Fides: Distributed Cyber-Physical Contracts

Lars Creutz
Institute for Software Systems
Trier University of Applied Sciences
Birkenfeld, Germany
Email: l.creutz@umwelt-campus.de

Jens Schneider
Institute for Software Systems
Trier University of Applied Sciences
Birkenfeld, Germany
Email: j.schneider@umwelt-campus.de

Guido Dartmann
Institute for Software Systems
Trier University of Applied Sciences
Birkenfeld, Germany
Email: g.dartmann@umwelt-campus.de

***Abstract*—Current work in the field of smart contracts is primarily aimed at developers and directly connected to an underlying cryptocurrency. Those self-enforcing contracts are suitable for financial applications, but often disregard regular agreements that do not rely on digital money or are difficult to specify in the form of program code. In order to promote social interaction and self-organization for all types of users, we present Fides, a framework for creating contracts based on natural language that focuses on security and privacy. The use of natural language, detached from the actual payment process, allows everyone to create digital contracts inside a decentralized peer-to-peer network without relying on an inefficient Blockchain solution. These agreements are not only intended for interactions between humans, but can also be established between devices by automation.**

I. Introduction

In [1], we introduced the the idea of *Cypher Social Contracts*, which is an alternative protocol to the currently available smart contract systems. In this paper, we introduce *Fides*[1], which is an implementation of a distributed cyber-physical contract system that focuses on social interaction, self-organization and privacy. *Fides* provides an optimized networking stack and includes changes to the previously described protocol to allow a better automation and validation of processed contracts.

We start in Section II, where we review related work and identify problems which reinforce the motivation for creating *Fides*. Then we highlight our contribution and what distinguishes *Fides* from other approaches in Section III. Section IV specifies the core fundamentals of the protocol and the additions that we made while implementing the first public version of our system. In Section V, we discuss the implementation in depth. In the last Section VI, we conclude our work and describe upcoming efforts on how we try to integrate our system in regions without Internet access, in order to be able to participate in digital services in those regions as well.

[1]The project is released under the MIT license and available at: https://gitlab.rlp.net/l.creutz/fides
Fides is described as "The Roman personification of honour in the keeping of word or oath". (Harry Thurston Peck. Harpers Dictionary of Classical Antiquities. New York. Harper and Brothers. 1898), Perseus Digital Library. https://www.perseus.tufts.edu/hopper/text?doc=Perseus:text:1999.04.0062 - Accessed June 2021

II. Related Works

Before we describe our system and its implementation in detail, we review related work and then discuss how *Fides* follows a different approach in Section III. For conceptual approaches related to the idea of *Cypher Social Contracts*, we refer to [1].

False promises are often made, especially with regard to smart contracts. A well-known example, which has already been addressed in [2], is that lawyers will become obsolete because smart contracts cannot be changed and clearly define the contents of an agreement. What smart contract advocates often disregard is the complexity of transitioning legal language to computer code and the open research questions whether these digital agreements can be legally binding [2], [3].

Furthermore, works like [4] point out the risks of regular smart contract systems from the point of view of developers. These include among others high security requirements, code verification and flawed tooling. Works like [5], [6] and [7] implement domain-specific languages, which is quite common in this field, especially if the results aim to be compatible to *Ethereum* [8]. Most of the work is incomplete with respect to the presented domain-specific language [6] or serves only as a proof of concept to introduce the presented language to existing smart contract systems. In addition, further manual effort is required to make the generated contracts usable [5]. While a domain-specific language can make certain guarantees with respect to the interpretation of human-readable code in combination with the resulting machine code, a domain-specific language, as its name implies, covers only one domain. In order to address a broad audience and provide guarantees, for example on legal certainty, a single domain-specific language will most likely not be sufficient. Accordingly, it is to be expected that multiple, independent domain-specific languages will emerge, which can lead to compatibility problems in the cooperation between different domains.

However, regular agreements of users or small service providers are hardly addressed. Small, regional companies for example, often lack both the resources for lawyers and the digitization of their own processes. In addition, the possible loss of money in the event of programming errors can also act as a deterrent when using cryptocurrencies [9]. Another problem is in the retrieval of external data within smart contracts. For a wide range of use cases, for example in

the transfer of corporate processes to smart contracts, data is often required that cannot be placed on a Blockchain. Oracles, which are third party services for providing additional data, are used for this purpose and introduce additional security risks [10]. An attack on the external data providers could lead to monetary loss in the context of smart contracts and digital currencies. Due to the non-alterability of decentralized contracts, an attack on the Oracle would be sufficient to manipulate the execution of a contract, which could result in permanent, irreversible damage.

The authors in [11] present a new smart contract language with the goal of separating the communication aspect from the computation in order to create safer smart contracts. An intermediate language can improve the overall security of smart contracts, especially when formal verification can be applied [12]. However, solutions like these lack another abstraction layer, so that the creation of contracts can be simplified. In addition to intermediate languages, other (domain-specific) languages would be needed to address more than just developers as a target audience.

In [13], the authors present another way to implement smart contracts by attempting to enable private information within the contracts. To do so, they separate contracts into two parts: the part which is executed locally and thus can contain private data and the public part which is used to check if the first action was actually executed. The data generated during execution is used in combination with a zero-knowledge proof to create a transaction that can be validated on the network without knowing the private input.

III. Contribution and Concept

The concept of *Cypher Social Contracts* [1] is inspired by the ideas of the Cypherpunk's manifesto [14] in terms of decentralization, privacy of online communication, freedom of speech and the amount of information disclosure necessary to form digital interactions. The complexity of many systems and their focus on developers as an audience makes it difficult for regular users to improve their own privacy. Our contribution is the combination and integration of the following aspects into a novel framework to enable everyone to specify digital agreements and to interact securely through them.

Transparency and natural language: We use a novel concept of transparent contract templates that can be defined with natural language and are therefore completely generic. These allow other users to derive contracts whose contents are specified by the reusable templates. The use of natural language allows users without programming experience to utilize our system. Additionally any template can be revoked, which prevents the creation of new contracts and creates the possibility to reflect time-limited offers. Any template can be understood as an *invitation to treat* [15]. The derived contract represents an offer, which can be accepted or rejected by the creator of the template. In order for a contract to be legally enforceable, the template must be written accordingly in legally safe language. However, this is also the case for any contract in the real world. One advantage of *Fides* is that the tasks can be broken down into individual, understandable tasks that can be reused in other templates. It is still up to legal experts to determine whether the novel contracts inside *Fides* can be used in a legal context. Works like [2] and [3] address the legal aspects of regular smart contracts.

