workload creates the need for a system where the build results can be aggregated. Because users have different needs, they should be able to choose their own trust models and use the packages they trust based on the build results from the different builders. With this as background, we ask how such a system can be designed and implemented in order to maximize user trust in that the packages they use have been derrived from the correct source code.

1.2.1 Original problem and definition

The Debian Reproducible Builds project uses .buildinfo files to store checksums of derived artifacts. These files can serve as proof that a package has been built from source by a particular builder. Storing the aggregated .buildinfo files from multiple builders in a system could increase user trust in .buildinfo files, and therefore in packages. With this in mind we ask the following question:

- To what extent can distributed and decentralized storage secure the integrity of .buildinfo files?

1.2.2 Scientific and engineering issues

- Which distributed data storage solutions are applicable for .buildinfo files?
- How can we model a relevant system for efficient evaluation of integrity preservation?

1.3 Purpose

As society relies more and more heavily on software and digital infrastructure, threats to those technologies are increasingly more important to mitigate. By supplying additional safeguards to the way we manage software, we can make it harder for malicious actors to take advantage of users.

One current way of managing software safely is to first downloade its source code and then building it on our own machines. This way, we can be confident in that we are running the software we intend to run. Such a method, however, is time consuming and therefore not particularly user friendly. The purpose of this project is to give alternative solutions with a focus on user trust while not relying on users' building packages themselves. Further more, this project seek to increase trust and security in Free Open Source Software and reduce the risk of supply-chain attacks on package archives.

1.4 Goals

The main goal of this project is to formulate a plan for how to store .buildinfo files in such a way that their integrity is maintained. This involves understanding the purpose of and context within .buildinfo files exist. The storage plan should be formulated based on this context and written as a formal specification.

1.5 Research Methodology

The project will take three different phases. Initially, a pre-study focusing on reproducible builds, Distributed Ledger Technologies and formal specfcation will take place. Its purpose will be finding possible technologies, solutions and evaluation methods for solving the issues mentioned under section 1.2.2. With this initial phases finished, an appropriate storage strategy for ensuring the integrity of .buildinfo files and a methodology for modeling such a storage system is decided. The last phase involves producing and evaluating the model of said system to produce an answer to the original research question stated in 1.2.1.

1.6 Delimitations

While this project utilizes .buildinfo files and reproducible builds as its core problem domain, no builds or .buildinfo files are necessarily going to be produced during it. More specifically, the interessting part of .builfinfo files in terms of the project are their meta information and context in the software ecosystem. Their actual content and semantics are mainly irrelevant for the project, and will most likely be represented in an abstract manner in any implementations and artifacts.

Chapter 2

Background

To bring the reader up to speed, this chapter covers an introduction on reproducible builds as seen within the Debian project. It also describes the core ideas in Distributed Ledger Technologies, examplified most notably with the blockchain Hyperledger Fabric, and temporal logic and formal specifications written TLA$^+$. Important terminology such as Confidentiality, Integrity, Availability is also presented. The chapter ends with a review of previous work related to this project.

2.1 Confidentiality, Integrity, Availability (CIA)

Within information security, the terms confidentiality, integrity and availability are at the core of how researchers and security auditors describe the security of information systems [7]. They each relate to the respective security risk where an actor can read, write or hinder information when they should not have been able to do so. *Confidentiality* is the ability to stop unauthorized information leakage *i.e.,* you must be authorized to read the information. In a similar vein, *integrity* is the ability to stop unauthorized changes to information. *Availability* is the extent to which a a resource is guaranteed to be present and available to those authorized to manage it.

2.2 Pretty Good Privacy (PGP)

2.3 Reproducible builds

As a response to supply chain attacks on package archives for open source software, several projects have started within the linux community in order to raise build reproducibility [3]. Traditionally, linux distributions come with package managers (such as apt **(cite)** (apt) or pacman **(cite)** (pacman)) that help users installing and managing programs. While many packages have their source code available online and can be built directly from it by each user, package managers commonly have the functionality to download pre-built programs from an archive. This is convenient for the user but comes with security risks. Using pre-built packages relies on trusting the builder to use the correct source code and that any dependencies needed to build the package are themselves non-malicious.

Building a package reproducibly means it is bit-by-bit identical every time it is built [2]. Verifying its correctness can therefore rely on multiple parties, each building it separately, instead of trusting a single builder. Each builder can supply a hash of the built software which, if everything has been done correctly, should all be the same. Reproducible builds allows a separation between distributing the software artifact and its verification. Different efforts to make builds reproducible have used various strategies, but a core similarity between them is the use of some kind of specification for the build-environment.

TODO: Add note on percentage of packages that are reproducible on Debian

2.3.1 .buildinfo

In order for builds to be reproducible on different computers, the Debian project uses .buildinfo files to describe the necessary parts of the environment in which a package was first built. By recreating this environment on a different machine, build artifacts become identical, as long as the package is reproducible. .buildinfo files include, among other properties, name and version of the source package, architecture it was built on, checksums for the build artifacts as well as other packages available on the system [2]. The .buildinfo files origin and authenticity is given by the builder signing it with their private PGP key. A user can verify that a package has been built from source by comparing its checksum from **hash example** with the one in a corresponding .buildinfo file from a trusted source.

Currently, .buildinfo files are distributed in a centralized archive **(cite)**. Because this is a single-point-of-failure, if a malicious actor takes control of this archive, it could be very hard for users to know whether or not a package should be trusted.

2.4 Distributed Ledger Technology (DLT)

Storing and managing data is commonly done in databases such as Relational Database Management Systems (RDBMSs) or key-value stores **(cite)**. Because of these solutions' often centralized nature, they come with both integrity and availability risks **(cite)**. They can become single-point-of-failures. If that data storage is interrupted or manipulated, a system relying on it is at risk. Distributed Ledger Technologies are an alternative solution to data storage, mitigating the shortcomings of traditional, centralized methods. DLT is an umbrella term for several different technologies which rely on decentralized append-only logs [8] . The data stored in such a network cannot be changed by a central node. Instead, there has to be a consensus over the participants on how a change is to be made, followed by that change being propagated to all nodes in the network. Depending on the application, different solutions to how consensus is made and how the ledger itself is represented have been designed, each with its strengths and weaknesses.

