Generating and Adjudicating Digital Legal Agreements Using Ethereum Smart Contracts

by

Kevin Y. Liu

S.B., Mathematics, Massachusetts Institute of Technology (2018)

S.B., Computer Science and Engineering, Massachusetts Institute of Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Author .....

Department of Electrical Engineering and Computer Science

May 24, 2019

Certified by .....

Lalana Kagal

Principal Research Scientist, Computer Science and Artificial Intelligence Lab

Thesis Supervisor

May 24, 2019

Accepted by .....

Katrina LaCurts

Chair, Master of Engineering Thesis Committee

May 24, 2019

Generating and Adjudicating Digital Legal Agreements Using Ethereum Smart Contracts

by

Kevin Y. Liu

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2019, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Across the world, countless legal agreements are drafted, signed, and disputed every day. In most cases, the drafting process consists of costly and lengthy revisions of paper-based legal agreements. Furthermore, once finalized and signed, these legal agreements are hard to track, manage, and arbitrate, as there is no standard system for storing and sharing paper-based legal agreements. Regardless of their purpose, these legal documents must be carefully managed and approved by teams of lawyers for all parties listed in the agreement, resulting in lengthy face-to-face meetings, and in the case of breaches, in-person disputes over resolutions.

In this thesis, we designed and implemented a novel blockchain-based alternative for generating, tracking, managing, and adjudicating legal agreements using Ethereum smart contracts. Through a front-end web-app, users can piece together common contract clauses with custom parameters to generate their own natural language legal agreements. Our interface then automatically creates equivalent digital smart contracts representing these parameters and clauses, deploying the smart contracts as permanent records onto the Ethereum blockchain. These smart contracts are formally verified with various techniques to ensure that they reflect the intents of the drafted contract and are free from execution vulnerabilities. To hold violators accountable, breaches by any signatory of the contracts can then be arbitrated digitally using secure voting by external arbiters. These violators can then be subject to monetary penalties paid in digital Ether tokens for breaching the agreement.

Thesis Supervisor: Lalana Kagal
Title: Principal Research Scientist, Computer Science and Artificial Intelligence Lab

Acknowledgments

I would like to thank my advisor Lalana Kagal for giving me the opportunity to work on this project for my Master of Engineering thesis. The outcome has been the result of a two-year collaborative process, starting in 2017 when Lalana advised my Advanced Undergraduate Research Opportunities Program (SuperUROP) project. Working on this project has been an incredibly enriching experience, and I am very thankful for Lalana's guidance, intellectual support, and patience.

I would also like to thank Ravi Rahman, the undergraduate researcher working on his own SuperUROP project in parallel with my thesis. His contributions to the Python language representation and the Z3- and KEVM-verification of the legal contracts were crucial to the success of this project. I really enjoyed the late night work sessions we held, iterating over the exact specifications needed to represent a legal contract in Python. I really appreciate Ravi's reliability, dedication, and willingness to voice his ideas in guiding the direction of the project.

Furthermore, I would like to thank Harsh Desai and Murat Kantarcioglu of the University of Texas at Dallas's Data Security and Privacy Lab. I had previously collaborated with them for my SuperUROP project, and they remained on board to provide feedback and guidance for my thesis project this past year. I really appreciate the time and encouragement that they offered me, as well as their wholehearted interest in seeing where this project would lead.

Lastly, I would like to thank my family and friends for their endless support. I could not have made it this far without them, and for that, I am eternally grateful.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In today's era of inter-connectivity, collaboration, and dependence, there is almost always a written legal basis for any interaction that occurs between individuals, companies, or entities. This might be a terms-of-service agreement, a payment for services contract, a trade record of financial securities, or an employment contract. These agreements are almost always filled with legal jargon and must be drafted by teams of lawyers over the course of many paper-based iterations. This process is very cumbersome and costly, and there may be unnecessary time delays related to the transfer of written paperwork between parties.

The above problems are only those that occur during the legal drafting process, before any party has even agreed to the terms. Once signed, multiple copies of these paper-based legal agreements are stored independently by each party. These copies are difficult to keep track of, and individual clauses of these contracts are difficult to look up and reference later. Furthermore, in the case of a suspected contract breach, lawyers must reevaluate the contract text to determine whether or not the breach actually occurred, and if so, what the consequences should be.

Given the difficulties with the current state of writing, managing, and adjudicating legal contracts, we propose a modern, digital alternative for generating, tracking, managing, and especially resolving breaches with legal contracts. This alternative will hopefully ease logistical problems related to the paper-drafting of legal contracts, as well as store all digitized contracts as permanent records that are immutable and

easily auditable.

1.1 Scope

The scope of all legal agreements is simply too large for a single thesis, so we chose to focus on legal data sharing contracts when evaluating our system. The choice of focusing on data sharing contracts was influenced by my previous work in the healthcare space during my Advanced Research Opportunities Program (SuperUROP) project, as well as the significantly increased need for security, privacy, and adjudication for these types of contracts. However, it is important to note that the system we developed can be harnessed to draft and model any type of legal contract as a digital smart contract on the Ethereum blockchain. Some other fields where our developed system can be heavily utilized are the financial space, enterprise software licensing contracts, and homeowner mortgage loans. Section 1.1.1 elaborates on why a digitalized legal contract system would vastly improve the current data sharing space.

1.1.1 Data Sharing Contracts

Every day, more and more data is generated around the world. Regardless of its purpose, for most data the sharing of it becomes paramount to increasing its value. Many applications ranging from personalized health care to crowdsourced traffic reports to pharmaceutical research require individuals and organizations to share data at an unprecedented scale. In healthcare, patient data is shared among institutions for improving treatment decisions [1]. First responders and aid workers share data between them to better coordinate disaster relief efforts [2]. Cities share the data that they collect about commute patterns with different stakeholders such as transportation companies to reduce traffic jams during rush hour [3]. Data sharing is crucial in today's world, but due to privacy reasons, security concerns, and regulation issues, the conditions under which the sharing occurs need to be carefully specified.

In many use cases, a basic data sharing scenario would involve a data requester (e.g. a medical research clinic), a data provider (e.g. a healthcare patient), and

the data itself. Data sharing can be a constructive concept to improve collaboration, analyze and utilize data to generate significant value, and maintain accountability and transparency. At the same time, the misuse of such shared data can create significant problems such as the leak of patient confidentiality and identity theft.

To prevent misuse, reduce liability, and comply with regulations, many organizations sign legal agreements before sharing data. These legal agreements serve to protect the interests of both parties. For example, the data distributor would likely wish to be notified if the data requester's data server is breached, and these conditions could be specified via a data sharing agreement. Likewise, the data requester would want to have some recourse if the data provider shares corrupted or falsified data. Many other restrictions and precautions are introduced in a data sharing contract to ensure that the collaboration between the two parties is as smooth as possible.

The necessity to draft and sign so many legal agreements in the data sharing field is a perfect application of our system. By digitizing all these contracts and maintaining them as permanent records on the blockchain rather than in email threads or file cabinets, users and institutions involved with data sharing can more easily draft, access, verify, and dispute these contracts.

1.2 System Goals

The goals of the system that we developed are two-fold:

First, the system created must be easy to use, without any required knowledge of the back-end Ethereum blockchain technology. There is a steep learning curve and set-up entry barrier with using the blockchain, and thus by eliminating these obstacles, many more users will be able to develop digital contracts using our system. The system will need to be designed in a way to abstract away the back-end technical layers while still maintaining the back-end's intended and provided functionality.

Second, the system created has to be robust in terms of legal contract modeling. If only a few specific contracts can be modeled with our digital system, there will not be any willingness or incentive for adoption of it. It is acceptable if the initial

iterations of the system only support the construction of a few specific data sharing clauses for prototyping purposes. However, the system must be easily extensible and adaptable, so that more functionality and clause support can be added by either system administrators or users of the system.

Whether our developed system accomplishes these two goals will determine its success and usefulness to lawyers and legal contract writers worldwide.

1.3 Contributions and Thesis Overview

In this thesis, we present a framework for lawyers and other users to digitally draft legal contracts. An automatic process then formally verifies and deploys these contracts onto the Ethereum blockchain. After being committed to the blockchain, the contracts are immutable, and can be easily adjudicated in the case of a suspected breach. A key contribution of our system is the creation of a crowd-sourced clause repository to promote clause re-use, saving time and effort in drafting legal contracts. Another contribution is the development of an easy-to-use front-end, abstracting away the blockchain and programming-based back-end so that anyone can use our system. A third contribution is the creation of a Python-based library of classes and functions to model a higher-level code-based representation of legal contracts. Users that are more programming oriented can develop their own contracts directly using this library. A final contribution is the creation of a democratic arbitration process residing digitally on the blockchain for easy dispute resolution.

Chapter 2 provides an overview of the technical background of frameworks and tools used in this thesis. Chapter 3 presents previous literature and related work. Chapters 4 through 7 detail our system's technical pipeline, including using the front-end interface to draft a legal contract, representing the written contract programmatically in Python and Vyper, and deploying the contract to the Ethereum blockchain. Chapter 8 contains an in-depth evaluation of our developed system. Chapter 9 discusses future work that we hope can further improve our system's functionality and usefulness. Chapter 10 concludes the thesis and reiterates our contributions.

Chapter 2

Technical Background

The following sections provide an overview of the various technologies we used to develop our system, as well as any background knowledge related to these technologies that is key to understanding the functionality and robustness of our system.

2.1 Blockchain Technology

First introduced in 2008 by Satoshi Nakamoto, blockchain technology is a means to introduce trust to a distributed and decentralized system. By using peer-to-peer transactions and audits without the need for a central authority or governing body, the goals of blockchain are to decrease costs, increase speeds, and increase security by eliminating potential vulnerabilities that stem from having a central point of trust. In essence, the blockchain can be thought of as a distributed database, except no individual party can alter or tamper with the content of data or time stamps in the log without a majority of other nodes agreeing to the changes. In the case of legal contracts, the restriction of time stamp and log editing is crucial, as each action by every party must be certified and auditable. Should a breach ever be suspected in the future, the permanent transactions committed to the blockchain can be revisited to determine exactly where and when the breach occurred. For new nodes joining the blockchain, there is not a single party they must trust, but rather they only need to trust that the majority of existing nodes in the network are non-malicious [4].

2.1.1 Proof of Work

Most blockchains rely on a process called "Proof-of-Work" (PoW) to maintain the legitimacy of transactions. When two nodes on the network enact a transaction, the transaction is broadcast to all nodes and recorded in a "block." These blocks are the building structures of the system, hence the name blockchain, and each block stores a certain number of bytes of information. Once a block is full, nodes independently perform PoW to commit the block to the permanent blockchain.