Self-organization and decentralized structure: All interactions are stored in an open, decentralized network in which everyone can actively participate. Due to the transparent nature of the agreements, *Fides* can be used without programming experience and is therefore suitable for non-expert users. Within our first paper [1], we described the network as a gossip peer-to-peer network. In this paper, we propose an optimized network architecture with full nodes and clients, where the full nodes are responsible to document the state of the agreements.

Privacy and security: Every information that is exchanged during the processing of contracts is encrypted by using forward secrecy [16] per contract. The transmitted information is thus only accessible to the two parties involved, although the complete network is used to preserve that data.

Detached from cryptocurrencies: Our contribution is novel in terms of generic digital contract processing in combination with secure private communication, detached from cryptocurrencies and applications with a focus on finance. Each payment process can be described in an abstract way within the agreement, enabling the use of a wide variety of payment systems.

Automation: Fides does not only allow the manual processing of contracts, but also supports automation by developers. Thus, it is possible that the easy-to-understand processes, which may be performed manually, can also be automated and integrated into existing system landscapes. An additional advantage is the possibility to continuously make changes to the underlying implementations and thus to further automate contracts step by step.

Cyber-physical contracts: Our system can be used to create contracts between cyber-physical devices which can be processed continuously after being implemented and automated once. The defined tasks per contract are comprehensible for people who may not understand the implementation.

Decentralized mapping of agreements: Our system differs from traditional smart contract systems like Ethereum [8], in that the network only documents the state of contracts. Any code in the respective system of the participating party is executed locally, which is not only energy-saving, but also completely sufficient for decentralized mapping of agreements. If an untrustworthy party tries to cheat within a contract, the entire interaction can be the basis of a legal dispute. Such an approach allows not having to disclose the underlying systems and business processes by having only abstract tasks within a contract, which are locally integrated into the existing systems. Attacks aimed at monetary losses are more difficult, as payment transactions are initiated or verified by the individual parties and are not tied to a digital currency within the application. Accordingly, zero-knowledge proofs like in [13] are not necessarily required, since the actions performed can be immediately reviewed by the other party.

Energy Efficiency: Due to the local processing of the contract, the system has a lower energy consumption, because the input data to a contract is only readable by the involved parties and does not cause any computation by other network nodes. Additionally, no special hardware is required to participate in the network. We show this in more detail through measurements in Section V-C1.

IV. Definitions

Before we describe the overall structure and implementation of *Fides*, we reiterate the definitions made in [1]. Fig. 1 shows an overview of the components and how they are interrelated.

A. Account

An account inside *Fides* is described as a tuple of private key $\mathcal{K}_{i,S}$ and public key $\mathcal{K}_{i,P}$ on the elliptic curve *Curve25519* [17]. For a user i, the account is defined as:

$$\mathcal{A}_i = (\mathcal{K}_{i,S}, \mathcal{K}_{i,P}) \tag{1}$$

Those keys are used to sign and verify transactions, using the signature system *Ed25519* [18], and intended to be used long-term. Short-term (ephemeral) accounts are used to derive a shared secret when encrypting data during the processing of a contract, using *X25519*[2] [17] Diffie-Hellman key exchange [19], and are defined as:

$$\mathcal{A}_{\mathcal{E}\,i} = (\mathcal{K}_{\mathcal{E}\,i,S}, \mathcal{K}_{\mathcal{E}\,i,P}) \tag{2}$$

We note that any user can create a new set of keys for every contract and therefore minimize the digital trace while using the system.

B. Template

Templates are the base of any contract, define the respective contents of an agreement and are defined as:

$$\mathcal{O}_j = (\mathcal{H}_{\mathcal{O}_j}, \mathcal{K}_{j,P}, B, T, R, r_M, V) \tag{3}$$

Here $\mathcal{H}_{\mathcal{O}_j}$ describes the unique hash of the template and $\mathcal{K}_{j,P}$ the public key of user j who created the template. B is description of the template whereas the other elements of the definition refer to the tasks of a template. In general, a task t is defined as the SHA256 hash of its description $\mathcal{D}$, whereas the description is a regular string object that can be human readable:

$$t_i = \mathcal{H}(\mathcal{D}_i) \tag{4}$$

In a template, the list of tasks T is defined as:

$$T = \{\mathcal{D}_1, \mathcal{D}_2, .., \mathcal{D}_N\} \tag{5}$$

Those tasks are mapped to the responsible user, which is either the creator of the template (the receiver of the contract) or the sender of the contract. In the implementation, we use boolean values, where *false* is mapped to the sender, and *true* mapped to the receiver:

$$R = \{r_1, r_2, ..., r_N\} \tag{6}$$

[2]The naming of the system is based on the proposal of the author of the curve. Here *X25519* refers to the Diffie-Hellman key exchange, *Ed25519* to the signature scheme and *Curve25519* to the underlying elliptic curve used by both: https://mailarchive.ietf.org/arch/msg/cfrg/-9LEdnzVrE5RORux3Oo_oDDRksU/ - Accessed May 2021

The list of tasks is used to calculate a Merkle tree [20] whose root hash is defined as r_M and reused inside the contracts. An addition that *Fides* makes to the protocol is to add *Validators* inside the definition of a template:

$$V = \{v_1, v_2, ..., v_N\} \tag{7}$$

Validators describe checks, defined at the time of the template creation, for the required input data, which are verified during the processing of the contracts by the respective parties. Validators are especially useful on automated contracts which were defined by programming instead of using the application through the command line interface. The currently available validators are:

- *Plaintext:* No validation, used by a regular user without programming.
- *Range:* Validate that a given value is inside a previously defined range for the given data type.
- *Regex:* Validate if the input data matches a given regular expression.
- *Signature:* Verify if the input data was signed using a known public key on the same elliptic curve.
- *Multiple signature:* Validate that the given data was signed by multiple separate keys.
- *SHA256:* Verify that the hash of the input data matches the given value.