The term is sometimes used interchangeably with blockchain, but while the latter uses a specific shape on its ledger, the former is more general. Other examples of DLTs are Certificate Transparency logs **(cite)** and peer-two-peer networks.

2.4.1 Merkle trees

Patent approved in 1982 [9] as a method for managing digital signatures, Merkle trees have since then been used for applications amongst file sharing and peer-to-peer communication [10], auditing certificate authorities [11] and running blockchains [12]. Merkle trees are directed acyclical graphs where each nodes' value is a hash based on the values of its child nodes. The leaves of the graph contain the data (or a hash thereof) relevant for a particular application, while the other nodes enable efficient proof mechanisms for validating the integrity of the data. For example: given a subgraph (*i.e.,* one with less data), verifying that its supergraph contains a certain value relies only on a subset of their differing nodes. This makes Merkle trees applicable to distributed systems where sending entire graphs between clients would be too expensive.

2.4.2 Consensus

When multiple systems or processes cooperate on a shared state, any change to this state needs to be agreed upon between the different entities. If no agreement, or consensus, can be found, the entities' different views of the state can drift away from each other. This can lead to an invalid system from which no meaningful progress can be made. The problem of creating consensus can be further complicated by assuming that entities can crash and be revived at any time, or even be malicious in the messages they send to the network.

A number of consensus algorithms exists, serving various applications. One way to differentiate them is whether they are proof or voting based [13]. In a proof based consensus algorithm, only the party that has provided a certain proof is allowed to change the data. Such algorithms can be found in some public ledger blockchains, such as bitcoin. The proof itself can be, for example, finding a number given certain constraints, which is known as proof-of-work. With proof-of-work, the greater computational investment any one participant makes, the greater is the probability that they will be allowed to change the blockchain. However, the greater the computational power is in the whole network, the more limited is any one participants possibility to control or use it maliciously. Other proof based algorithms exist, but they are all centered on connecting responsibility with some type of resource investment. Voting based consensus algorithms on the other hand relate more to a more intuitive understanding of agreement, *i.e.,* democratic voting. Agreement is made only when a certain fraction of the nodes have voted in acknowledgment to a certain decision. This relies on knowing how many nodes there are on the network in total, making voting based consensus less usable in certain scenarios. While simple in idea, a voting based consensus algorithm can become complicated in practice. The algorithm should not only be able to find consensus in perfect conditions, instead a realistic solution should work even if some nodes on the network crashes and, perhaps, even if some nodes are malicious. A consensus algorithm that can handle both of these kinds of issues is called Byzantine resilient [14] or that it has Byzantine fault tolerance [13]. If the algorithm only handles crashes but not malicious actors it has Crash fault tolerance.

Paxos??

2.4.3 Blockchain

Originally described for Bitcoin [6, 5], blockchain is a technology based on Distributed Ledger Technology for storing transactions without needing a centralized organization. Transactions are represented as simple strings of characters which allows them to model essentially anything. This is why blockchains can be used for a broad spectrum of applications; *e.g.,* currencies, ownership contracts etc. **(cite)**. The name stems from the setup of a blockchain ledger where groups of, closely related in time, transactions are appended together as a block to the current chain by including a hash of the previous block in the latest one. By grouping transactions together, the throuput of the network improves. A consensus algorithm is used to create a total ordering of the blocks so that every node on the network eventually holds the same ledger.

TODO: Add figure for how one block connects to the next.

TODO: Public vs Private vs Consortium blockchain

2.4.4 Peer-to-peer

TODO: Add information about IPFS

TODO: Summaries several different DLT

2.5 Hyperledger Fabric

Hosted by the Linux Foundation[15], the Hyperledger Fabric, or Fabric, is a permissioned blockchain framework with a novel and flexible approach to DLT. A Fabric network can, for example, choose a consensus algorithm suitable for that particular usecase and define specific requirements for when a change to the ledger may be allowed [16]. This section will describe and discuss the main components of a Hyperledger Fabric network and how they work together.

2.5.1 Overview

The Hyperledger Fabric ledger is permissioned. Compared to a public one (such as Bitcoin [6]), this means that it is only available to certain participants. This is regulated by Membership Service Providers (MSPs) on the network, verifying the identities of nodes through Certificate Authorities (CAs) **(cite)**. Besides MSPs, the nodes on the network can take on one of the roles of *peer* or *orderer*. Every node on network belongs to some organization whos'

MSP determines its role. In this sense, a Fabric network is a network of organizations rather than one of nodes.

Every peer stores' the network's entire blockchain ledger and validates transactions and changes to the ledger. Orderers, on the other hand, clump together transactions into blocks and delivers them to the peers. This separation of concern is one of the unique features of Hyperledger Fabric, and makes consensus algorithm selection possible. Changes onto the ledger are made by invoking smart contracts, called chaincodes within Fabric. Chaincodes are authorized programs that are run by peers on the network. If multple peers (according to its endorcement policy) get the same result from running the chaincode, any updates are written to the ledger. For performance reasons, a key-value store representing the current "world-state" is continuously derrived from the ledger and stored on the peers. This allows both reading and writing to happen without going through the entire ledger itself.

TODO: Bootstraping ordering service with a genesis block containing a configuration transaction [16]

2.5.2 Transaction flow

A transaction in Hyperledger Fabric goes through a number of steps before a change to the ledger happens. Figure 2.1 shows an overview of an example transaction. First, a client application invokes a particular chaincode by (as of version 2.4) sending a transaction proposal to the Fabric gateway on the network. The gateway is a service running on a peer which takes care of the transaction details, allowing the client to focus on application logic [17]. After receiving the proposal from the client, the gateway finds the peer within their own organization with the longest ledger, the *endorsing* peer, and forwards it to them. The endorsing peer runs the transaction (*i.e.,* chaincode) and notes what parts of the world-state it had to read from and which it will write to (the *read-write set*). This information together with the chaincode's Endorsement Policy informs which organizations has to accept, or endorse, the transaction for it to be valid. Only at the time when every necessary organization have endorsed the transaction can any change be made to the ledger.