PoW involves solving a complex cryptography problem, the simplest example of which is factoring a large integer $n = pq$ into the product of its two primes p and q. These problems require large amounts of computational power to solve but are easily verifiable by other nodes on the network. This discourages attempts to commit fraudulent blocks, as well as deters denial-of-service attacks by rate-limiting the number of transactions that can be committed to the blockchain. Each node's proposed solution to the problem is broadcast to all other nodes, which then check the correctness of the solution. Only when a majority of nodes have approved of the solution is the commitment deemed legitimate and the block permanently stored in the blockchain. Thus, the only way for a malicious party to commit fraudulent transactions is to control 51% of the computational power of the network, which is unfeasible and cost-prohibitive.

The incentive for solving these cryptography problems and committing blocks is that the first node to legitimately commit each block receives a share or token of reward associated with the blockchain. For example, on the Bitcoin blockchain, the first node would receive a payment of one Bitcoin, which has monetary value. This incentive has given rise to the term "mining," where nodes solve these PoW problems to "mine" for rewards [5, 6]. This has led to the rise of a whole ecosystem of blockchain mining related entities. As for profit businesses, many individuals and companies run server farms dedicated to solving these cryptographic problems in locations with cheap electricity. Other firms have arisen to develop software and hardware technology solely for blockchain and Bitcoin mining [7].

The technical soundness of PoW legitimizes our system as a means to digitize legal contracts, as there is almost no chance a malicious party can submit false transactions to or mutate legal contracts that are deployed on the blockchain.

2.1.2 Block Linkage and Efficiency

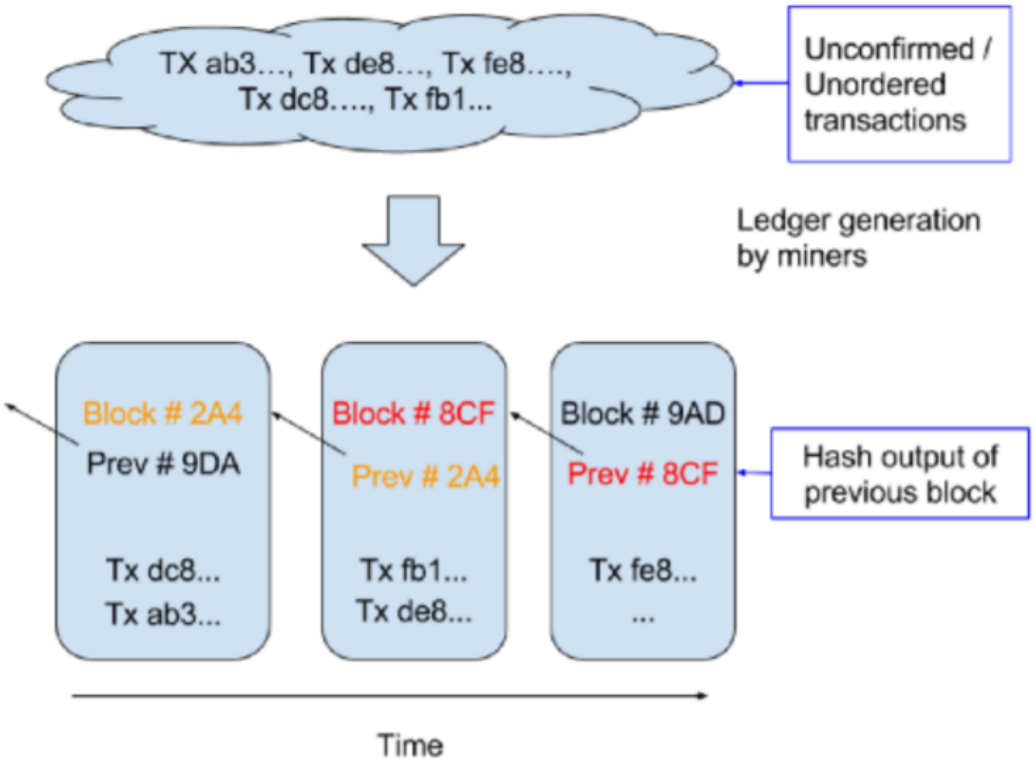

Figure 2-1: Architecture of a blockchain. Each block is comprised of many committed transactions, and stores its own hash and the hash of the previous block.

In order for nodes on a blockchain to verify that transactions are legitimate, one might think that all nodes on the blockchain would need to store a record of every transaction in the blockchain's complete history. This would be extremely inefficient and redundant, and waste valuable disk space. Instead, blocks on the blockchain are linked together by hashing, shown in Figure 2-1, where consecutive blocks in the blockchain store the previous block's hash [6].

For a node to verify the integrity of its local copy of blockchain information, it would need to only compare the stored hash of its last known block to information

stored by another node. If its stored hash matches with the knowledge of other queried nodes, the original node can be certain that the information it has stored locally about the blockchain is still verified and up-to-date. This is because the hash of the latest block depends on the hash and transactions of the previous block, which depend on the hash and transactions of the block before, etc [4]. At any given time, only a few nodes must store the complete transaction history of the blockchain in order for all transactions to be recoverable by all nodes. The rest of the nodes simply need to verify that their hashes are correct, thus ensuring that their chain is still legitimate.

To determine whether historical blocks can be discarded in exchange for only storing their hashes, a node simply checks the latest transaction time stamp for all accounts that transacted in that block. If this time stamp exceeds a certain threshold in the past, the accounts can be deemed dormant and the block transactions can be discarded from storage. In his paper, Nakamoto estimates that storing only block headers would cost each node about 4.2 megabytes per year, an insignificant amount compared to the local disk storage of existing servers today [8].

2.1.3 Applications

Nakamoto intended for the blockchain to serve as the basis of a digital currency and financial transaction system called Bitcoin. His vision was to create a "purely peer-to-peer version of electronic cash [to] allow online payments to be sent directly from one party to another without going through a financial institution" [8].

Since 2008 however, research and development in blockchain technology has skyrocketed, and many more uses for this technology have emerged. Financial companies and banks are now investigating using the blockchain to manage and audit transactions. Transport companies are using the blockchain to track shipments, imports, and exports. Food producers and restaurants have adopted the blockchain to assess the freshness of products in their supply chains. Due to the blockchain's permanent and auditable nature, many companies have turned to it to store everything from house deeds to birth certificates to academic degrees [5, 6, 9].

The applications for blockchain are endless, and we hope that this system will contribute as one of the many beneficial uses of this technology.

2.2 Ethereum

Ethereum is a separate blockchain technology from the blockchain that powers Bitcoin. It was proposed in 2013 and developed in 2014 by Vitalik Buterin, who later open sourced the project. Ethereum introduces the concept of smart contracts, which we elaborate on in Section 2.2.1, and is the most popular blockchain framework today for creating decentralized applications (DApps). A DApp is defined as any application using the decentralized blockchain rather than a centralized server to serve its back-end functionality. This project develops a DApp on the Ethereum blockchain. The major development languages for smart contracts on the Ethereum blockchain are Solidity, and more recently, Vyper.

2.2.1 Smart Contracts

Smart contracts are snippets of digital code deployed on the Ethereum blockchain that control and permission the transfer of cryptocurrency or other data transactions. Like the way a traditional legal contract defines rules and penalties in an agreement between two parties, a smart contract defines similar rules with an added benefit of being able to enforce these obligations. Because contracts reside on a distributed public ledger and all transactions are known to all nodes, the requirement of a trusted third party is removed [10]. The permanent and immutable nature of the blockchain also allows for legal contracts to maintain an audit trail. This trail can enable arbiters to easily vote on breaches that parties have reported, and could possibly even allow smart contracts to automatically detect these breaches should they occur via other transactions on the chain. Smart contracts have already enabled companies and individuals to establish autonomous banks, keyless access to online resources, crowdfunding platforms, and trading of financial derivatives, to name just a few applications [6].

Each smart contract that is committed to the Ethereum blockchain is assigned an address on the chain, just like an account. Smart contracts can be triggered or queried by sending transactions to this address, whereupon the contract will perform some predefined action depending on how it was programmed and possibly return data in a transaction back to the querent. Another benefit of smart contracts having their own addresses is that they can act as unbiased "middlemen" between the parties of a digital legal agreement on the blockchain. Instead of the parties having to transact with each other, parties can transact solely with the contract, which can then execute its code and forward the relative transaction data to the other parties. This increases the auditability of the contract and protects the address anonymity of each party participating in the legal contract.

Although smart contracts are deployed publicly on the blockchain, access controls can be programmed to protect the sensitive data contained inside these contracts. The smart contract code can check the address of the user querying the contract, and if this address is not in a list of permitted addresses, the contract will not return any info to the user. This way, although every node on the network maintains a hashed copy of the smart contract, only parties and arbiters who are a part of the contract can access the its contents.

2.2.2 Vyper

Vyper is a relatively new language used to write Ethereum smart contracts. Developed and then open-sourced in 2017 by Buterin, Vyper aims to model writing Ethereum smart contracts in a Pythonic-typed language. This is generally considered easier than writing smart contracts in Solidity [11], Ethereum's default development language, because Python is known for its ability to fast-prototype various projects and there is no need to learn a completely different programming language.

Vyper aims to provide three benefits to smart contract developers: ensuring that smart contracts are free from vulnerabilities, simplifying the language and compiler learning curve, and maximizing human-readability of developed code. To fulfill these goals, Vyper's built-in features include array bounds and overflow checking, provable

computational upper bounds for function calls, strong typing, and declared constant variables. These features help ensure that smart contracts are easy to develop and audit in Vyper, while still being secure from vulnerabilities.

Of course, there are trade-offs to making the code more readable and secure. Vyper is unable to provide class inheritance, function and operator overloading, recursion, and infinite loops. Adding any of these features would introduce the potential for attacks on the smart contract that could use up all the Ether within the contract by causing the contract to execute indefinitely [12]. Because our system is designed to model sensitive legal contracts, these trade-offs seem reasonable to ensure its security.

2.3 Verification Techniques

Two different verification techniques are used in various steps in our system.

Z3 is used to check if the Python representation of the contract accurately reflects the contract drafter's intentions for each clause. Z3 will also check whether all clauses can be satisfied at the same time, or whether the contract contains conflicting clauses.

KEVM verification, based on the K-framework, is applied at the Ethereum bytecode level. It tests the same objectives as Z3, except it is directed towards a lower-level language and is closer in the pipeline to where the contracts are actually deployed onto the blockchain.