C. Contracts

A contract $\mathcal{C}_{i,j}$ is always an agreement between two parties i and j and derived from a valid template $\mathcal{O}_j$. Therefore a contract must include a reference (the hash value $\mathcal{H}_{\mathcal{O}\,j}$) to the used template inside its definition:

$$\mathcal{C}_{i,j} = (\mathcal{H}_{\mathcal{C}_{i,j}}, \mathcal{K}_{i,P}, \mathcal{K}_{j,P}, N, R, r_M, \mathcal{H}_{\mathcal{O}\,j}) \tag{8}$$

$\mathcal{H}_{\mathcal{C}_{i,j}}$ represents the unique hash value of the contract. Besides the public keys of the involved parties, a contract contains a reference to the number of tasks N, the list of who is responsible for which task R and a reference of the calculated Merkle tree root hash r_M. The reuse of these objects requires less memory and by using Merkle trees it is not necessary to include the actual tasks from the template in the contract. Both templates and contracts also contain a 4 byte nonce which should ensure the uniqueness of the respective objects inside our implementation. Every contract is in a state, which is defined by the number of interactions with it:

- *init:* Contract was created and published.
- *rejected/live:* The other party (owner of the template) rejected/accepted the contract.
- *finished:* All tasks have been fulfilled.

D. Transactions

A transaction of user i is defined as:

$$\mathcal{T}_i = (\mathcal{H}_{T_i}, \mathcal{S}_i, \mathcal{K}_{i,P}, type, \mathcal{H}_{T_l}) \tag{9}$$

Each transaction, which is identified by its hash value $\mathcal{H}_{T_i}$, must be signed using the private key $\mathcal{K}_{i,S}$ of the currently responsible account. This signature $\mathcal{S}_i$ can be verified using the public key $\mathcal{K}_{i,P}$ included in the transaction. Additionally, a reference to a previous transaction $\mathcal{T}_l$ (the hash value $\mathcal{H}_{T_l}$)

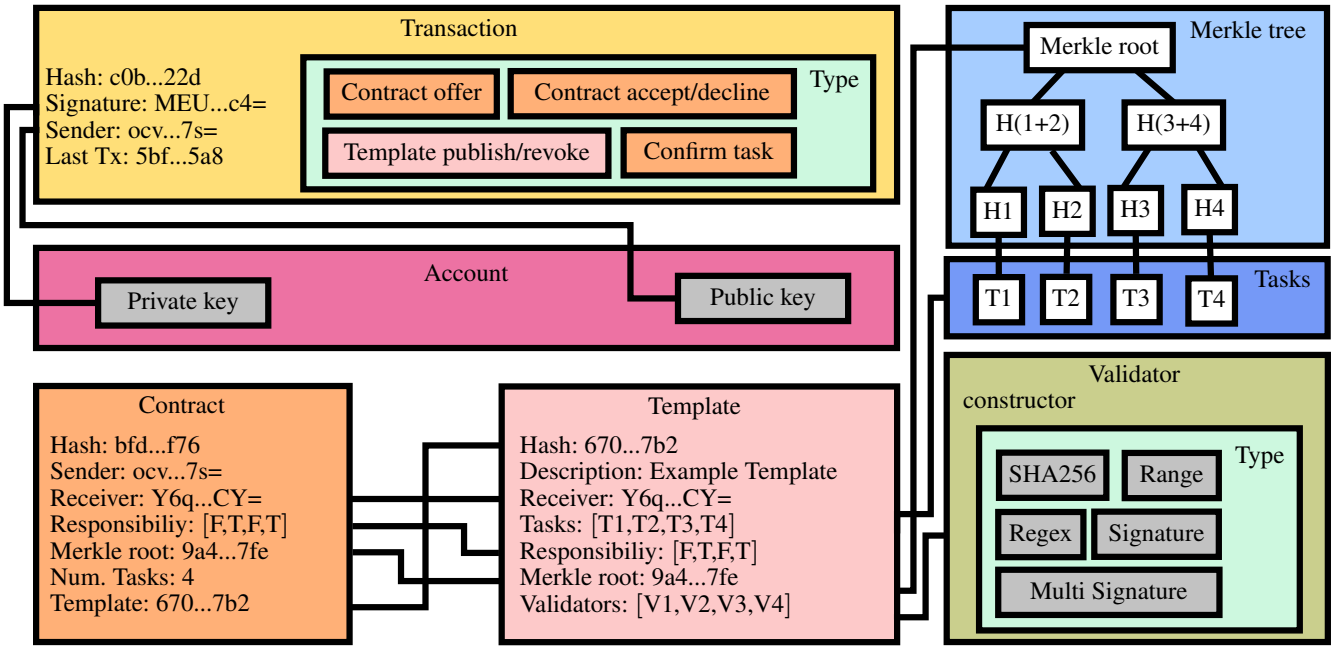

Fig. 1. Overview of the *Cypher Social Contract* system. The figure shows the relationship between the introduced definitions. Some aspects, such as the type of transaction, have been shortened for better readability and are discussed in more detail in Sections IV and V.

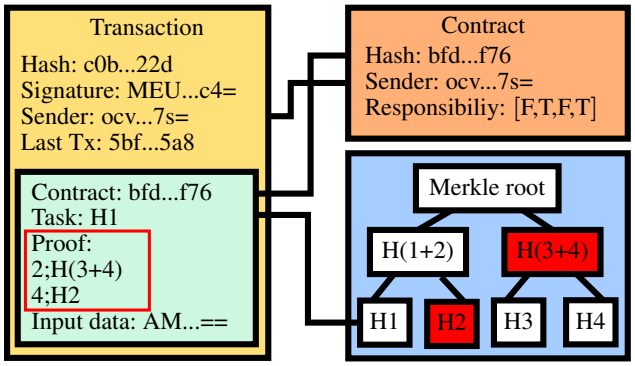

Fig. 2. An example of a simplified transaction that contains the necessary data to confirm a task in an active contract. The transaction addresses the previously created contract and the hash value of the task to be confirmed. Additionally included is the proof in the form of array elements and their position with respect to the full binary tree. Before any input is encrypted and added to the transaction, the validator assigned to that task will check the unencrypted input data.

is included on every type of transaction except when it comes to the publication of a template. Transactions are the only way to change the state of the contracts or templates and always originate from a user. Transactions can be of the following different types that add different elements to (9):