The Fabric gateway is responsible for forwarding endorsement requests with the transaction proposal to peers of each necessary organization and gathering their responses. Each of these peers will then run the transaction proposal and sign their endorsement for it with their private key if they deem it correct. The gateway receives the read-write sets and endorsements, validates

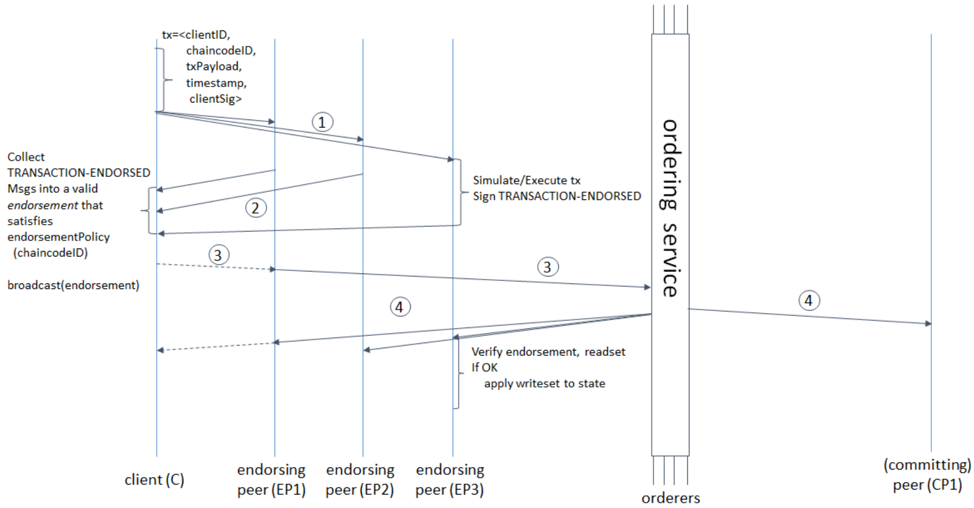

Figure 2.1: A swimlane sequence diagram over the Hyperledger Fabric transaction flow. Note that this is for an earlier version from before the Fabric gateway. As of version 2.4, the behaviour of the client in the diagram is instead performed by the gateway, while the client only initiates a transaction. The image [20] was created by Hyperledger and is licensed under CC BY 4.0 [21].

them, and sends a final version of the transaction to the ordering service. The actual transaction contains the read-write set and the endorsements from the different organizations.

As the ordering service runs on other orderer nodes from the peers, how the ordering service is implemented is completely separated from the functionality of the peers. Its purpose is to group transactions into blocks and order and distribute them to all the peers on the network. By default, Fabrics' ordering service uses Raft, which is a voting based crash-fault tolerant consensus algorithm [18]. Attempts have been made to add a Byzantine-fault tolerant ordering service to Fabric [19], but so far none have been added to the project.

When a peer receives a block from the ordering service they add it to their locally stored ledger, validate each transaction and, if valid, update its local world-state according to the transaction write set. They also notify the client application of the transactions status. Validation has two parts. First, the transaction must fulfill its endorsement policy, and, secondly, the subset of the world-state contained in the read set must not have changed. All transactions, valid and invalid, are added to the ledger, but they are marked to know which ones are which.

```
OR('Org1.member', AND('Org2.member', 'Org3.member'))
```

Figure 2.2: Endorsement policy syntax example

2.5.3 Endorsement Policies

An update to the Fabric ledger is only possible if the endorsement policy relevant to a transaction has been fulfilled. Endorsement policies describe which and how many organizations or peers that need to run and endorse a transaction proposal. They are defined as logical formulas referencing relevant organization and role for the peers that have to endorse it. Figure 2.2 shows an example endorsement policy with three organizations called Org1, Org2 and Org3 where all endorsing peers must have the *member* role.

An endorsement policy can be defined on three different granularity levels; chaincode-, collection- and key-level. Chaincode- and collection-level policies are decided on when a chaincode is commited to the network. The former is a general policy which always has to be fulfilled everytime the chaincode submitted as a transaction. Collection-level policies on the other hand are rules for when a chaincode reads or writes from a private collection. A chaincode can have multiple private collections, each with its own collection-level policy. When a chaincode accesses such a collection, the additional policies have to be satisfied as well. Key-level policies are similar to collection-level ones in that they only become relevant when a chaincode touches a subset of the world-state. A difference being that key-level policies are declared in the execution of a chaincode. A usecase for this is when adding a new asset to the ledger with a specific owner. By setting a key-level endorsement policy to the key of the asset, any transaction that tries to change the asset will have to be endorsed by, for example, its owner before it is commited to the ledger.

2.5.4 Chaincode

Many current blockchains have some notion of a smart contract [5], *i.e.,* methods to programmatically change the ledger only when certain properties, or contracts, have been fulfilled. Smart contracts make blockchains more general-purpose as they can be used to model many different protocols and applications. **examples plz**

In Hyperledger Fabric, smart contracts are called chaincode. As there is no inherent asset on the Fabric ledger over which peers can make transactions, chaincodes are the only way the ledger can be changed. They are in other words the only way to create, update and remove assets. Chaincodes has to follow the *fabric-contract* Application Programming Interface (API) to be run by peers on the network, but what language they can be written in is flexible. The official language options are Java, Javascript and Go, but other languages can be supported in theory. Listing 1 examplifies how chaincodes can look and the mechanisms through which it interacts with the worldstate. A chaincode smart contract interacts, through the fabric-contract API, with the world-state key-value representation of the ledger and can query, update and set values in it.

Running a chaincode on the network does not update the ledger directly (see 2.5.2 for how ledger modifications are implemented). Instead a read-write set of the keys that have been queried from and the changes to the ones that have been set during the execution of the chaincode are generated. The result of the chaincode is this read-write set, which is then further used to update the ledger itself.

TODO: Find some good references for chaincode besides the official documentation.