2.3.1 Z3

Z3 is a Satisfiability Modulo Theories (SMT) solver open-sourced by Microsoft Research. It takes in a "formula," which in our case is a linkage of the clauses in a legal contract, and generalizes the boolean satisfiability of this formula. Logical formulas are represented by bit-vectors, and different boolean values are assigned to these bit-vectors to see whether the combination of these vectors passes validity and duality unsatisfiable tests [13]. These tests are used to assess whether the digital contract's clauses evaluate to the correct values given different hypothetical events that might occur after the contract's deployment.

Z3 is used by many developers of software verification and analysis tools to verify that their programs function as intended. One of the major strengths of Z3 is that its input proofs can be written in Python. This is highly beneficial as the higher-level representation of contracts in our system is also Python-based. [14–16]. We use Z3 verification early in our system, right after the Python version of the contract has been generated, so that later compilations of the contract do not need to be performed if any of the Z3 proofs fail.

2.3.2 KEVM

KEVM is an executable formal specification of Ethereum's bytecode-based stack language, based on the K-framework. The K-framework allows programming languages to be defined as a combination of "configurations, computations, and rules." It is used to build and test a mathematical model of a program to formally verify its accuracy [17]. KEVM harnesses this potential specifically for Ethereum, giving developers of smart contracts a method to formally verify their contracts before permanently deploying them on the blockchain. It does so by programmatically modelling the Ethereum Virtual Machine (EVM), which is the virtual stack machine within each Ethereum node that is responsible for executing the bytecode of smart contracts [10, 18].

Some of the guarantees that KEVM provides are type checking and domain-specific access control checking. Specifically, the framework allows smart contract developers to be sure that their smart contracts are not vulnerable to third-party access attacks, unlike early versions of Ethereum smart contracts. This vulnerability was most notably for the 2016 hack that exploited it, when a smart contract on the Ethereum blockchain was hacked for $50 million [19].

Similar to Z3, KEVM can also be used to check whether a smart contract outputted by our system functions as intended and accurately represents the legal clauses that the lawyer intended to include. Python can also be used to write the input proofs to KEVM, so there is little difficulty in translating proofs in our Python representation to be used in KEVM verification.

Chapter 3

Related Work

Our framework spans the topics of templated legal contract drafting, programmatically representing legal agreements, and formal verification of smart contracts. To our knowledge, no other single system integrates these three domains into one unified framework. Due to our project's scope discussed in Section 1.1, our system also relates to blockchain data sharing solutions. Existing research in each of these individual fields are addressed in Sections 3.1 to 3.4.

3.1 Templated Legal Contract Drafting

Multiple services have already tried to digitalize the legal contract writing process, seeking to eliminate the tedious task of drafting contracts on paper. Most of these services exist online, and provide users with templates for different legal use cases which are pre-filled with placeholder text. Some sites even extend past just serving lawyers, enabling individuals and small businesses with no legal experience to draft legal contracts without having to pay costly legal fees.

The three best known examples of these services are LegalZoom [20], Rocket Lawyer [21], and HotDocs [22]. Each of these sites enables users to simply sign into a web interface, use a template to draft their own legal contract, and generate a PDF or DocuSign secure digital document that they can either print or send electronically to other parties. These three services have proven to be commercially successful and

instrumental in saving significant time in the legal contract drafting process.

Our system's front-end web interface was based off the success of these platforms. We designed the web interface with ease-of-use as a key focus. The locations to add parties, arbiters, clauses, and clause parameters are easy to spot, and a user needs only to point-and-click to fill out a draft for a contract. Our web interface was also developed so that clauses could be templated and repeatedly used between contracts, drawing from the re-use of templates that these services provide.

3.2 Programmatic Legal Contract Representation

In the past few years, much work has been done in finding ways to represent legal contracts programmatically. This seems to be a very hard task, as there needs to be a way to develop a code-based system with a finite amount of object types to represent an infinite space of natural language texts that might appear in a legal contract. The work in this field influenced the design of our Python library.

In terms of converting natural language data sharing contracts into deployable smart contract code, the closest work has been done with Ergo as part of the Accord Project. Ergo is an open-sourced project looking to create a collection of legal contract schemas. Users fill in certain parameters to be included in the natural language data sharing contract, and the written language contract is populated with fields that act as placeholders for these parameters [23–25]. Some examples can be viewed on the Ergo Playground site [26]. In Ergo's case, although the platform is advertised as being for the blockchain, the final natural language legal contract is only converted to Ergo's intermediate logic language, rather than to Ethereum smart contract code. Our implementation will convert parameterized contracts into Vyper code that can be directly compiled and deployed onto the blockchain.

Researchers at the Treasury Department's Office of Financial Research have developed a computational representation of financial agreements. In their paper, Flood and Goodenough formalize the fundamental legal structure of a financial contract as a finite-state machine. They define the different states that the contract can be in,

as well as different actions that will transition the contract between these states. By enumerating all possible events that might occur and affect the contract, the authors were able to automatically model the state of the contract at any point in the future. In their paper, they list the key pieces of programmatically representing legal contracts as the list of actions that may be taken, outside world events, and logical strings that link chains of actions and events together to determine the final state [27]. Drawing from this, our system defines various actions that parties can or cannot take for the duration of the contract, as well as clauses that link sub-clauses and actions together to evaluate whether the contract still holds. These action and clause objects are further elaborated upon in Section 5.1.

Daskalopulu and Sergot researched logical-based tools for the analysis and representation of legal contracts, focusing predominantly on contracts relating to large-scale engineering projects and long-term trading agreements. From conducting an analysis of what was needed in the legal contract drafting process, they found that the best tools to programmatically generate contract representations were those which "determined whether given agreements were legally binding, enabled parties to specify their requirements and checked whether these requirements were compatible, and assisted drafters in writing the final outputted legal document." The presence of all three of these characteristics would result in the most helpful legal contract drafting tools. Daskalopulu and Sergot did not find any existing tools which satisfied all three requirements [28]. Our system aims to fill this void by providing a built-in Python library to represent legal contracts in which natural language descriptions can be easily recovered. Our system also provides automatic formal verification of the contracts to check whether their internal clauses are compatible, and that the clauses match the intended requirements of the user.

3.3 Blockchain Data Sharing

Because our project uses data sharing contracts as its scope for evaluation, it makes sense to review some prior work concerning data sharing over the blockchain. To our

knowledge, much of the past literature in blockchain data sharing has been in the area of healthcare, so we discuss this field below.

In terms of genomic data, Koepsell and Covarrubias have attempted to use a multi-chain blockchain platform as a medium to share anonymized genomic data for scientific research. This blockchain implementation protects sensitivity of data by only allowing researchers to view the metadata of the genomes rather than the personal information of the genome donors. However, a drawback is that only researchers are allowed to sign up as nodes on the blockchain. Data providers may not sign up, and must instead provide their genomic data to a researcher on the network. Researchers then upload the genomic data that they have collected to share with other researchers in the network [29]. This severely restricts the utility of this blockchain as data providers must still sign an off-chain agreement with researchers on the usage of their data.

In the field of medical imaging data, researchers at UCLA have developed their own implementation of a private chain to share data. Data providers upload their own data, which is then shared among physicians and personal heath record vendors. Although this implementation does support constraints on the data sharing that limit which vendors can view what data, it requires data providers to write their own data access smart contracts and deploy them themselves on the blockchain [30]. This is a complex task with a steep learning curve, so many data providers might become discouraged and less willing to share their data on this platform.

MedRec is a platform developed by MIT's Media Lab for medical record sharing based on the Ethereum blockchain. It allows health care providers to share medical records between themselves and with their patients, letting patients view an aggregate of their personal records from all health care providers that they have previously visited. Health care providers maintain control over the patient data stored on their servers, but enter patient-provider relationships for data access via smart contracts. These relationships are similar to the generalized data sharing agreements that our system enables. All patient data access and data sharing by health care providers is logged in an immutable ledger [31]. MedRec contains many privacy and security

features built into Ethereum smart contracts, but lacks the adjudication mechanism that our system provides in case of a data breach or data misuse.

Enigma is another project by the MIT Media Lab that researches protocols for decentralized personal data management on the blockchain. This system allows for automated access-control to data stored on the blockchain. The peer-to-peer network also allows multiple parties to store and run analysis on data while keeping the contents of the data private without the need for a central authoritative party. Enigma functions as a hybrid blockchain, with a specialized blockchain connected to a public mainstream blockchain. This specialized blockchain serves to manage access control, identities, and a permanent record of data sharing transactions, while the public blockchain handles payments and all the public metadata of the data sharing. Computation on this data is performed off-chain, and the data itself is stored in a distributed hash table. Enigma addresses many of the issues related to sensitive data sharing over the blockchain by including methods to restrict access to the actual data being shared, but does so by creating its own "Enigma blockchain" [32, 33]. Our system also enables users to securely share data by means of an enforceable legal contract, but does not require the use of a hybrid-blockchain model.

To determine what types of legal clauses and parameters we should include in the first iteration of our digital legal contract drafting system, we used Grabus and Greenberg's work studying data sharing agreement attributes. In it, they determine six higher-level categories that key attributes in data sharing agreements fall into, with these six categories being broken down further into 15 mid-level and 90 lower-level categories. A selection of these clause types is shown in Figure 3-1 [34]. We used this research and included a subset of these parameters and attributes in our first iteration of our system.

Many of these previous works provide a means to restrict data access based on preferences set by the data provider, but none give the ability to indicate a suspected breach of the agreement after the data has already been shared. This decreases the robustness of these systems as there is no recourse to data providers or requesters when agreements are breached. This project extends the above solutions for blockchain

based data sharing and not only provides an easy-to-use contract creator for generating natural language data sharing contracts, but also allows all parties a means to suspect a breach and enforce the contract terms. Also, data storage on our system is truly decentralized, with individuals storing their data in their own locations instead of in a central database. This makes the system very robust, scalable, and adaptable to different use cases.

3.4 Smart Contract Privacy and Verification

Now that we have seen work related to the drafting and functionality of blockchain smart contracts, it is important to note that these smart contracts would be meaningless if no effort is made to formally verify them before they are deployed. It is also paramount that the information contained in our system's outputted contracts remains private to only the parties and arbiters included in the contract.

An initiative to develop a hybrid blockchain system that preserves privacy in data sharing contracts is Ekiden, a collaboration between UC Berkeley and Cornell researchers. Ekiden aims to use hardware-based Trusted Execution Environments (TEEs) in combination with a blockchain to move execution of code and smart contracts off the chain into these isolated hardware modules to enable privacy preserving smart contracts. To do this, the smart contracts which are stored on the blockchain are encrypted, and to execute them they must be decrypted within secure hardware-based enclaves. The result of this execution is then again encrypted and stored onto the blockchain, where other secure enclaves can decrypt these results to reach a consensus on execution validity [35].