- *Template publish:* Publish a new template $\mathcal{O}_j$ which can be used to derive contracts from.
- *Template revoke:* Set an active template to inactive what prohibits the creation of new contracts. The template is addressed by its hash value $\mathcal{H}_{\mathcal{O}_j}$.
- *Contract offer:* An active template was used to create a new contract $\mathcal{C}_{i,j}$. Now the owner of the template can decide whether to accept or decline the offer. The transaction also includes an ephemeral public key $\mathcal{K}_{\mathcal{E}i,P}$, which is later used to encrypt exchanged data.
- *Contract accept/decline:* Accept/decline a contract identified by its hash value $\mathcal{H}_{\mathcal{C}_{i,j}}$. Provided the contract is accepted, the transaction additionally contains the other temporary public key $\mathcal{K}_{\mathcal{E}j,P}$.
- *Confirm task:* Confirm a task of a live contract. This transaction must include the necessary information about that task and a proof to show the network that this task is part of the contract. For this purpose we use the included proof in combination with the Merkle tree root hash r_M of the contract to check if the task is indeed part of that tree. It is important to note that this check can be performed without accessing the template (due to the reused elements when creating the contract) and therefore we just need to handle $\log_2(N)$ hash values per transaction. Any user input required to confirm that task is encrypted using the exchanged ephemeral keys and added to the transaction. An example of a simplified transaction is shown in Fig. 2.

V. Implementation

Having covered the basic definitions of the protocol, the following section now presents the core components of the implementation.

A. API

The API describes how *Fides* is structured and how it can be used in external programs. In general, the API is divided into the following components:

- *abst:* Abstractions of the system, for example to use *Fides* within a different context. The module exists to create a separation to the core functionality. Currently we are working on a compatible *LoRaWAN* abstraction to be able to create and process contracts without Internet connection.
- *cli:* Implementations regarding the command line interface.
- *core:* The core modules of *Fides*: account management, networking, configuration, encryption, storage, validators, templates, contracts, transactions and serializable types.
- *external:* External modules/libraries separated from the other components.
- *hooks:* Automation for templates or contracts. Hooks can be started from the API or via the command line interface.
- *utils:* Utility functions like logging or time functions used by many parts of the application. Furthermore, this module contains additional functions for assigning aliases for contracts, templates and accounts to simplify the usage of the command line interface.

It should not be the scope of this paper to describe every aspect of the API in detail, therefore Algorithm 1 illustrates some core concepts when using the framework, like how to create and publish a template or use an existing template to create a contract.

Algorithm 1 Brief example on how the API of *Fides* can be used. We show the usage from the perspective of two users. The first user Bob creates and publishes a template, which is then used by Alice to create a new contract.

```
from fides.core.config import ACCOUNT_PATH
from fides.core.account import load_key
```

```
from fides.core.template import Template
from fides.core.types import Validator
t ← Template()
a ← load_key(ACCOUNT_PATH + "bob")
t.tasks = ["First task", ...]
t.responsibility = [True, ...]
t.validators = [Validator(...), ...]
t.description = "My first template"
t.finalize(a)          ▷ Calculated template hash: 582...592
t.publish(a)
```

```
from fides.core.contract import Contract
c ← Contract()
a ← load_key(ACCOUNT_PATH + "alice")
c.create("582...592")                    ▷ Template hash
c.finalize(a)
c.publish(a)
```

B. Command line interface

The system can be used through the command line interface *fds*, which is recommended for regular users. Fig. 3 shows a brief overview of the available commands within the application, which are aligned with the API. In order to create or automate applications in the form of templates and contracts, experienced developers should use the API directly and import the module into their application accordingly. The API and *fds* do not interfere with each other and can therefore be used in parallel, for example to manually check the current state of a contract while other activities are automated in the background.

```
thinkpad :: ~ $ fds
Usage: fds [OPTIONS] COMMAND [ARGS]...

Options:
  --version  Show the version and exit.
  --help     Show this message and exit.

Commands:
  account   Generate, manage accounts
  contract  Create and interact with contracts
  endpoint  Manage connection to other nodes
  hooks     Manage hooks
  init      Initialize Fides
  log       Show entire log
  lora      Manage LoRaWAN abstraction
  network   Manage networking daemon [fidesd]
  status    Show status information
  template  Create and manage templates
  tx        Infos about transactions
thinkpad :: ~ $
```

Fig. 3. Command line interface *fds*

C. Network

Within our first paper [1], we described the network as a gossip peer-to-peer network which was sufficient to test the core functionality of the system. However, in order to provide a system for the general public, some major changes were made to the network architecture to reduce any potential attacks on the system. One common problem inside a gossip network is the presence of spam. Since our system is based on trust and the templates or contracts are defined solely in natural language, it is not possible to reduce spam. Therefore, our approach is not to prevent spam, but to make targeted spam of individual network participants more difficult and not profitable. In order to describe the network architecture of *Fides*, a distinction must first be made between *clients* and *full nodes*. The type of network participant is defined via the global configuration file which is created when initializing the system using *fds*. The same configuration contains the information about which network to join. Each network is precisely defined by its name, which means that semantically different networks can coexist with the same version of *Fides*, as well as that the users' data is not mixed within these networks, provided that the configuration remains the same for all users.

1) Full nodes: Full nodes are users or institutions that form the network by storing and verifying templates and contracts. In order to participate in the network, they must open the corresponding TCP port so that clients are able to communicate with them. The purpose of full nodes is

TABLE I
TEMPLATE INDEX $\mathcal{I}_T$

Table	Contents	Description
Metadata	Template, state, owner, transaction[3]	Necessary information about a specific template
Templates	Template hash, contract hash, contract transaction	Connection of contract and used template
Transactions	Hash, data	Transaction data referenced by the other tables

TABLE II
CONTRACT INDEX $\mathcal{I}_C$

Table	Contents	Description
Contracts	Contract hash, state, last_tx, transaction[4], current_task	All the necessary information to manage the state of a contract
Transactions	Hash, data	Transaction data that interacted with a particular contract

to provide a distributed index that contains objects such as templates or contracts.