TODO: Describe the core api for chaincode

2.6 Formal specification

Distributed systems can be complex to design and hard to reason about. This is mainly due to the non-deterministic interleaving of subsystems acting individually **(cite)**. To ensure that a design of a distributed system works as intended, formal specification notation and tools can be used. These do not generate actual implementations of a system, but instead a precise description of system properties [23]. By modeling a system with a formal specification, its properties can be automatically verified or refuted. It also forces the developer to think in terms of system invariants instead of *how* the system should be constructed. Because formal specifications are models of (and not actual) systems, there is no guarantee that they correspond correctly to a real implementation. However, because they do not include as many concrete details, testing the correspondance between a specification and an implementation can be done on a higher abstraction level compared to the unit-testing common in software verification.

TODO: Summaries several different formal specification methods

Listing 1 Modified chaincode excerpts from Hyperledgers sample project fabcar [22]. The examples showcases the main chaincode API endpoints `getState` and `putState` for retrieving and, respectively, updating the ledger.

```
/*
* Copyright IBM Corp. All Rights Reserved.
*
* SPDX-License-Identifier: Apache-2.0
*/
⋮
async queryCar(ctx, carNumber) {
  const carAsBytes = await ctx.stub.getState(carNumber);
  if (!carAsBytes || carAsBytes.length === 0) {
    throw new Error(`${carNumber} does not exist`);
  }
  return carAsBytes.toString();
}

async createCar(ctx, carNumber, make, model, color, owner) {
  const car = {
    color,
    docType: 'car',
    make,
    model,
    owner,
  };

  await ctx.stub.putState(carNumber,
                          Buffer.from(JSON.stringify(car)));
}
```

$\langle[x \mapsto 0, y \mapsto 0, \ldots], [x \mapsto 1, y \mapsto 0, \ldots], [x \mapsto 1, y \mapsto 1, \ldots], [x \mapsto 2, y \mapsto 2, \ldots], \ldots\rangle$

Figure 2.3: A behaviour where the variables x and y increment concurrently.

2.7 TLA+

TLA+ [24] is a language for formal specification where a model describes a systems' behaviour over time. It has been used successfully in industry to verify and finding bugs in distributed systems [25, 26] and can be written in both a mathematical notation style and the more pseudocode-esque PlusCal. The specification can then be verified by the TLC tool which simulates executions of the model in order to find potential faults. TLA+ is built on the Temporal Logic of Actions (TLA) [27], but adds improvements for writing more modular and larger specifications.

2.7.1 Temporal Logic of Actions (TLA)

The semantics of TLA are based on infinite sequences of states called behaviours. Here, a state s is a mapping from symbolic variable names to values *e.g.,* $[x \mapsto 1, y \mapsto "a", z \mapsto 42]$. Behaviours represent the history of states that a program execution enters, one state for each atomic change. A program incrementing a counter could for example have the behaviour $\langle[x \mapsto 0, \ldots], [x \mapsto 1, \ldots], [x \mapsto 1, \ldots], [x \mapsto 2, \ldots], \ldots\rangle$. Note that x does not increment at every state in this behaviour. This is an example of *stuttering i.e.,* x is allowed to remain the same or increment at every step. Stuttering is an important concept because it allows logical formulas, or programs, to be modular. As an example of this, we can imagine a second counter program running concurrently with the first. The two programs are both incrementing variables but not necessarily at the same time, so at certain states we need stuttering to descibe the combined program. An example of this is shown in figure 2.3.

2.7.1.1 Propositional logic in TLA

The core of TLA is propositional logic **propositional?** with common connectives **connectives?** such as $\wedge$, $\vee$, $\implies$, $\neg$, $\forall$ and $\exists$.

$g \triangleq x + y - 2$
$s \triangleq [x \mapsto 0, y \mapsto 2]$
$s[\![g]\!] \equiv s[\![x]\!] + s[\![y]\!] - 2 \equiv 0$

Figure 2.4: Semantic meaning of a state function g given a state s.

2.7.1.2 State Functions and Predicates

TLA allows the use of "regular" mathematics which can be used to form expressions such as $x^2 + 5 * y - 3$. These make up the body of non-boolean state functions and their boolean equivalent predicates. The meaning of such expressions is evaluated relative a state. This is done by substituting the value of a variable in the state for the same variable in the expression. This definition can be written
$s[\![f]\!] \triangleq f(\forall v : s[\![v]\!]/v)$
where

- f is a state function or a predicate
- $\triangleq$ signifies equality by *definition*
- $s[\![v]\!]$ is the value of variable v in state s
- $s[\![v]\!]/v$ substitutes $s[\![v]\!]$ for v

Figure 2.4 shows an example of a substitution in a state function.

2.7.1.3 Actions

So far TLA is quite similar to propositional and predicate logic **which one?** but with the addition of *actions*, it changes quite radically. An action can be seen as the link between two states in the behaviour of a program. It describes change from one state to the next. Syntactically, this is done by separating variables into two groups: *un-primed* variables x, y, $z, \ldots$ and *primed* variables x', y', $z', \ldots$. Primed variables represent the value of their un-primed counterpart in the next state, and an action is just a boolean expression with both primed and unprimed variables. Incrementing a variable can for example be written as the action $x' = x + 1$. In natural language, this means that the variables value in the next state should be one more than in the previous. Similarly to how we did it for state functions and predicates, we can define the semantic meaning of an action $\mathcal{A}$ as
$s[\![\mathcal{A}]\!]t \triangleq \mathcal{A}(\forall v : s[\![v]\!]/v, t[\![v]\!]/v')$

where s and t are both states and t follows directly after s. In other words, actions are relations between states that are following each other in time.

The changes in the increment counter program can, as mentioned, be represented by an action. To support stuttering we can add the second possibility that the variable stays the same. This is written as the disjunction $(x' = x + 1) \lor (x' = x)$ *i.e.,* either x is incremented or it stays the same. As it turns out, this is a commonly used concept when writing specification so TLA provides a shorthand for it written $[\mathcal{A}]_f$. We pronounce $[\mathcal{A}]_f$ as "square $\mathcal{A}$ sub f". f here is any state function, but is often a sequence of the variables that are allowed to stutter. For example, with the increment program we can write $[x' = x + 1]_{\langle x \rangle}$.