Other research conducted in smart contract privacy on the blockchain has experimented with storing the documents offline in secure servers, and only using the blockchain to verify that the documents exist and have not been tampered with. Known as "Proof of Existence," this technique involves storing the hash or fingerprint of the document on the blockchain, linked to the time-stamp that the user submitted the document for approval. This fingerprint is irreversible, meaning that no mean-

ingful information can be extracted from it even if all nodes on the blockchain can view the public fingerprint. Instead, to check that the off-chain document has been unmodified and is still valid, the same hash function is applied to the document at a later date and the hashed fingerprint is compared to the one stored on the blockchain [6, 36, 37]. This method protects the privacy of the sensitive contents of the document, but is not very useful for our system. Our system needs to enable parties and arbiters who are in the contract to view and audit the contents of the legal agreement, so only storing the hashed fingerprint of the contract on the blockchain would not be that helpful.

In terms of formally verifying smart contracts, most work has dealt with using one of three frameworks: KEVM, F*, and Isabelle/HOL. All of these frameworks formally verify smart contracts by creating programmatic representations of the EVM, and then executing inputted proof statements within this EVM model. Each individual proof is checked to see whether it always holds within the Turing-complete model. All three frameworks depend on the input proofs being written systematically and correctly, but are excellent in finding security and implementation vulnerabilities given these proofs [38–41].

Our framework relies on KEVM for formal verification of its automatically generated smart contracts. More details about how KEVM works can be found in Sections 2.3.2 and 6.2.

3.5 System Uniqueness

Although each of the topics addressed in this chapter has seen impressive research conducted in it, whether it be contract drafting services or formal smart contract verification, none of them address all the challenges that our end-to-end digital legal contract framework attempts to solve.

Templated legal contract drafting services provide nice user-interfaces for developing these documents, but still only output static legal documents that are either printed out or stored on a hard-disk somewhere. There is no potential for an electronic

adjudication process afterwards. Our system aims to change this by automatically deploying the contracts onto a decentralized blockchain, where they can be accessed and audited at any point in the future.

The previous work done in programmatically representing legal contracts and in blockchain data sharing all require the user to have knowledge of how to code, either in a proprietary language such as Ergo, or in a smart contract development language such as Vyper or Solidity [11, 12]. Our system eliminates this requirement and turns it into an option for advanced users, whereas regular users without this technical background can directly use the point-and-click web interface to draft their contracts.

Finally, although the various smart contract verification tools are similar to what we use in our system, in previous work the proofs for these verifications are only written after reading through the smart contracts in question. What makes our system unique is that the proofs are written before the smart contracts are even generated, in the legal contract's higher-level representation drafting stage. The advantage of this is that no intended meanings of the contract are lost in the translation and compilation process from higher-level language into smart contract code. Rather, users agree to the complete list of proofs to be formally verified, and then leave it to our system to guarantee the correctness of generated smart contracts in two independent steps of our framework, first with Z3 and then with KEVM.

Privacy & Protection

Sensitive Information

Regulations	*Preparing data*	*Access*
• Regulation used to define sensitive data (e.g., HIPAA, FERPA, etc.) • Compliance with federal/state/international data protection laws and regulations	• Identification of confidential/special categories of information (e.g., pii, proprietary) • Individual identifiers removed/anonymized prior to transfer	• Who has access to pii/confidential data • Who has access to proprietary information
Privacy	*Avoiding re-identification*	*Exceptions*
• Anonymization of data • Confidentiality and safeguarding of PII/sensitive data • Removal/nondisclosure of company/personnel identification in materials and publications • No contact with data subjects	• No direct/indirect re-identification • Statistical cell size (how many people, in aggregated form, can be released in groups) • Merging data with other sets (e.g., allowed with aggregated data—not in any way that will re-identify	• Exceptions to confidentiality • Conditions of proprietary information disclosure • Conditions of pii disclosure (who, what, and for what purpose?) • Limitations on obligations if data becomes public • Limitations on obligations if data is already known prior to agreement • Limitations on obligations if data given by 3rd party without restriction

Security

• Sharing non-confidential data • Password protection/authentication of files • Encryption	• Security training for involved personnel • Establishing infrastructure to safeguard confidential data

Data Handling

Use		*Physical*
• Each data field/elements to be accessed • Use of data: only for project-specific/research, or analytical use • Documenting all projects using the data	• Modification of data • Compliance with data updates (changes, removal, corrections) • Sharing data	• Copy/reproduction of data • Storage of data • Transfer of data (e.g., allowed methods)
Results		*Personal Gain*
• Presentation of data • Publication of data (e.g., prior approval needed or right to publically disclose publication)	• Results/reports and associated documents (e.g., must be provided copies) • Right to remove/delete confidential data from proposed publications	• Sale of/profit from data (e.g., noncommercial use only) • Licensing of data • No reverse engineering

Termination

• Conditions for termination • Destruction or return of data after agreement • 3rd party destruction or return of dataset • Confirmation of data destruction	• Data retained or used for period of time after termination • Which rights and obligations remain in effect after termination

Figure 3-1: A selection of the most frequently appearing clauses in legal data sharing contracts as found by Grabus and Greenberg.

Chapter 4

Front-End Interface

The entry point for most users of our system will be the front-end web interface. This interface serves to abstract away all the technical parts of the blockchain back-end, such that the only knowledge necessary to use our system is how to use a computer and modern web browser. Users will access the web interface by navigating to a URL in their browser, and either logging into a preexisting account or registering for a new one. Each party that intends to be a signatory on a digital legal contract should create a separate account. Upon creating a new contract, users will be presented with the view shown in Figure 4-1.

4.1 Contract Metadata

On the left side of the web interface, users will add all the parties who will be signatories of the contract. Each party will be assigned an address on the blockchain. These addresses will be given permission to access and query the final deployed contract on the blockchain.

There is also space to add arbiters of the contract. These arbiters are neutral, third-party mediators of the contract should any party suspect a breach of the terms. Usually, these arbiters are governing bodies or institutions with authority in their fields. For example, a health care related data sharing contract might have the National Institutes of Health, American Medical Association, U.S. Department of Health,

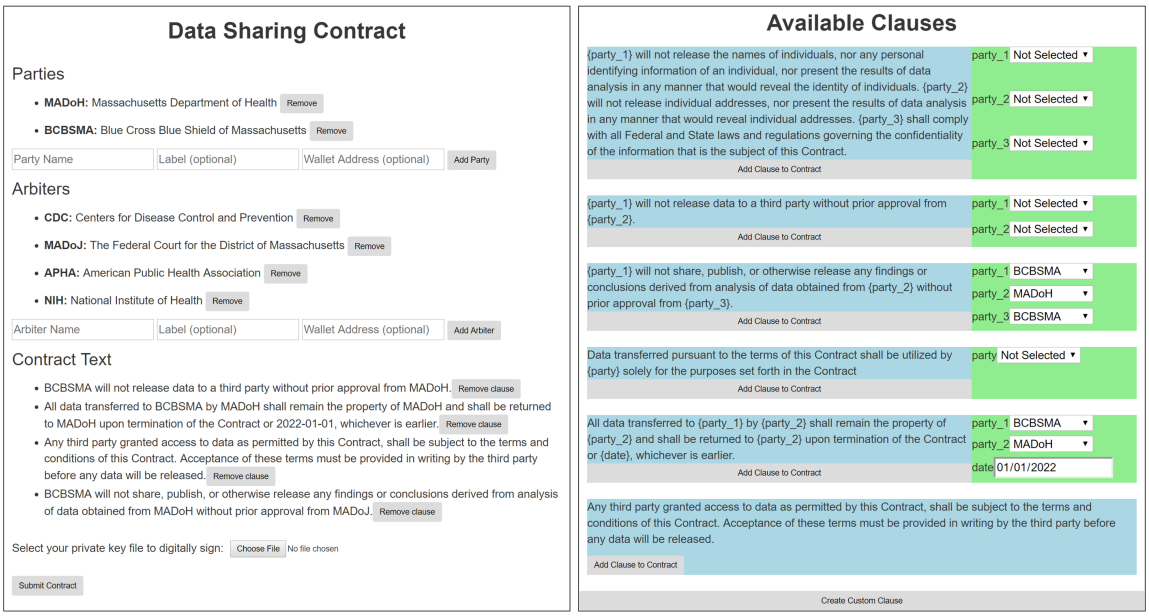

Figure 4-1: A screenshot of the front-end web interface that allows users to draft legal contracts.

etc. listed as its arbiters. Given this however, an arbiter role can also be granted to any entity agreed upon by all parties of a contract.

Finally, there is a space for a party to upload a digital signature file, such as a Secure Sockets Layer (SSL) key. This will authenticate the party's identity and serve as the party's digital signature in signing the legal agreement.

4.2 Adding Clauses

When drafting contracts, users of our system have the option to either select pre-written clauses with custom parameters, or to create their own completely custom clauses. Both options are discussed in the following subsections.

4.2.1 Pre-Existing Clauses

One option that users have is to select pre-written clauses from a library of common clauses, shown on the right in Figure 4-1. This library is populated with the most popular types of contract clauses that have been used in contracts created by other

users, aggregated across all users and with parameters removed to preserve anonymity. The library will automatically appear on the right-side of the contract drafting view. Users fill in the custom parameters of the clauses that they wish to include in their contracts, which are populated based on the parties and arbiters that have been added to the contract. Once a clause is added to a contract, the left side of the web interface will display the natural language text of this clause. Future iterations of our interface will also introduce search functionality to the clause library, as well as filtering clauses by type of legal contract or by how frequently they are used by other users of our system.

The library of common clauses shown to every user of the interface is crowd-sourced from other users of the system. Contributing to this library will be addressed in Section 5.2. Creating new clauses solely for one's own contract is covered in Section 4.2.2.

4.2.2 Custom Clauses

Contract drafters have the additional option of creating their own custom clauses should no suitable ones be found in the clause library. These custom clauses will be built from other previously defined clauses present in the library, or from the boolean building blocks of clauses, which we will refer to as "actions." These actions are simple descriptors and conditions to be checked in terms of a legal contract, such as whether a date has passed, whether a payment has been made, or whether data has been shared to third parties. Each action evaluates to a boolean value of `True` or `False`, and its value can be easily checked by any party or arbiter.

To create a custom clause, users click on the button at the bottom of the clause list and are presented with the modal shown in Figure 4-2. In this modal, five different clause types can be created:

- **OR clause:** A linkage of two existing clauses or actions with an *OR* modifier. In the newly created clause, at least one of the two linked clauses or actions must be satisfied.