In order to implement the remote procedure call (RPC) methods we use *gRPC*[5]. We distinguish between template index $\mathcal{I}_T$ and contract index $\mathcal{I}_C$ which were both realized by using a *SQLite*[6] database. The template index maps the state of active templates and the corresponding offers of users for the specific template. Table I describes the contents of the index. The template index is updated by users of a specific template or by the owner when accepting or declining new offers. A matching contract index $\mathcal{I}_C$ is created for each published contract. In accordance Table II shows the contents of a contract index. To implement a decentralized network architecture with a distributed hash table like lookup, we implemented a modified version of the *Chord* protocol [21]. The basic concept of *Chord* describes an efficient, distributed and generic peer-to-peer lookup service. Therefore a number of nodes are connected within a ring structure where each node knows its predecessor, successor and has a routing table, called the finger table. Within this structure it is possible to store key-value pairs on nodes, depending on their node id, and perform lookups with high probability in $O(\log N)$ time [21], by addressing its direct neighbours or by using the finger table. Due to continuously stabilizing and updating the finger tables, this system design allows to keep the ring structure intact when nodes join or leave the network. Splitting between two types of indices offers the advantage of referencing a contract in two places within the network when it is created: On the template index and on its own separate contract index, which contributes to the reduction of targeted spam. For example, if a full node manages a template, a malicious party cannot spam that node with targeted data, because newly created contracts are distributed uniformly across the network to the number of

[3]Hash of the transaction that published the template
[4]Hash of the transaction that published the contract offer
[5]https://grpc.io/ - Accessed May 2021
[6]https://sqlite.org/ - Accessed July 2021

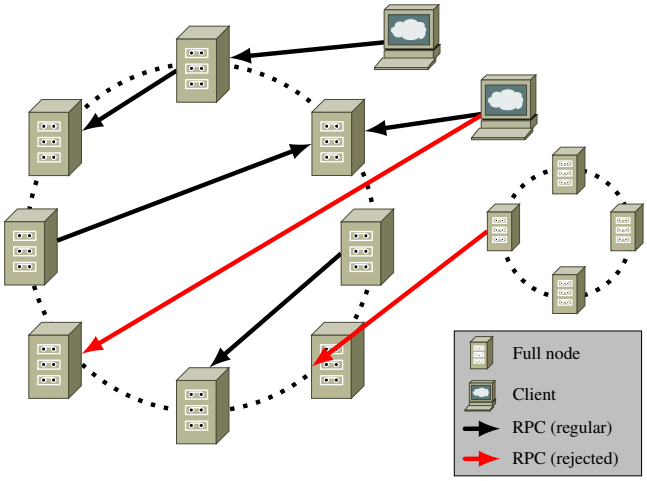

Fig. 4. Network architecture of *Fides*. The isolation prevents other full nodes within a different network of disturbing the ring structure. It is shown that full nodes are allowed to perform RPC calls when they are part of the network (inside the ring). Clients are only allowed to perform a subset of RPC calls on full nodes (outside the ring).

full nodes. For the use in *Fides* we have added some additional features to our *Chord* implementation, like the validation of the caller that invokes the function. We check if the origin of the call is inside the same network by computing the id of the node to verify that only other full nodes are able to alter the structure of the network:

$$id = \textbf{SHA256}(network \oplus ip \oplus port) \tag{10}$$

Here $\oplus$ denotes the concatenation. The other full nodes must accept requests on that public ip:port combination and state to use the same network as the full node validating the call. *Fides* extends the implementation by adding more remote procedure calls that can be used by clients to update the index, creating an isolation level between full nodes and clients that is illustrated in Fig. 4. Clients do not need to know the current ring structure to submit their transactions. This supports the privacy of the users, since with each transmission another node within the ring can be addressed. Within the implementation, a randomly chosen endpoint in the same network is selected. Full nodes take care of the assignment of the transaction to the responsible node within the ring. To do so, they first check whether they themselves are responsible for the request by comparing the distance of their node id to the hash of the addressed object, which are both SHA256 hash values. If another node is responsible, the complete request is forwarded using the routing table of the protocol. Thus, the receiving node knows only the information from the other full node, but not the exact origin of the transaction.[7] An important point to stabilize the network is the handling of the respective indices when a full node leaves the network. Thus, if the application terminates

[7]We note that the information could still be used to draw conclusions about the actual origin of the transaction. The timestamp (if not manipulated) can provide hints about when the transaction was sent. In addition, the type of the node can indicate whether the transaction, if it is sent directly to the responsible full node without being forwarded, is from a client, since clients do not allow requests on the public port.

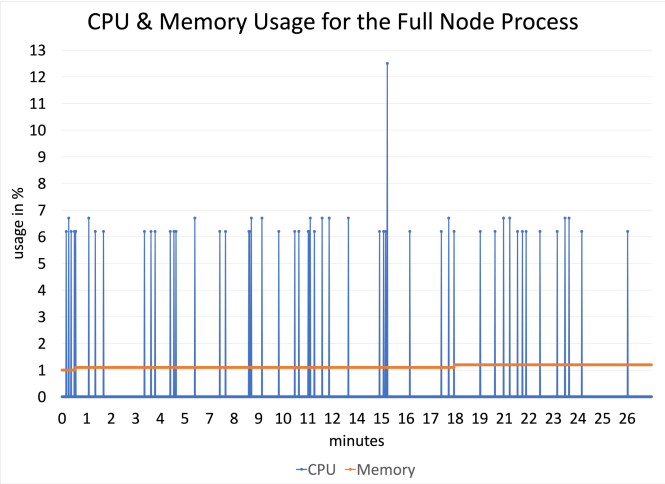

Fig. 5. CPU and Memory usage on a full node. The average memory usage is constant at about 1.1%, even with different transaction sizes during the measurement. The average CPU usage is 0.2%.