2.7.1.4 Temporal operators

Where actions describe a single step of change between two states, we have already mentioned that program executions are represented in TLA by behaviours *i.e.,* sequences of states. To be able to describe whole behaviours we need some way of lifting actions from acting on pairs of states to sequences of states. We can perform this lifting in TLA with the *always* temporal operator, written as $\Box\mathcal{A}$. Informally, this is a formula that is true only if the action is true for all pairs of states in a behaviour. More generally, we can write $\Box F$ where F is a logical formula built from predicates and actions. For a behaviour $\langle s_0, s_1, s_2, \ldots \rangle$, the operator can be defined as

$\langle s_0, s_1, s_2, \ldots \rangle [\![\Box F]\!] \triangleq \forall n \in \mathbb{N} : \langle s_n, s_{n+1}, s_{n+2}, \ldots \rangle [\![F]\!]$

where $\mathbb{N}$ is the set of natural numbers. To understand this notation, it should be noted that $\langle s_0, \ldots \rangle [\![F]\!]$ is true if and only if F is true in the behaviours *first* state s_0.

Besides $\Box$, another common temporal operator is called *eventually* and written $\Diamond$. $\Diamond F$ denotes a formula that will be true at some point, and can be derrived in terms of the *always* operator as

$\Diamond F \equiv \neg\Box\neg F$

This is equivalent to the definition

$\langle s_0, s_1, s_2, \ldots \rangle [\![\Diamond F]\!] \triangleq \exists n \in \mathbb{N} : \langle s_n, s_{n+1}, s_{n+2}, \ldots \rangle [\![F]\!]$

i.e., F should be true in *some* state s_i. Combinations of $\Diamond$ and $\Box$ can describe some interesting properties such as

- $\Box\Diamond F$, or *infiniently often*
- $\Diamond\Box F$, or *from some point onwards*
- $\Box(F \implies \Diamond G)$, or *leads to*

The last one means that if F is true in some state, G has to be true at the same or a later state. This can also be written with the shorthand $F \leadsto G$.

2.7.1.5 TLA formulas

Not all combinations of the above mentioned logical and temporal operators are allowed TLA formulas. Instead, only formulas built from simple predicates (with no temporal operators or actions) or $\Box[\mathcal{A}]_f$ for some action $\mathcal{A}$ and state function f can be used. This is most noticable from a typical TLA specification of a program. With an initial state predicate $init_\Phi \triangleq x = 0$ and an action describing the next state $next_\Phi \triangleq x' = x + 1$, we get the TLA formula

$\Phi \triangleq init_\Phi \vee \Box[next_\Phi]_x$

i.e., either the value of x is equal to 0 or it is one greater or equal to the value of x in the previous state.

2.7.2 TLC Model Checker

TLA+ is part of a toolbox made for supporting the creation and testing of formal specifications. Instead of validating a TLA model by hand, the TLC model checker allows mechanic verification by simulating its possible executions while looking for invalid states and other errors [28]. If TLC finds a problem, it terminates the simulation and notifies the user with a timeline of the behaviour that lead up to the particular error. The user can then improve their model, and gain a better understanding of their specification and system. Because verifying a specfcation means TLC has to simulate every possible behaviour, this can take a long time to do. A practical way to resolve this is to limit the state space in the model, but that also limits the type of properties TLC can test. For example, ensuring the absence of integer overflow from the sum of two 32-bit integers would imply testing all possible pairs, but this would take far too long to test. If we instead limit the possible integer values to, for example, $0 \ldots 5$ the simulation might terminate without error but we have clearly not managed to validate the absence of overflows.

Instead, the TLC model checker is instead more appropriate for validating time-related and concurrency problems. Here follows a short description of some of the properties that TLC can test.

2.7.2.1 Safety

A specification that can never do anything “bad” is said to follow its *safety* property. Variations of this is common to test in software engineering practices such as unit-testing. Safety checking in TLC is straightforward as it is done by simply testing every new state during simulation for the relevant property. To examplify this, we can consider type invariants as safety properties. An variable might for example only be allowed to be an integer and nothing else.

2.7.2.2 Liveness

While a safe program is guaranteed to never do anything bad, whether or not it will actually do anything at all is not certain. Liveness properties describe what "good" things a specifcation necessarily will do. Some examples can be that an execution is guaranteed to terminate, that no deadlock between processes can happen, or that all processes will progress.

In TLA+, liveness properties are written with temporal operators such as $\Box$ or $\Diamond$. Reaching a certain value can for example be written as $\Diamond\Box F$ *i.e.,* eventually a state is reached from which F is always true.

2.7.2.3 Fairness

Two additional examples of liveness properties are the weak and strong fairness properties. A *fair* specification is guaranteed to execute an action as long as it is possible. If the action is weakly fair then it will be performed as long as it is repeatedly possible to do so. If it is strongly fair, then it will be executed unless it is no longer possible to do so at least once. More formally they are defined, for action $\mathcal{A}$ and state function f, as

$WF_f(\mathcal{A}) \triangleq (\Box\Diamond\langle\mathcal{A}\rangle_f) \vee (\Box\Diamond\neg Enabled\langle\mathcal{A}\rangle_f)$

$SF_f(\mathcal{A}) \triangleq (\Box\Diamond\langle\mathcal{A}\rangle_f) \vee (\Diamond\Box\neg Enabled\langle\mathcal{A}\rangle_f)$

where

- $\langle\mathcal{A}\rangle_f \triangleq \mathcal{A}\wedge(f' \neq f)$ *i.e.,* the action A is executed and changes variables in f
- $\neg Enabled\langle\mathcal{A}\rangle_f$ means that $\mathcal{A}$ is impossible to execute.