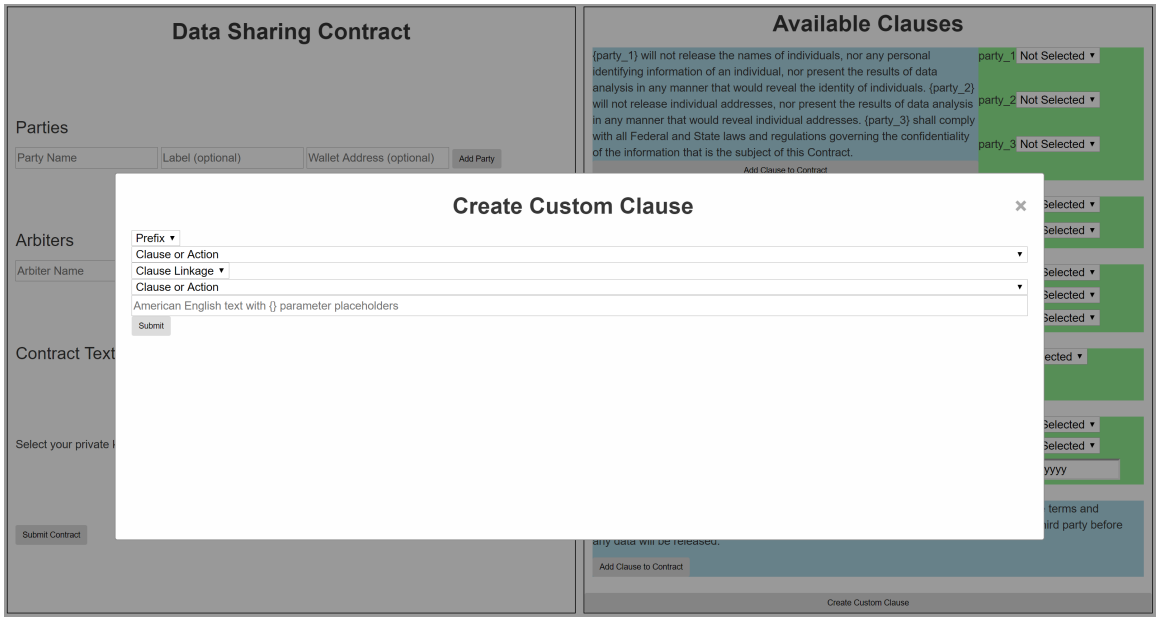

Figure 4-2: A screenshot of the modal that guides users through the creation of a custom clause.

- **AND clause:** A linkage of two existing clauses or actions with an *AND* modifier. In the newly created clause, both of the linked clauses or actions must be satisfied.

- **NOT clause:** A single clause or action, prefixed with a *NOT* modifier. In the newly created clause, the original clause must not be satisfied.

- **IF-SHALL clause:** A linkage of two existing clauses or actions such that if the first clause or action is satisfied, the second one must be satisfied too.

- **IF-MAY clause:** A linkage of two existing clauses or actions such that if the first clause or action is not satisfied, the second one must also not be satisfied.

The modal guides the user through the clause creation process by restricting the prefix and linkage types depending on what the user has selected in other fields. Dropdowns allow the user to select the clauses and/or actions to include in their custom clause. These clauses and actions display parameter placeholders for variables such as party, duration, date, etc., so that the user knows which parts of the clause can

be customized further when inserting it into the contract. Actions and clauses are described in further technical detail in Sections 5.1.3 and 5.1.4, respectively.

Once a user has developed their custom clause, they must also write the parameterized natural language text associated with the clause. This will be the text that is displayed on the left side of the interface when the clause is added to the contract. This text will also appear in the final committed contract.

After a user develops their custom clause, clicking save adds it to the user's personal library of clauses. It is now available to be added to the contract or included in additional custom clauses. Note that including the clause in the user's personal library does not add it to the global library that is shown to other users. This both serves to protect the privacy of information revealing clauses that the user may develop, as well as reduce the clutter on other users' dashboards. Adding to the global library of clauses is addressed in Section 5.2.

4.3 Contract Submission

Once the user is satisfied with the contract, they must upload their digital signature file, and then submit the contract for review. Parties who have been added to the contract can then log in to their accounts and view the current state of the contract. Once all parties have agreed to the most recent revision of the contract by each signing with their digital key file, our system's back-end will automatically compile the contract down to Ethereum bytecode and deploy it permanently on the blockchain. This process requires multiple steps, which are addressed in Sections 5 to 7.

4.4 Post-Submission Maintenance

After a contract has been deployed on the blockchain, users can log in to the web interface to view existing legal agreements that they are participating in. Upon log in, users can query the current state of the contract, which is covered in Section 7.2.1. Users can also submit a suspected breach notice and ask the arbiters of a contract

to vote on whether this breach has occurred. Breach requests and the arbiter voting process are covered in Section 7.3.

Chapter 5

Python as a Higher Level Contract Representation

One of the goals for this project was to develop a code-based higher-level representation of legal contracts that is not restricted to the blockchain. The motivation was that if users of our system want to export their legal contracts off the chain into another form of digital media, they could export the code-based representation of their contracts. We decided to use Python for this representation, as Python is known for its success in code prototyping and is also a very text-like, readable programming language. Furthermore, the learning curve for Python is not as steep as other programming languages, so users of our system who wish to bypass the web interface and write contracts directly using our Python library can choose to do so without much upfront investment.

Our system uses this Python representation as its backbone. From Python, we serialize and de-serialize the digital contracts into and from the JavaScript Object Notation (JSON) format for easy storage and retrieval. We use the Python representation of contracts to write Z3 and KEVM proofs for them. We also compile the Python to Ethereum's Vyper smart contract language for deployment onto the blockchain.

5.1 Classes

We created a few classes in our Python library to represent each part of a legal contract. They are described in Sections 5.1.1 to 5.1.4. We capitalize the names these classes when referring to them in this chapter to distinguish between the class types.

5.1.1 Parties

A Party object represents an individual or entity who has an interest in the fulfillment of terms in a contract. Party objects must have a name when they are initialized, and are either initialized with or assigned a unique address on the Ethereum blockchain. Using the example in Appendices A.1 and B.1, a Party could have a name of "Rhode Island Department of Health," and an Ethereum address identifier of 0x2B5634C42055806a59e9107ED44D43c426E58258.

5.1.2 Arbiters

An Arbiter object represents an individual or entity assigned to mediate and decide disputes when the Parties of a contract cannot come to a mutual agreement. Most of these disputes are presented in the form of contract breaches, when one Party in a contract suspects that another Party or Parties have violated the terms of the legal agreement.

The Arbiter class shares many of the same attributes as the Party class. Arbiter objects must have their own names and unique addresses on the Ethereum blockchain. Using the example in Appendices A.1 and B.1, an Arbiter could have a name of "Centers for Disease Control and Prevention," and an Ethereum address identifier of 0x689C56AEf474Df92D44A1B70850f808488F9769C. Unlike Party objects, Arbiters do not have any interest in the fulfillment of terms in a contract. They are simply neutral third Parties used to manage disputes.

5.1.3 Actions

Action objects are the building blocks for all legal Clauses produced using our system. Actions are defined as specific behaviors or deeds that Parties can perform. Each Action must evaluate to a binary value of `True` or `False`, representing whether or not the deed it represents has occurred. During a legal agreement's lifespan, various Parties and Arbiters can vote on the current state of each Action, which would determine if the overall contract has been breached or remains in good standing.

Each Action is assigned an identifier name in our Action repository when it is initialized, and also a description of what the Action represents. Also, when an Action object is instantiated, two optional Python lists of Party and Arbiter objects are passed in to the constructor.

The Party list contains all Parties that have an interest in the value of that Action. This list usually includes all Parties of a contract, so it is by default set to all Parties if no optional list is passed in. However, in the case of a multi-Party legal agreement, it could be the case that one or more Parties does not have any interest in the outcome of some of the Clauses in the contract, and do not mind if those Clauses are breached. Then, for the Action objects contained solely in these Clauses, their Party lists would be subsets of all Parties in the contract.

The Arbiter list contains all Arbiters that have the power or ability to judge the truth value of that Action. Similar to the optional Party list, it is by default set to all Arbiters for each Action object. However, there are cases where not all Arbiters are able to vote on the truth value of an Action, due to jurisdiction laws or lack of information access. In these cases, the Arbiter list for these Actions is set to the subset of Arbiters that do have the power to accurately and fairly judge the truth value of these Actions.

Each Action object also takes in an optional boolean value when it is initialized in our Python module. This boolean value will be the starting truth value of that Action the moment the contract is signed and deployed on the blockchain. By default, this value is set to `False` if no optional boolean parameter is passed in.

Finally, each Action object takes in an optional Arbiter quorum value. In the case of a suspected breach of contract, when Arbiters are voting on the truth value of each Action that they have been requested to vote on, it is possible that one or more Arbiters either abstains from voting, does not have enough information to vote, or simply is unreachable to vote on an Action(s) truth value. In this case, the Action's current boolean state value can only be overturned by an Arbiter vote if the minimum quorum of Arbiters have submitted a vote on the issue. Otherwise, the boolean state for the Action will remain what it was prior to the vote request. Breach voting is covered further in Section 7.3.

For example, an Action object representing that the Department of Health (`doh`) must comply with state laws can be written in Python as:

```
ComplyWithLawsAction(doh, "State", arbiters, parties, default=True,
↪  quorum=3)
```

5.1.4 Clauses

Clause objects model complete legal statements to be included in the legal contracts created by our system. They are represented as linkages of Actions and other Clause objects. Each Clause or Action in a Clause object is linked by a logical operator. Five different Clause objects can be created, which all inherit from the parent Clause class. The available Clause types were listed in Section 4.2.2, and are repeated here with their Python module names for convenience. Note that each Clause class below inherits from the parent Clause class:

- **OrClause:** A linkage of two existing clauses or actions with an *OR* modifier. In the newly created clause, at least one of the two linked clauses or actions must be satisfied.
- **AndClause:** A linkage of two existing clauses or actions with an *AND* modifier. In the newly created clause, both of the linked clauses or actions must be satisfied.
- **NegatedClause:** A single clause or action, prefixed with a *NOT* modifier. In the newly created clause, the original clause must not be satisfied.

- **IfShallClause:** A linkage of two existing clauses or actions such that if the first clause or action is satisfied, the second one must be satisfied too.

- **IfMayClause:** A linkage of two existing clauses or actions such that if the first clause or action is not satisfied, the second one must also not be satisfied.