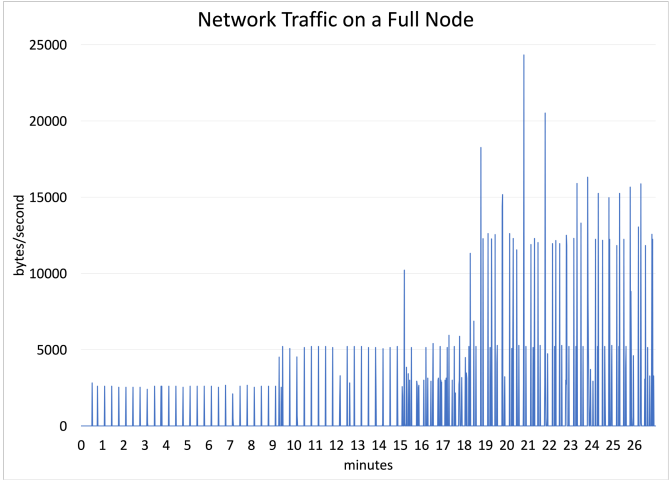

Fig. 6. Network traffic captured on the default port at a full node. The regular background update intervals, as well as individual transactions that contain more data, are shown.

correctly, the contents of the index must be passed to the immediate successor of the node [21]. For this purpose, the current index node sends a request to its immediate successor. The request contains the wish to transfer an object, which is identified by its hash and type. The requested node checks the correctness of the request by first validating the information about the node as described in (10) afterwards making sure that the requesting node is indeed the immediate predecessor. After all checks are successful, the new index node requests the objects to be transferred from its predecessor and creates the same index. We have chosen that procedure and transfer the index only when nodes leave, not when a new nodes joins. This is based on the assumption that index nodes are usually executed on continuously accessible systems (e.g. servers). Furthermore, the state of the network is restored by the users in all cases anyway (see Table IV), which is also used if a full node is abruptly unavailable and thus has not initiated the transfer (e.g. due to a power failure). Nevertheless, at this point we would like to describe a possible attack on the system that is related to faulty clients. To the extent that a full node documents the state of a contract and cannot transmit the index in the event of an error, an untrusted party can manipulate the processing of the contract by deleting the last transaction (locally) and sending a new transaction with different data. The index node, which at that point does not know the contract (due to failure), will build that index from scratch and assume that the transactions are correct. Such an action would be immediately noticed by the other party and expose the attempted fraud, because the malicious transaction cannot be applied to the other (correct) local state. To conclude our description of full nodes, we would like to discuss the requirements to the underlying hardware. Since *Fides* is intended to promote self-organization, especially for structurally weak regions, one condition for the application is that it must also be functional on non-powerful, cheap hardware. Within our experiments we

TABLE III
DETAILED INFORMATION ABOUT THE USED PROCESSOR WITHIN THE SIMULATED HARDWARE

Model name	Intel(R) Xeon(R) CPU E5-2640 v3
Sockets	1
Cores per socket	2
Threads per core	1
CPU MHz	2600
L1d cache	32K
L1i cache	32K
L2 cache	256K
L3 cache	20480K

have used a virtual server with a simulated dual core processor and 4GB of memory. Table III shows the more detailed specifications of the processor and emphasizes that a similar system can be operated cheaply both self-hosted and within a Cloud environment. The information was retrieved using the *lscpu*[8] command. Fig. 5 shows one example measurement that contains the transactions of multiple users during a timespan of about 30 minutes. Here, the utilization of memory is about constant at 1.1%, while the CPU usage is 0.2% on average. A spike[9] can be noticed from time to time, but the overall utilization of the system by the full node process remains below 13 percent. The measurements were performed in one-second intervals using the *top*[10] application on the virtual server. In addition, the measurement in Fig. 6 shows the network traffic at the same time period. The graph shows the automatic update intervals of the currently active users, between which manual updates were still performed from time to time. Spikes in the graph show the publishing of transactions with contents, for example the publishing of templates or transactions during

[8]https://man7.org/linux/man-pages/man1/lscpu.1.html- Accessed October 2021

[9]The spike at minute 15 was caused by a transaction that included encrypted input data to the confirmation of a task. However, later similar transactions of about the same size or bigger did not cause a higher CPU usage.

[10]https://man7.org/linux/man-pages/man1/top.1.html - Accessed May 2021

TABLE IV
ALIGNMENT OF THE LOCAL INFORMATION OF AN OBJECT WITH THE CURRENTLY AVAILABLE INFORMATION OF THE INDEX

Local state	Index state	Action
any	no state at index	Republish entire object
behind	advanced	Get transaction(s) and apply to local object
advanced	behind	Update the index with the missing transaction(s)

the processing of contracts. This measurement was performed using *tcpdump*[11]. The figures are intended to illustrate that the load on the system is small, even if the number and size of transactions increase over time.

2) Clients: Clients describe regular users who have decided not to contribute to the functioning of the network and only want to manage their templates and contracts via *Fides*. Users connect to the network using their local database to locate at least one full node inside the same network. New nodes are not automatically added, which makes it possible to use private instances of the network, for example within a corporate infrastructure. If the full nodes contained therein are located behind a firewall and are thus not accessible from the public Internet, any number of clients or full nodes from the same network can access the system's functions without inadvertently making information publicly available. Furthermore, it is still possible to add new nodes, for example through the command line interface, by traversing the entire network or manually adding new endpoints. In general, when regular clients start the network, for example through *fds*, they need to check whether the local state of their contracts/templates is still valid and equal to the state of the network. Therefore they ask the full nodes to obtain the current version of the index of the object to verify and apply the action described in Table IV. This approach allows both parties to restore their local state in the event of a failure of the full node. Accordingly, over the period of the synchronization intervals, the same view is obtained in the network. For example, if one party restores an old state, the other party updates it again until the correct state is recovered. Fig. 7 shows the complete interaction from creating a template to processing a contract between two clients within the same network. Since *Fides* should not only be cheap to run as full node, but should also work for regular users on low-end hardware, all functions of the system work flawlessly on Raspberry Pi computers from version 3 and higher. Furthermore, the Raspberry Pi also serves as a hardware platform on which the previously mentioned *LoRaWAN* abstraction was implemented.