2.7.3 PlusCal

TLA is, as can be seen in section 2.7.1, quite different from a regular programming language. This has the benefit of being simple and flexible, but can result in hard-to-read code as specifications start to grow in size. It is also not entirely clear from its syntax how an algorithm described in for example pseudocode can be translated to it.

PlusCal is an alternative syntax specifically made for describing concurrent algorithms exactly while bringing the simplicity of writing pseudocode [29]. It is written within a comment in a TLA⁺ file and translated into TLA⁺ as part of the parsing step of TLC. PlusCal is

2.8 Pedersen Commitment Scheme

2.9 Related work area

2.9.1 Decentralized File Distribution

A number of proposals for distributed and decentralized data management are recorded in the research litterature. Ince, Ak, and Gunay [30] describe a combination of blockchain and the P2P network InterPlanetary File System (IPFS) for package storage. They envision a proof-of-work consensus algorithm based on the act of rebuilding a package where builders gain rewards in terms of a productiviy score with every rebuild they add to the ledger. At a certain score, a builder is promoted to an approver and can validate other builders rebuilds. Because blockchains are limited in storage space, only packages' addresses are stored in the ledger with the actual artifacts being distributed through IPFS. While an interesting concept, no prototype or evaluation is provided by the paper. A similar approach was taken by Zichichi, Ferretti, and D'Angelo [31] but with the application of managing personal data. They managed different kinds of personal data by partitioning it through smart contracts, each with its own access list for node authorization.

Instead of using blockchain and P2P technologies together, Blähser, Göller, and Böhmer [32] show a prototype system for distributed package management using only a peer-to-peer technology called Hypercore protocol. They rely on Hypercore protocol to act both as a distributed file system as well as an append only ledger. This approach is convenient but does not have any built-in safeguards against malicious actors or packages. On the other hand, Liu et al. [33] used blockchain technology single-handledly to construct a distributed Domain Name System (DNS). Because of the limited amount of data in a zone file, no separate distributed file system was needed for their application. Each request to the service was done through a smart contract and their prototype system were able to serve requests with a 0.006025 seconds average response delay and a failure resolution rate of 2.14%.

2.9.2 Formal verification of Smart contracts

Writing smart contracts that works correctly is hard. To aid this, several researchers have looked into using formal methods to verify them. Bhargavan et al. [34] translated the Solidity language for writing smart contracts into the general-purpose language F* made for program verification. The authors then used the type system of F* to enforce certain safety properties in the smart contract. Beckert et al. [35] took a similar approach but for the Hyperledger Fabric chaincode. By adding pre- and postconditions in comments to chaincode written in Java and translating these to Java Modeling Language **(acronym)** they managed to deduce smart contract safety properties statically. In a slightly different direction, Latif, Rehman, and Zafar [36] modelled a blockchain for waste management using TLA+. Their initial system description was written in UML **(acronym, cite)** which turned out to be relatively straightforward to translate into a formal specification language. The system's safety and liveness properties could then be validated with the TLC model checker.

Xu and Fink [37] examplifies how a design-by-contract methodology [38] when designing smart contracts can be enacted with TLA+ modeling and verification. They specify a system for purchasing and selling estates directly in TLA+ and uses its safety checks to ensure that any drawn up contract (between buyer and seller) is followed. As part of their specification, they also include checks for some vulnerabilities common in smart contracts. These include eliminating invalid and additional states and *trap doors i.e.,* invalid events or transitions.

In their PhD thesis, Elsayed [39] implemented a solution for trapping computer worms in a network. When a worm was detected by a host on the network, its signature was added to a blockchain built on Hyperledger Fabric. This acted as an immutable threat log, used by the network to be able to find and stop the worm. From the use of a blockchain, the log has high availability and other interesting security properties that makes it resistant to attacks. Besides implementing the system, the author moddeled with TLA+ to model check its correctness, safety and liveness properties. This model included both the ineractions between smart contracts as well as the consensus algorithm used by the blockchain.

To reduce the friction between writing a veryfiable specification of a smart constract and implementing it in a regular programming language, some researchers have worked on intermediate representations from which both specifactions and implementations can be derived. Kolb et al. [40]

describe the *Quartz* language for this purpose. *Quartz* is a statically typed language with which smart contracts are written as transitions in a finite-state machine. Simple authorization requirements and invariants can be annotated onto the transitions. An implementation in the Solidity smart contract language and a model in PlusCal can be derived from a *Quartz* program. The finalized translation in TLA⁺ includes a specification of the Solidity execution environment to allow a wider set of requirements to be verified while retaining ease of use. In a series of case studies they found that smart contracts written in *Quartz* were on average 0.68 the number of lines of equivalent Solidity programs, while keeping execution overheads at less than 20% for all cases but one.

TODO: Add work on infering formal specifications from logs

2.10 Summary

Sammanfattning

Det är trevligt att få detta kapitel avslutas med en sammanfattning. Till exempel kan du inkludera en tabell som sammanfattar andras idéer och fördelar och nackdelar med varje - så som senare kan du jämföra din lösning till var och en av dessa. Detta kommer också att hjälpa dig att definiera de variabler som du kommer att använda för din utvärdering.

It is nice to have this chapter conclude with a summary. For example, you can include a table that summarizes other people's ideas and benefits and drawbacks with each - so as later you can compare your solution to each of them. This will also help you define the variables that you will use for your evaluation.

References

[1] Ken Thompson. "Reflections on trusting trust." In: 27.8 (1984), p. 3.

[2] Chris Lamb and Stefano Zacchiroli. "Reproducible Builds: Increasing the Integrity of Software Supply Chains." In: *IEEE Software* (2021). Conference Name: IEEE Software, pp. 0–0. ISSN: 1937-4194. DOI: 10.1109/MS.2021.3073045.

[3] *Reproducible Builds — a set of software development practices that create an independently-verifiable path from source to binary code*. URL: https://reproducible-builds.org/ (visited on 02/07/2022).

[4] *Public rebuilderd instances*. URL: https://rebuilderd.com/ (visited on 02/07/2022).

[5] Massimo Di Pierro. "What Is the Blockchain?" In: *Computing in Science Engineering* 19.5 (2017). Conference Name: Computing in Science Engineering, pp. 92–95. ISSN: 1558-366X. DOI: 10.1109/MCSE.2017.3421554.