The constructor of each Clause object takes in an arbitrary number of Actions or Clauses, depending on the Clause object being created. For example, the NegatedClause constructor specifies that exactly one Action or Clause object should be passed in. The IfShallClause and IfMayClause constructors take exactly two Actions or Clauses. The OrClause and AndClause can be passed an arbitrary number of at least two Actions or Clauses.

The newly created Clause object can then be passed into other constructors to recursively create more complex Clauses. Examples of Clauses can be seen in Appendix B.1.

5.2 Contributing to the Clause Library

If a user wishes to create a generic Action or Clause object that can be used by other users, they are welcome to add to the web interface's public repository, described in Section 4.2.1. Note that Python knowledge is required to contribute to this library.

To add a Clause or Action, users have two options. The first is to start from scratch and write their own Action and/or Clause object definitions. Users may also combine various Action and Clause objects predefined in the downloadable Python file that our system provides containing all Action and Clause objects currently in the public library. Users can then upload a Python file to the web interface containing their defined Action or Clause object that they wish to add. This file must also contain the natural language English text of what the Action or Clause represents, to help other users understand its meaning when incorporating this Action or Clause into their own contracts.

When contributing to the repository, users must also write proofs to guarantee

that a Clause evaluates as intended. The specifications for writing these proofs and what they guarantee are covered in Section 5.4.

Note that publicly contributed Action objects to the repository will not appear on any user's dashboard. Rather, to reduce clutter, the dashboard will only show contributed Clauses. However, Actions added to the repository will appear in the downloadable Python file that the server provides contributors. Contributed Actions will also appear as selectable components when the user uses the web interface's custom Clause creation tool, described in Section 4.2.2.

5.3 Python Contract Representation

Combining all the classes defined in Section 5.1, we can construct a Python object representation of every contract that is drafted using our system. This Python representation is either written as a whole by users more accustomed to drafting their contracts using our Python library, or is automatically compiled by our system from the contract details that the user has filled in on the web interface. A full example of a legal contract converted to its Python object representation is shown in Appendix B.1.

This Python representation is then automatically converted into four separate formats. The first is a simple serialized JSON file containing the contract details for easy storage and machine-readable export. An example of this JSON file is reproduced in Appendix B.2. The second is a human-readable Portable Document Format (PDF) document in case users wish to view the traditional form of the legal agreement they had drafted. The third output is an automatically compiled `.smt2` file used for Z3 verification, which is addressed in Section 5.4. The final output, dependent on whether Z3 verification passes, is an automatically compiled Vyper file representing the deployable smart contract version of the legal agreement. The Vyper format is covered in depth in Section 6.

5.4 Z3 Verification of Python Representation

The first of two formal verification systems built into our framework, Z3 is used to check the Python representation step of our system. Every legal agreement represented in Python is automatically compiled to a `.smt2` file, the input format required by the Z3 Theorem Prover. An example of the contents of this file is replicated in Appendix B.5. Each proof, whether it was written by the Clause contributor, or whether it is a generic proof for the type of Clause being added, is checked individually by Z3.

A proof for an individual Clause fixes the result of one or more sub-clauses or sub-actions and the result of the Clause itself. A proof also contains the expected outcome of whether the Clause is satisfiable or unsatisfiable based on these fixed components. By combining proofs with both satisfiable and unsatisfiable outcomes, we are able to assert that a Clause functions correctly under all possible states. For example, for a generic IF-MAY Clause, some results and satisfiability assertions that can be written are:

IF value	MAY value	Clause Result	Satisfiable?
True	True	True	True
True	True	False	False
True	False	True	True
True	False	False	False
False	True	True	False
False	True	False	True
False	False	True	True
False	False	False	False

Table 5.1: Table of fixed results and the Z3 proofs' expected satisfiable outcomes for a generic IF-MAY Clause object.

The English representation of the above truth and satisfiability table is as follows:

- If the *IF* portion of the Clause is `True`, then the whole Clause must always evaluate to `True` regardless of the truth value of the *MAY* Clause. This is because when the *IF* condition of the Clause is satisfied, the Parties in the contract are not bound anymore by any conditions set forth in the *MAY* Clause. Thus, in

our Z3 proof we can assert that in this case, the whole Clause evaluating to `True` is satisfiable, but that the whole Clause evaluating to `False` is unsatisfiable.

- If the *IF* portion of the Clause is `False`, then the whole Clause must always evaluate to the opposite of the *MAY* Clause's value. This is because the Clause as a whole is `True` if and only if the *MAY* condition does not occur when the *IF* portion is `False`. Thus, in our Z3 proof we can assert that in this case, the whole Clause evaluating to the same parity as the *MAY* portion is unsatisfiable, but that the whole Clause evaluating to the opposite parity as the *MAY* portion is satisfiable.

After running the theorem prover, Z3 will return a solution of possible Action states if the proof is satisfiable, or return a proof showing why no such combination of states exists if the proof is unsatisfiable. Any discrepancies in the satisfiability of the proofs compared to the expected results are displayed to the user. Discrepancies raised by the theorem prover can signal various problems with the drafted legal agreement. First, the Python representation of a Clause might be incorrect. Second, it is possible that there is a contradiction between Clauses in the legal agreement. Lastly, it is possible that one or more of the Clauses that the user has included in the contract draft actually guarantees something other than what the user had intended. From the outputted discrepancies, a user could go back to the drafting process and fix these errors before submitting an updated contract for verification.

It is important to note that Z3 verification does not guarantee certain properties of the contract. Namely, Z3 does not guarantee that the natural language text of a Clause matches its Python language representation. Z3 also does not guarantee that parameters such as Parties or dates inputted by the user into the placeholder fields are specified as intended [13, 42].

Chapter 6

Converting Python to Vyper

Once the Python representation of the contract has been formally verified by Z3, it is automatically compiled into Vyper. Vyper is the programming language used to write the Ethereum smart contract representation of our legal agreements so that they can be deployed on the blockchain.

The predominant language used to write Ethereum smart contracts is Solidity [43]. We chose Vyper over Solidity when designing our system for multiple reasons. First, Vyper was developed based on Python, so it shares much of the same syntax and many of the same object representations as Python. Second, Vyper was developed with a focus on security and protection against unintended contract vulnerabilities, as well as a focus on the readability of produced code. When dealing with smart contracts representing digital legal contracts, we need our system to be as secure as possible, and the generated smart contract code should be as readable as possible for easy checking to see whether it matches the lawyers' intentions. Vyper language details are discussed in Section 2.2.2. Some features supported by Solidity are not present in Vyper, but we considered the trade-offs and determined that the presence of these features was not worth the increased potential for the contract execution vulnerabilities that they introduced.

6.1 Python to Vyper Compiler

Vyper is generated by our system using our Python to Vyper compiler. The source code for this compiler is replicated in Appendix B.3. This compiler takes in as input a Python object representing the complete legal agreement, including Parties, Arbiters, Actions, and Clauses. It then outputs a `.vy` file on the server representing the Ethereum smart contract. An example of a produced Vyper file is replicated in Appendix B.4.

6.2 KEVM Verification

Before the generated Vyper code can be deployed on the blockchain, we introduce another formal verification step in our system. The purpose of this additional step after Z3 verification has already passed is to check whether the legal contract representation was corrupted in any way by our Python to Vyper Compiler. Once a smart contract is deployed on the blockchain, it becomes immutable and self-executing, so it is important to correct bugs or other implementation mistakes before the legal contract is deployed.

We use KEVM, an Ethereum Virtual Machine execution simulator based on the K-framework, described in Section 2.3.2. KEVM verifies at the Ethereum bytecode level, which is generated by inputting the Vyper contract into the Vyper language's built-in compiler [12]. The same proofs that were used for Z3 verification are then converted to the K-framework. KEVM then validates in a similar fashion to the Z3 Theorem Prover that each proof holds, except on the Ethereum bytecode level. This verification ensures that the generation of Vyper source code and the Vyper compiler work as expected, and that the contract logic specified in the generated byte-code both is free of security vulnerabilities and matches the lawyers' and parties' intents. By using the same proof format, our framework does not require additional work by users to benefit from this additional verification step.

Chapter 7

Deployment on Ethereum Blockchain

After the automatically generated Vyper smart contract has been formally verified, the digitized legal contract is ready to be permanently deployed onto the Ethereum blockchain. This section will cover the process of taking the compiled Vyper file and committing it as a blockchain smart contract, as well as post-contract submission querying and the breach process.

7.1 Tools

This section addresses the tools that our system uses to deploy Vyper smart contracts to the Ethereum blockchain and that we used to test our system implementation.

7.1.1 Rinkeby Test Network

To test our system, we deployed the smart contracts onto Ethereum's Rinkeby Test Network. The Rinkeby Test Network is a blockchain designed to model the main Ethereum blockchain, but runs on a disjoint set of nodes. It serves as a public test network, where developers can deploy their drafted smart contracts and test interactions with them at no cost and without having to worry about vulnerabilities related to monetary attacks. This is because the Ether tokens distributed on the Rinkeby network can be obtained for free from the Rinkeby Ether faucet. To avoid

malicious overuse of the Rinkeby Ether faucet and accumulation of unneeded tokens, the faucet requires users to authenticate by publicly posting a unique string to one of their social media profiles. Only after the posting has been confirmed and seen by the Rinkeby faucet is the testnet Ether awarded to the requester's account [44, 45].

The main difference between Rinkeby and the main Ethereum network is that Rinkeby functions with a Proof-of-Authority rather than a Proof-of-Work consensus algorithm. This means that the nodes on the Rinkeby network are all controlled by the Rinkeby organization, rather than by decentralized parties. Functionality wise, Rinkeby acts the same as the main Ethereum network for testing smart contracts because the contract confirmation process by the various nodes does not affect the contract once it is deployed [46].

7.1.2 MetaMask

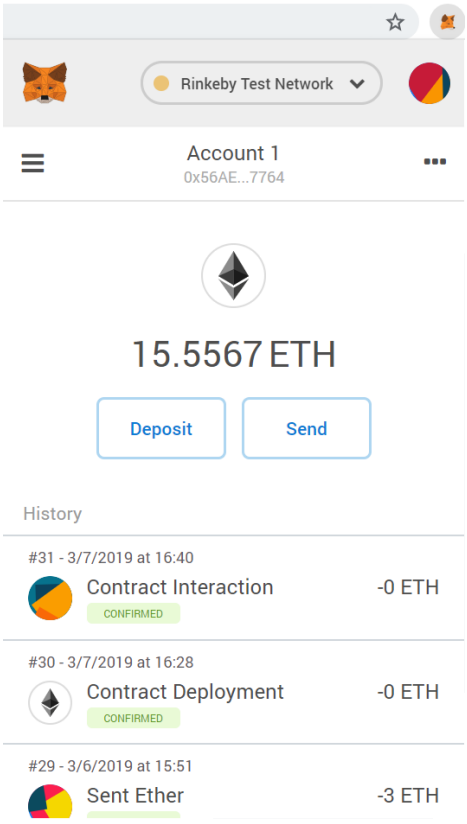

Figure 7-1: The MetaMask Chrome extension, shown here connected to the Rinkeby Test Network.