D. Encryption

Because *Fides* has a special focus on privacy and anonymity of the users, we use forward secrecy [16] to securely transmit sensitive information. The transparent nature of our template definition allows any user to see which information needs to be shared before interacting with the other party. In order to transfer this information securely, a separate account is generated for each contract (2). Generally, the regular account is used to access the temporary account, which is stored encrypted and is only readable by the current user due to the file system access restriction. Referring again to the definitions introduced earlier, the transactions for exchanging temporary keys are as follows:

$$\mathcal{T}_j = (\mathcal{H}_{T_j}, \mathcal{S}_j, \mathcal{K}_{j,P}, \mathcal{O}_j) \tag{11}$$

$$\mathcal{T}_i = (\mathcal{H}_{T_i}, \mathcal{S}_i, \mathcal{K}_{i,P}, \mathcal{C}_{i,j}, \mathcal{K}_{\mathcal{E}\,i,P}) \tag{12}$$

$$\mathcal{T}_j = (\mathcal{H}_{T_j}, \mathcal{S}_j, \mathcal{K}_{j,P}, \mathcal{H}_{\mathcal{C}_{i,j}}, \mathcal{K}_{\mathcal{E}\,j,P}) \tag{13}$$

The first transaction (11) publishes the template which is then used by i who includes the created contract and an ephemeral public key $\mathcal{K}_{\mathcal{E}\,i,P}$ (12). When accepting the contract (13), the other party references it by its hash value and also attaches its temporary public key $\mathcal{K}_{\mathcal{E}\,j,P}$. We first use the described method to exchange ephemeral keys using Elliptic Curve Diffie-Hellman (ECDH) [19] which are used when a task of the contract is confirmed. The hash of the last transaction $\mathcal{H}_{T_l}$ is used as input parameter to the HMAC-based Key Derivation Function (HKDF) [22]. The obtained secret is then used to symmetrically encrypt the input data using AES-256 [23]. To comply with the chosen AES mode CBC [24], we use Public Key Cryptography Standards #7 (PKCS7) [25] to add padding to the input data. The returned initialization vector and ciphertext are then added to the transaction.

E. Communication

Having covered the fundamental aspects of the implementation in the previous paragraphs, we will now describe the local communication of the system in order to illustrate the overall system. The majority of communication is handled by a daemon process (*fidesd*), which is usually started using the command line interface. The daemon process manages the following components:

- *RPC-Server*: The entry point for any regular communication with the system. Here, the server listens only to local connections and implements a RPC method, which receives already finalized transactions that are then transmitted to the network. That method is called by the core objects such as contracts or templates (see Algorithm 1). In order to implement the RPC methods, we rely on the same library as on the full nodes, which has the advantage that any message size limitations[12] or errors already occur locally. If a transaction raises an error during the local communication, it cannot be transferred to the network. If an erroneous transaction does arrive at a full node, it can be concluded that it is a deliberately wrong-acting party or an attacker, which makes it possible to exclude that party from using the system. In addition to the method for transmitting transactions, the RPC server also implements functions

[11]https://man7.org/linux/man-pages/man1/tcpdump.1.html - Accessed May 2021

[12]The default gRPC message limit in python is 4MB. Fides limits the transaction size to 1MB since release 0.1.4.

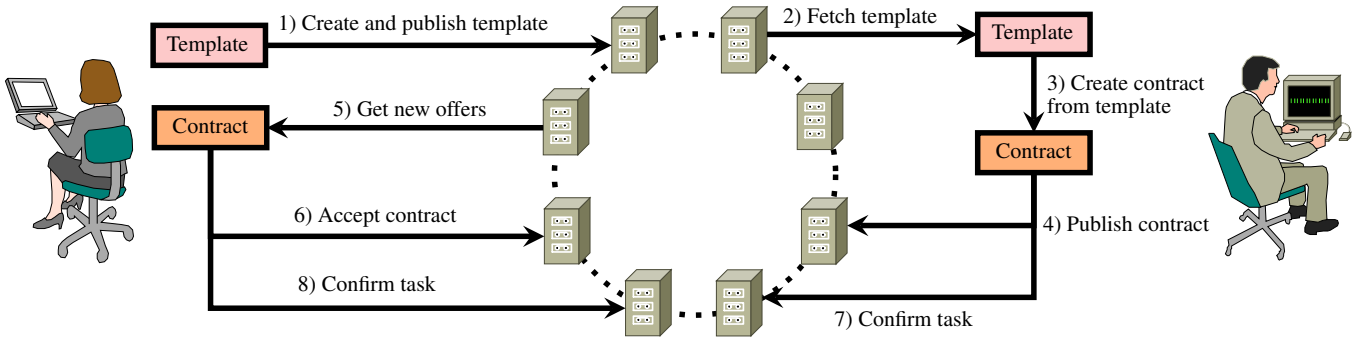

Fig. 7. General communication flow using the example of two clients. The person shown on the left creates a template, which is imported by the other person to create a contract from it. The processing takes place on the local objects, whose state is monitored by the distributed index. For simplicity, the arrows are used for transactions as well as for general requests to the network. The parties always use different full nodes to transmit their transaction to the network.

to safely stop the network daemon and collect status information that is used by the command line interface.

- *Publisher*: This component forwards the received transactions to the network. Currently, there is only a single (regular) publisher, which is used to transmit the transactions. In general, the component is derived from an interface, which allows to develop further abstractions that interact with the network in a different way.
- *Chord-Server*: For full nodes *fidesd* additionally manages incoming connections. Therefore the daemon starts another RPC server that implements the functionality described in Section V-C1. The exact functionality depends on the configuration, so it is also possible to create private subnets with full nodes or to test applications locally without participating in a public network.

In order to improve the usability, some functions of the command line interface use the publishing interface directly, for example to be able to check the state of a contract manually without starting the entire communication daemon. However, information is only obtained in this way, not sent.

F. Automation

The features described so far primarily addressed regular users and described the underlying architecture of *Fides*. In order to additionally address developers, the module *Hooks* will be described in the following, which allows the dynamic automation of templates and contracts by observing the current state of the system. In general, a *Hook* is bound to an object (contract or template) and contains the following components:

1) *Callback*: Function to be called when the condition of the *Hook* is activated. The callback is a method inside an arbitrary *Python* script on the same system.
2) *Arguments*: Additional arguments of the user that will be passed to the callback.
3) *Called*: A simple flag to indicate whether the callback has already been called.
4) *Live Forever*: Another flag to declare that the *Hook* should continue to run after the first invocation of the callback.
5) *Interval*: The interval in which the condition of the *Hook* is monitored.

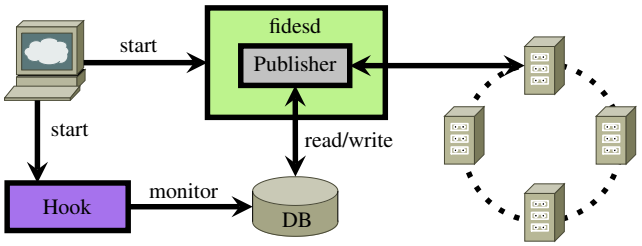

Fig. 8. Each Hook is independent of the core system and monitors the state of the database separately. The networking deamon is responsible for the communication with the network, whereby incoming or outgoing transactions change the state of the objects, causing the associated Hook to execute the contained callback.