[6] Satoshi Nakamoto. "Bitcoin: A Peer-to-Peer Electronic Cash System." In: (), p. 9.

[7] Spyridon Samonas and David Coss. "THE CIA STRIKES BACK: REDEFINING CONFIDENTIALITY, INTEGRITY AND AVAILABILITY IN SECURITY." In: (), p. 25.

[8] Niclas Kannengießer et al. "Trade-offs between Distributed Ledger Technology Characteristics." In: *ACM Computing Surveys* 53.2 (Mar. 31, 2021), pp. 1–37. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3379463. URL: https://dl.acm.org/doi/10.1145/3379463 (visited on 02/16/2022).

[9] Ralph C. Merkle. "Method of providing digital signatures." U.S. pat. 4309569A. Univ Leland Stanford Junior. Jan. 5, 1982.

[10] Erik Daniel and Florian Tschorsch. "IPFS and Friends: A Qualitative Comparison of Next Generation Peer-to-Peer Data Networks." In: *arXiv:2102.12737 [cs]* (Jan. 18, 2022). arXiv: 2102.12737. URL: http://arxiv.org/abs/2102.12737 (visited on 02/02/2022).

[11] Ben Laurie, Adam Langley, and Emilia Kasper. *Certificate Transparency*. Request for Comments RFC 6962. Num Pages: 27. Internet Engineering Task Force, June 2013. DOI: 10.17487/RFC6962. URL: https://datatracker.ietf.org/doc/rfc6962 (visited on 02/17/2022).

[12] Nazanin Zahed Benisi, Mehdi Aminian, and Bahman Javadi. "Blockchain-based decentralized storage networks: A survey." In: *Journal of Network and Computer Applications* 162 (July 2020), p. 102656. ISSN: 10848045. DOI: 10.1016/j.jnca.2020.102656. URL: https://linkinghub.elsevier.com/retrieve/pii/S1084804520301302 (visited on 02/02/2022).

[13] Giang-Truong Nguyen and Kyungbaek Kim. "A Survey about Consensus Algorithms Used in Blockchain." In: *Journal of Information Processing Systems* 14.1 (Feb. 28, 2018), pp. 101–128. DOI: 10.3745/JIPS.01.0024. URL: https://doi.org/10.3745/JIPS.01.0024 (visited on 02/11/2022).

[14] Michael J. Fischer. "The consensus problem in unreliable distributed systems (a brief survey)." In: *Foundations of Computation Theory*. Ed. by Marek Karpinski. Red. by G. Goos et al. Vol. 158. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 127–140. ISBN: 978-3-540-12689-8 978-3-540-38682-7. DOI: 10.1007/3-540-12689-9_99. URL: http://link.springer.com/10.1007/3-540-12689-9_99 (visited on 02/17/2022).

[15] *Linux Foundation Projects*. Linux Foundation. URL: https://www.linuxfoundation.org/projects/ (visited on 03/23/2022).

[16] Elli Androulaki et al. "Hyperledger fabric: a distributed operating system for permissioned blockchains." In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18: Thirteenth EuroSys Conference 2018. Porto Portugal: ACM, Apr. 23, 2018, pp. 1–15. ISBN: 978-1-4503-5584-1. DOI: 10.1145/3190508.3190538. URL: https:

//dl.acm.org/doi/10.1145/3190508.3190538 (visited on 02/21/2022).

[17] *Fabric Gateway — hyperledger-fabricdocs main documentation*. URL: https://hyperledger-fabric.readthedocs.io/en/release-2.4/gateway.html (visited on 02/22/2022).

[18] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm." In: (), p. 16.

[19] Artem Barger et al. "A Byzantine Fault-Tolerant Consensus Library for Hyperledger Fabric." In: *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). May 2021, pp. 1–9. DOI: 10.1109/ICBC51069.2021.9461099.

[20] *Transaction Flow — hyperledger-fabricdocs main documentation*. URL: https://hyperledger-fabric.readthedocs.io/en/release-2.4/txflow.html (visited on 03/23/2022).

[21] *Creative Commons — Attribution 4.0 International — CC BY 4.0*. URL: https://creativecommons.org/licenses/by/4.0/ (visited on 03/23/2022).

[22] *hyperledger fabric-samples*. GitHub. URL: https://github.com/hyperledger/fabric-samples (visited on 03/24/2022).

[23] Axel van Lamsweerde. "Formal specification: a roadmap." In: *Proceedings of the conference on The future of Software engineering - ICSE '00*. the conference. Limerick, Ireland: ACM Press, 2000, pp. 147–159. ISBN: 978-1-58113-253-3. DOI: 10.1145/336512.336546. URL: http://portal.acm.org/citation.cfm?doid=336512.336546 (visited on 03/02/2022).

[24] Leslie Lamport. "Specifying Concurrent Systems with TLA+." In: (), p. 372.

[25] Rajeev Joshi et al. "Checking Cache-Coherence Protocols with TLA+." In: *Formal Methods in System Design* 22.2 (Mar. 2003), pp. 125–131. ISSN: 0925-9856, 1572-8102. DOI: 10.1023/A:1022969405325. URL: http://link.springer.com/10.1023/A:1022969405325 (visited on 03/02/2022).

[26] Chris Newcombe et al. "How Amazon web services uses formal methods." In: *Communications of the ACM* 58.4 (Mar. 23, 2015), pp. 66–73. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/2699417. URL: https://dl.acm.org/doi/10.1145/2699417 (visited on 03/02/2022).

[27] Leslie Lamport. "The temporal logic of actions." In: *ACM Transactions on Programming Languages and Systems* 16.3 (May 1994), pp. 872–923. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/177492.177726. URL: https://dl.acm.org/doi/10.1145/177492.177726 (visited on 03/02/2022).