We used MetaMask to test deploying the compiled Vyper contracts that our system outputs to the Rinkeby testnet blockchain. MetaMask is a Google Chrome extension that simulates running an Ethereum node directly within one's browser. Users are able to run DApps using MetaMask without having to install a full Ethereum node. MetaMask supports having multiple accounts, which enables us to test the various interactions that each party and arbiter can have with a deployed contract. The extension can simulate a node on the real Ethereum blockchain, a decentralized testnet, or a locally hosted private blockchain [47]. Figure 7-1 shows the default user screen for MetaMask.

7.1.3 Remix for Smart Contract Compilation and Deployment

Remix is an online smart contract integrated development environment (IDE) developed by the official Ethereum organization. It supports writing smart contracts in both Solidity and Vyper, as well as a multitude of other less popular smart contract development languages. Using built-in compilers for these languages, Remix can compile the smart contract code into universal Ethereum bytecode. Via the MetaMask extension, Remix can then connect directly to the various blockchain networks supported by MetaMask and deploy the Ethereum bytecode onto the chosen blockchain.

During compilation of the smart contract code, Remix automatically checks for various security vulnerabilities and implementation errors such as transaction origin restrictions, contract execution exhaustion, dynamic array iteration, and lack of return statements. Severe errors stop compilation and are pointed out as errors to the user, while warnings and other non-critical errors appear in the log at the bottom of the IDE window [48]. We used Remix to test both the correctness of our system's outputted Vyper contracts, as well as the querying and arbitration of deployed legal contracts on the Rinkeby testnet.

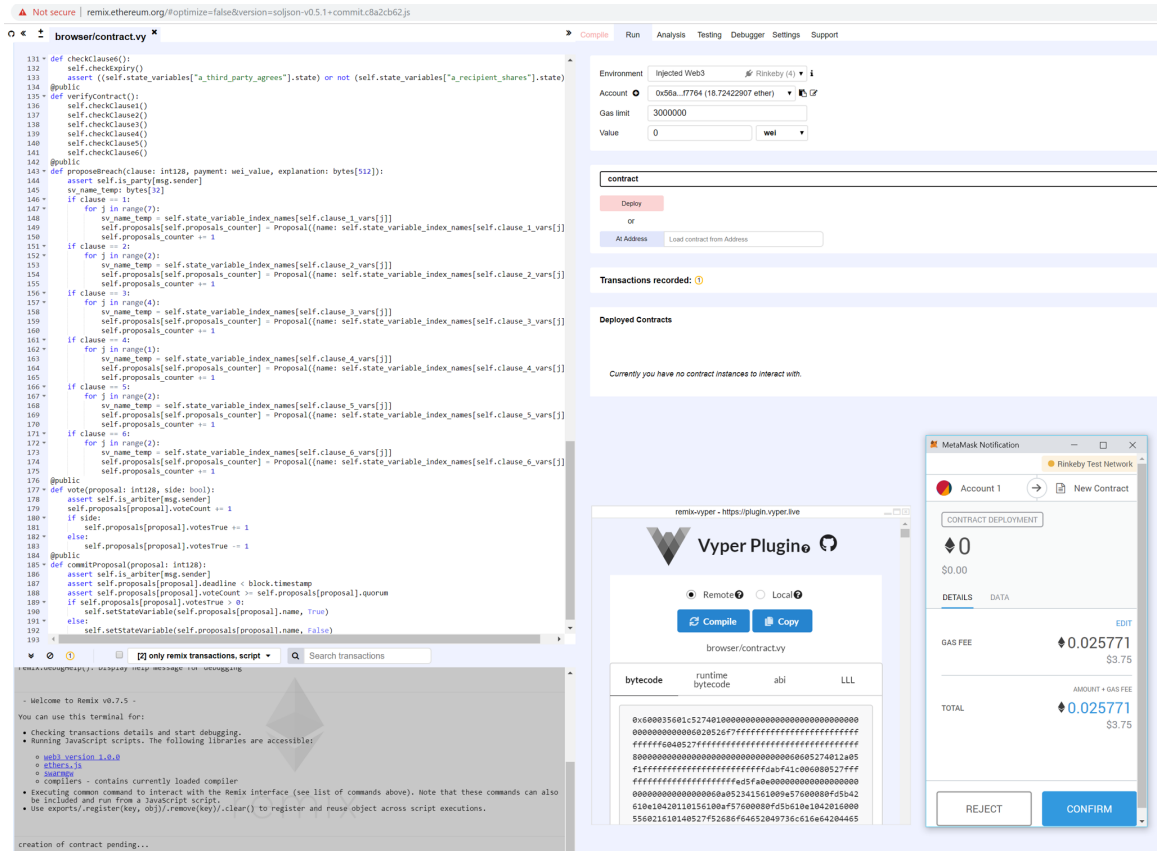

Figure 7-2: An example Vyper contract generated by our system being deployed onto the Rinkeby testnet via Remix using the MetaMask browser extension.

7.2 Contract Deployment

To demonstrate deployment of the Vyper file automatically compiled by our framework, we copy the file contents into the Remix IDE, and then use the built-in Vyper plug-in to compile this code into Ethereum bytecode. As seen in Figure 7-2, this plug-in displays the compiled bytecode and any compilation or syntax errors. After making sure the MetaMask extension is set to the Rinkeby Test Network, clicking the red "Deploy" button brings up a MetaMask confirmation window. This window displays the estimated transaction cost to deploy the contract to the blockchain. This minimal cost is a fee paid to the Ethereum miners to confirm the contract and add it to a block. In future iterations of our system, this fee will be paid for by the parties listed on each contract. After confirming the transaction details in the MetaMask screen, the contract is sent in a transaction to be deployed on the blockchain.

Finally, a contract is committed to the blockchain after at least six miners confirm the accuracy of the transaction, and a confirmation message displays in the Remix interface.

Note that Remix and MetaMask provide a visual and manual method to deploy our system's legal contracts to the blockchain. This is for demonstration purposes only. In our actual system framework, all of this deployment is automatic and the transaction is sent directly from an Ethereum node running on our web server. The Vyper compiler plug-in is also not used to compile the Vyper contract. Rather, a locally installed Vyper distribution and compiler on the server-side compiles the smart contract code into Ethereum bytecode.

7.2.1 Querying Contracts

In future iterations of our system, after the legal smart contract has been deployed onto the blockchain, if users wish to query or review the contents of the contract, they can simply log-in to the web interface to see a list of deployed legal contracts that they are listed as a party on. This functionality has been designed and prototyped but not yet implemented.

We again use the front-end Remix interface to show a more elaborate version of what the user will be able do. As shown in Figure 7-3, the user is presented with a list of function calls in the deployed contract, as well as methods to access the values of different variables in the contract. The user simply fills in the necessary parameters to call the specific function or method, and the result is returned in Remix. The two grey grids at the bottom of Figure 7-3 show queries to check whether the second and sixth clauses of an example deployed legal contract still hold, and the checkmark represents a successful result of `True`.

7.3 Adjudication Process

This section details the events that occur should any party listed on a contract suspect a breach of terms during the lifetime of the legal agreement.

7.3.1 Proposing a Breach

The party representing the accuser must first submit a breach proposal, stating that they believe a breach to the contract's terms have occurred. To do so, the party can log-in to our front-end web interface, where they will see a list of all legal contracts they are listed as a party in. Clicking the "Problem with this contract" button next to the contract allows the party to submit a breach proposal. In the proposal, the party first selects which clause or clauses of the contract they believe the violator has breached. They then fill out the reason why they suspect there is a breach, any evidence they might have found to support their theory, an amount for monetary compensation that they are requesting the breaching party pay, and a deadline for arbiters to vote on this breach. Clicking "Submit" automatically submits a transaction to the blockchain smart contract notifying it about the possible breach.

A breach request notification is also sent out to all other parties on the contract. Before the arbiters begin investigating the breach, the other parties on the contract have a pre-determined amount of time to add additional evidence and reasoning to the dispute. This design enables the defendant(s) to bring counterclaims, similar to physical legal proceedings. This additional evidence and reasoning is also sent to the smart contract.

7.3.2 Arbiter Voting and Resolution

After the pre-investigation time has expired, our system automatically sends all arbiters of the contract a notification that a party has requested a breach vote. At the same time, the deployed smart contract creates a breach proposal on the blockchain for arbiters to vote on. This breach proposal contains the contents submitted by the victim party in the initial request as well as any counterclaims, and opens voting on the proposal for the pre-defined time chosen by the victim party.

Once notified, arbiters can log-in to our system's web interface and review the submitted evidence and reasoning for the suspected breach. They can then conduct both an on-chain and off-chain investigation of the matter. The vote request then asks

all arbiters to vote on the `True` or `False` state of each Action object present in the disputed clauses. Based on their investigation, arbiters may then vote on this state by submitting either `True`, `False`, or `Abstain`. An arbiter may choose to abstain from voting on the state of one or more actions should they feel that there is not enough evidence to make a sound ruling, or that they do not have enough expertise on the specific action to determine its fair state.

After the voting period, all votes are tallied, and Action states are updated as appropriate. In order for the arbiters' votes to hold, the minimum quorum specified in each Action object has to have been met, and the arbiters who voted must have a simple majority in agreement on the new state. If these conditions are not met, the state of the Action remains what it was before the breach vote. This closing and tallying of the votes, and updating of Action states, are all performed automatically by the blockchain smart contract.

After each Action has been updated to its new `True` or `False` state, the smart contract then reevaluates the value of each disputed clause. Should it detect that the clause previous evaluated to `True` but now evaluates to `False`, the compensation requested by the victim is automatically deducted from the breacher's account and paid to the victim.