TABLE V
CAPABILITIES TO AUTOMATE PROCESSES WITHIN *Fides* THROUGH THE *Hooks* MODULE.

Hook	Description
Contract accepted	Exclusively run the callback if a contract has been accepted. Usually used by the sender of a contract.
Contract rejected	Execute the callback bound to a contract if that contract was declined by the owner of a template.
Contract changed	React to any state change. Recommended if that contract is entirely automated.
Contract complete	Monitor a finished contract. Can be used to automate the export of the contract data and cleanup the used ephemeral keys.
Template used	Run the defined callback if a template has been used. Important for template owners to automate the processing of new contract offers.

The module accesses the local databases and monitors whether changes have been made. This happens, for example, when the state of a contract changes either actively by sending a transaction, or passively by updating the index. Fig. 8 gives an overview of how the module is integrated into *Fides*. Table V describes the available types of *Hooks* and briefly explains their field of application. Any *Hook* can be started either dynamically within a program or after an export by the command line interface.

VI. Conclusion and future work

The paper presented *Fides*, a framework for creating, processing and automating contracts based on natural language with a strong focus on privacy.

We started by formally introducing the underlying idea of the *Cypher Social Contracts* protocol with additional features, which were created as a result of implementing *Fides*.

Next, we addressed the implementation by describing both the API and the command line interface and went into detail about the distributed network architecture for the communication of the network participants. Besides the local communication, we also discussed the encryption used by the parties when exchanging data. Furthermore we presented measurements that showed that the application can be used even on low-power hardware. This supports the core idea of the system, namely self-organization, since no special hardware is needed to participate in the network. Additionally, in order to address potential developers, we presented ways to automate the processing of templates and contracts.

Our next steps are to introduce the system to the general public, especially in rural areas. For this purpose, we created an abstraction that enables contract participation via *LoRaWAN* using a Raspberry Pi, so that regions without Internet access can also participate in digital services. Therefore we implemented a protocol to translate *LoRaWAN* payloads to *Fides* transactions, which is used in combination with a secure middleware service. Both the abstraction and the specification of the middleware will be available in a later release of the software.

Furthermore, we continuously extend the functionality of the released framework and improve the usability in order to address a broader audience.

Acknowledgment

This work has been funded by the Federal Ministry of Transport and Digital Infrastructure (BMVI) within the research initiative "mFUND" under the grant number 19F2102F.

References

[1] L. Creutz and G. Dartmann, "Cypher social contracts a novel protocol specification for cyber physical smart contracts," in *2020 International Conferences on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*, 2020, pp. 440–447.

[2] E. Mik, "Smart contracts: terminology, technical limitations and real world complexity," *Law, Innovation and Technology*, vol. 9, no. 2, pp. 269–300, 2017. [Online]. Available: https://doi.org/10.1080/17579961.2017.1378468

[3] E. Gyr, *Blockchain und Smart Contracts: die vertragsrechtlichen Implikationen einer neuen Technologie*. Rechtswissenschaftliche Fakultät der Universität Bern, 2019.

[4] W. Zou, D. Lo, P. S. Kochhar, X. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[5] C. K. Frantz and M. Nowostawski, "From institutions to code: Towards automated generation of smart contracts," in *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*, 2016, pp. 210–215.

[6] E. Regnath and S. Steinhorst, "Smaconat: Smart contracts in natural language," in *2018 Forum on Specification Design Languages (FDL)*, 2018, pp. 5–16.

[7] M. Wöhrer and U. Zdun, "Domain specific language for smart contract development," in *2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 2020, pp. 1–9.

[8] V. Buterin *et al.*, "Ethereum white paper," [Online]. Available: https://github.com/ethereum/wiki/wiki/White-Paper (Accessed June 2021), 2013.

[9] K. Krombholz, A. Judmayer, M. Gusenbauer, and E. Weippl, "The other side of the coin: User experiences with bitcoin security and privacy," in *International conference on financial cryptography and data security*. Springer, 2016, pp. 555–580.

[10] A. Egberts, "The oracle problem - an analysis of how blockchain oracles undermine the advantages of decentralized ledger systems," 2017.

[11] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao, "Safer smart contract programming with scilla," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360611

[12] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security*, ser. PLAS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 91–96. [Online]. Available: https://doi.org/10.1145/2993600.2993611

[13] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contracts platform," 2017.

[14] E. Hughes, "A Cypherpunk's Manifesto," [Online]. Available: http://www.activism.net/cypherpunk/manifesto.html (Accessed June 2021), 1993.

[15] A. Burrows, *A Casebook on Contract*. Bloomsbury Publishing, 2018.

[16] W. Diffie, P. C. V. Oorschot, and M. J. Wiener, "Authentication and Authenticated Key Exchanges," 1992.

[17] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *Public Key Cryptography - PKC 2006*, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228.

[18] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, vol. 2, no. 2, pp. 77–89, Sep 2012. [Online]. Available: https://doi.org/10.1007/s13389-012-0027-1

[19] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theory*, vol. 22, pp. 644–654, 1976.

[20] R. Merkle, "Protocols for Public Key Cryptosystems," 04 1980, pp. 122–134.

[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, ser. SIGCOMM '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 149–160. [Online]. Available: https://doi.org/10.1145/383059.383071

[22] H. Krawczyk, "Cryptographic extraction and key derivation: The hkdf scheme," in *Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference*, ser. Lecture Notes in Computer Science, vol. 6223. Springer, 2010, pp. 631–648. [Online]. Available: https://www.iacr.org/archive/crypto2010/62230625/62230625.pdf

[23] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, and J. Dray, "Advanced encryption standard (aes)," 2001-11-26 2001.

[24] W. F. Ehrsam, C. H. W. Meyer, L. Smith, and W. L. Tuchman, "Message verification and transmission error detection by block chaining," 4 1976, US Patent US4074066A. [Online]. Available: https://patents.google.com/patent/US4074066A/en

[25] B. Kaliski, "Pkcs #7: Cryptographic message syntax version 1.5," Internet Requests for Comments, RFC Editor, RFC 2315, March 1998, [Online]. Available: http://www.rfc-editor.org/rfc/rfc2315.txt (Accessed July 2021).