[28] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model Checking TLA+ Specifications." In: *Correct Hardware Design and Verification Methods*. Ed. by Laurence Pierre and Thomas Kropf. Red. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen. Vol. 1703. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 54–66. ISBN: 978-3-540-66559-5 978-3-540-48153-9. DOI: 10.1007/3-540-48153-2_6. URL: http://link.springer.com/10.1007/3-540-48153-2_6 (visited on 03/01/2022).

[29] Leslie Lamport. "The PlusCal Algorithm Language." In: *Theoretical Aspects of Computing - ICTAC 2009*. Ed. by Martin Leucker and Carroll Morgan. Vol. 5684. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 36–60. ISBN: 978-3-642-03465-7 978-3-642-03466-4. DOI: 10.1007/978-3-642-03466-4_2. URL: http://link.springer.com/10.1007/978-3-642-03466-4_2 (visited on 03/08/2022).

[30] M. Numan Ince, Murat Ak, and Melih Gunay. "Blockchain Based Distributed Package Management Architecture." In: *2020 5th International Conference on Computer Science and Engineering (UBMK)*. 2020 5th International Conference on Computer Science and Engineering (UBMK). Sept. 2020, pp. 238–242. DOI: 10.1109/UBMK50275.2020.9219374.

[31] Mirko Zichichi, Stefano Ferretti, and Gabriele D'Angelo. "On the Efficiency of Decentralized File Storage for Personal Information Management Systems." In: *2020 IEEE Symposium on Computers and Communications (ISCC)*. 2020 IEEE Symposium on Computers and Communications (ISCC). ISSN: 2642-7389. July 2020, pp. 1–6. DOI: 10.1109/ISCC50000.2020.9219623.

[32] Jannik Blähser, Tim Göller, and Matthias Böhmer. "Thine — Approach for a fault tolerant distributed packet manager based on hypercore protocol." In: *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC). ISSN: 0730-3157. July 2021, pp. 1778–1782. DOI: 10.1109/COMPSAC51774.2021.00266.

[33] Jingqiang Liu et al. "A Data Storage Method Based on Blockchain for Decentralization DNS." In: *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*. 2018 IEEE Third International Conference on Data Science in Cyberspace (DSC). June 2018, pp. 189–196. DOI: 10.1109/DSC.2018.00035.

[34] Karthikeyan Bhargavan et al. "Formal Verification of Smart Contracts: Short Paper." In: *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*. CCS'16: 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna Austria: ACM, Oct. 24, 2016, pp. 91–96. ISBN: 978-1-4503-4574-3. DOI: 10.1145/2993600.2993611. URL: https://dl.acm.org/doi/10.1145/2993600.2993611 (visited on 03/09/2022).

[35] Bernhard Beckert et al. "Formal Specification and Verification of Hyperledger Fabric Chaincode." In: (), p. 5.

[36] Saba Latif, Aniqa Rehman, and Nazir Ahmad Zafar. "Blockchain and IoT Based Formal Model of Smart Waste Management System Using TLA+." In: *2019 International Conference on Frontiers of Information Technology (FIT)*. 2019 International Conference on Frontiers of Information Technology (FIT). ISSN: 2334-3141. Dec. 2019, pp. 304–3045. DOI: 10.1109/FIT47737.2019.00064.

[37] Weifeng Xu and Glenn A. Fink. "Building Executable Secure Design Models for Smart Contracts with Formal Methods." In: *Financial Cryptography and Data Security*. Ed. by Andrea Bracciali et al. Vol. 11599. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2020, pp. 154–169. ISBN: 978-3-030-43724-4 978-3-030-43725-1. DOI: 10.1007/978-3-030-43725-1_12. URL: http://link.springer.com/10.1007/978-3-030-43725-1_12 (visited on 03/29/2022).

[38] B. Meyer. "Applying 'design by contract'." In: *Computer* 25.10 (Oct. 1992). Conference Name: Computer, pp. 40–51. ISSN: 1558-0814. DOI: 10.1109/2.161279.

[39] Mohamed Ahmed Seifeldin Mohamed Elsayed. "Blockchain-based containment of computer worms." Thesis. University of Victoria, Dec. 22, 2020. 114 pp.

[40] John Kolb et al. "Quartz: A Framework for Engineering Secure Smart Contracts." In: (), p. 17.

Appendix A

Apache License Version 2.0

`Apache License`
`Version 2.0, January 2004`
`http://www.apache.org/licenses/`

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction, and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

   "Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

(d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade names, trademarks, service marks, or product names of the Licensor, except as required for reasonable and customary use in describing the origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or agreed to in writing, Licensor provides the Work (and each Contributor provides its Contributions) on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied, including, without limitation, any warranties or conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for determining the appropriateness of using or redistributing the Work and assume any risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory, whether in tort (including negligence), contract, or otherwise, unless required by applicable law (such as deliberate and grossly negligent acts) or agreed to

in writing, shall any Contributor be liable to You for damages, including any direct, indirect, special, incidental, or consequential damages of any character arising as a result of this License or out of the use or inability to use the Work (including but not limited to damages for loss of goodwill, work stoppage, computer failure or malfunction, or any and all other commercial damages or losses), even if such Contributor has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing the Work or Derivative Works thereof, You may choose to offer, and charge a fee for, acceptance of support, warranty, indemnity, or other liability obligations and/or rights consistent with this License. However, in accepting such obligations, You may act only on Your own behalf and on Your sole responsibility, not on behalf of any other Contributor, and only if You agree to indemnify, defend, and hold each Contributor harmless for any liability incurred by, or claims asserted against, such Contributor by reason of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS

APPENDIX: How to apply the Apache License to your work.

```
To apply the Apache License to your work, attach the following
boilerplate notice, with the fields enclosed by brackets "{}"
replaced with your own identifying information. (Don't include
the brackets!)  The text should be enclosed in the appropriate
comment syntax for the file format. We also recommend that a
file or class name and description of purpose be included on the
same "printed page" as the copyright notice for easier
identification within third-party archives.
```

Copyright {yyyy} {name of copyright owner}

Licensed under the Apache License, Version 2.0 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at

```
http://www.apache.org/licenses/LICENSE-2.0
```

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an “AS IS” BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

For DIVA