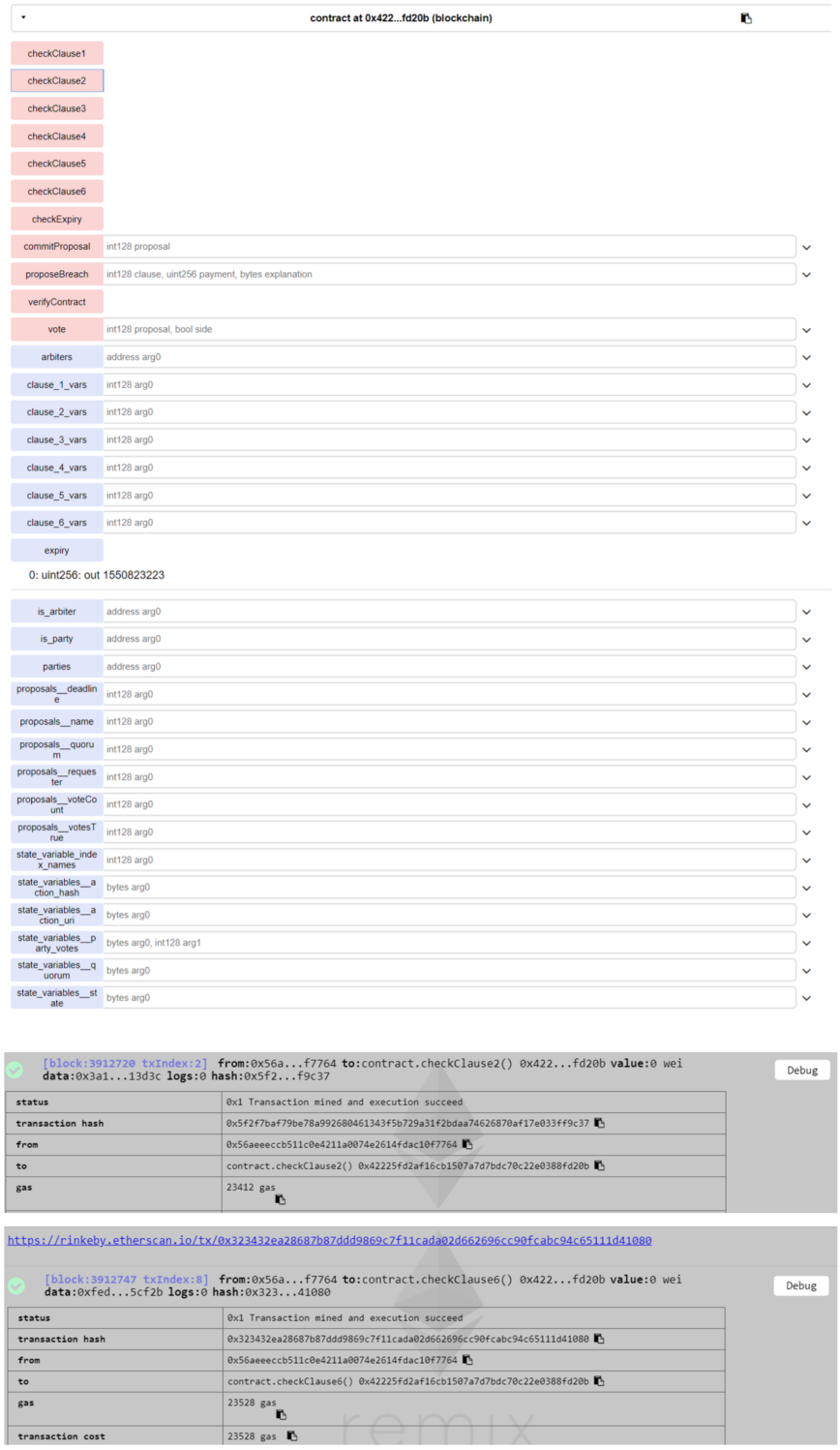

Figure 7-3: Using the online Remix IDE to query a legal contract deployed on the blockchain. The two queries at the bottom of the figure show that those specific clauses have not been breached for this specific contract.

Chapter 8

Evaluation

To measure the success our system, we needed to evaluate it on four different metrics. The first was how many clauses in natural language legal texts could be accurately represented by our system. The second was how effective our contribution of a clause repository was to saving drafters' time in writing the contracts. The third was to evaluate the success rate and run-time of our automatic formal verification steps. Finally, we needed to evaluate that the blockchain deployment and adjudication processes work as intended.

8.1 System Coverage

Although we were influenced by the data sharing legal contract scope when designing our system it is important that our system is able to model many different clauses from any legal field with our Action and Clause structure. If the system were unable to model a substantial percentage of clauses that appear in real-world paper-based legal contracts, our framework's utility would be severely decreased.

Evaluation was performed by randomly selecting five pages of legal text from two sample data sharing agreements, which are reproduced in Appendices A.1 and A.2. We then attempted to reproduce each of the clauses and actions in these agreements on both our front-end web interface, to simulate a normal user, and our Python library module, to simulate a more advanced user.

8.1.1 Drafting Using the Web Interface

Before beginning to draft the contracts on the web interface, the clause repository was pre-filled with common data sharing agreement clauses and actions chosen from Grabus and Greenberg's work in Figure 3-1 to simulate the crowd-sourced library of clauses present on every user's front-end web interface [34]. Some examples of these clauses are limitations in data sharing to only approved parties, data sharing compliance with existing laws, monetary payment to be made on a specified date or after a specified time, etc. The default five clause types explained in Section 4.2.2 were also included in the repository.

Working through the different contract clauses, we first noticed that a lot of contract text is merely filler text or background information. These paragraphs did not hold any legal meaning, but did help the reader understand the context of the contract and the definitions within it. Because our system's goal is to only model the legal clauses of a contract, we do not have the capability to include this filler text in the final contract. It would have to be added in between clauses after our system outputs the contract.

For the remaining text, there were a total of 93 actions and clauses we needed to model with our system. Unfortunately, a limitation of the front-end interface is that it only lets users create new custom Clause objects, but not new custom Action objects. The Action objects that users wish to include in their contracts or custom clauses must already be written in the crowd-sourced repository. The reason for this restriction is that Action objects require a lot of input in their Python object representation, detracting from the ease-of-use of the front-end. We hope that this problem will be mitigated as more and more users use the system, so that the Python back-end clause repository can be populated with many different Actions for users to choose from.

Given this limitation, out of the 93 actions and clauses we evaluated on, we could accurately represent 61 of them for a 65.5% coverage rate using the front-end web interface. This represents moderate coverage, and should be suitable for users wishing

to draft simple contracts. Note that this value should be interpreted with some caveats. First, as more and more users use and contribute to the repository, the coverage should increase as more usable Action and Clause objects will exist in the repository. Second, we only evaluated on legal data sharing contracts because this was the scope we chose. It is quite possible that other legal fields might require other types of clauses, although we believe that the five default clauses we allow should cover most legal clauses from any field. Other legal fields will definitely require a whole set of different Action objects to exist in the repository before the web interface can be used to draft contracts, but we hope that as lawyers from those fields begin to use our system they will help populate the repository with the most common Action objects from these fields.

8.1.2 Drafting Using Python

We then evaluated our system's coverage on the same 93 actions and clauses drafted using our provided Python library. The Python method of contract drafting does not have the limitation of users not being able to create custom Action objects, so more coverage of the sample contracts was observed. Out of the 93 actions and clauses we evaluated on, we were able to accurately represent 77 of them for a 82.8% coverage rate. This is a significant improvement over the 65.5% rate found when evaluating on the web interface, showing that with the simple contribution of more Action objects to our back-end clause repository, the web-interface could drastically improve its coverage rate. Again, this will come with more and more users using our system.

8.1.3 Limitations

In addition to the web interface limitation of not being able to draft custom Actions stated in Section 8.1.1, other limitations of our system were the reason for less than full coverage of the example contracts we tested on. In both the web-interface and Python methods, we were unable to represent three types of clauses using our system:

The first type of clause was that which allowed parties listed in the contract to specify mutable roles. Consider the following example, taken from clause 6 in Appendix A.2:

> **6. ROLES AND RESPONSIBILITIES.** The Camden Coalition and the CFS agree to provide notice to all other parties of changes to any of the roles listed below:
>
> c) The following staff member(s) is/are assigned to roles related to the proper management, processing, and distribution of data under this Agreement:
> [John Doe, Lead Data Analyst]
>
> d) Principal Investigators or Lead Data Analyst(s) will abide by any protocols and procedures established by the governance structure developed by the Camden Coalition, CFS, and all other parties.

This clause allows for the party John Doe to change roles during the lifetime of the legal agreement, as long as proper notification is made by two other parties, the Camden Coalition and the CFS.

Due to our Python representation of a clause as well as a contract's immutability after being deployed on the blockchain, we are unable to accurately represent or allow changing roles for parties during the lifetime of a smart contract. One solution would be to specify a generic "Lead Data Analyst" party when drafting the contract and then maintain an off-chain list of all names attached to that role, although this would decrease some important information stored in the deployed contract.

Another type of clause that our system's current implementation is unable to model is a clause which specifies proxies to perform an action. For example, a party might specify in a legal contract that in the case that the original party is unable to receive payment for services on a certain payment date, the payment is to be sent to a third-party that will be specified closer to the payment date. There is no way to accurately represent this clause in either our web interface or Python model, as our system assumes that all entities and specifications of the contract are known at the time of signing.

A final type of clause we are unable to represent is a clause which permits only explicitly specified actions, but not any others. Consider the following example, taken

from clause 1 in Appendix A.2:

> 1. **PURPOSE AND INTENDED USE OF DATA SHARING.** The data will be used for the following purposes:
>   (a) For inclusion in the Camden Promise Neighborhood case management system.
>   (b) For research and evaluation purposes.

Though not explicitly stated, this clause implies that using the data for any purpose other than the two listed is a breach of contract. Our Python representation of clauses only allows for checking if specific actions occur, rather than an infinite set comprising of any possible action except the two listed. One workaround to representing this clause would be to represent it all as one action, and then rely on arbiters during the adjudication process to vote on whether other uses occurred. The challenge with representing this clause is that the set of unpermitted actions is subjective, as some parties may consider some unlisted behaviors as permitted while other parties may not. Including this clause would make the interpretation of the final legal contract more debatable, which detracts from the intentions of our concrete code-like contract representation.

We hope that these limitations can be addressed in future work, which is discussed in Section 9.

8.2 Clause Re-use Efficiency

We evaluated our contribution of the clause repository by measuring the frequency of reuse or shared logic between clauses in the two sample contracts. After analyzing the 77 actions and clauses we were able to represent with our system, we found that only 17 distinct clauses and actions needed to be in the repository in order to represent all 77. Figure 8-1 shows the frequency that each distinct action and clause was reused in constructing the representation of these 77 actions and clauses. The top four most frequently used clauses and actions in our repository are labelled in the figure. The amount of reuse demonstrates the usefulness of our system's clause repository, as

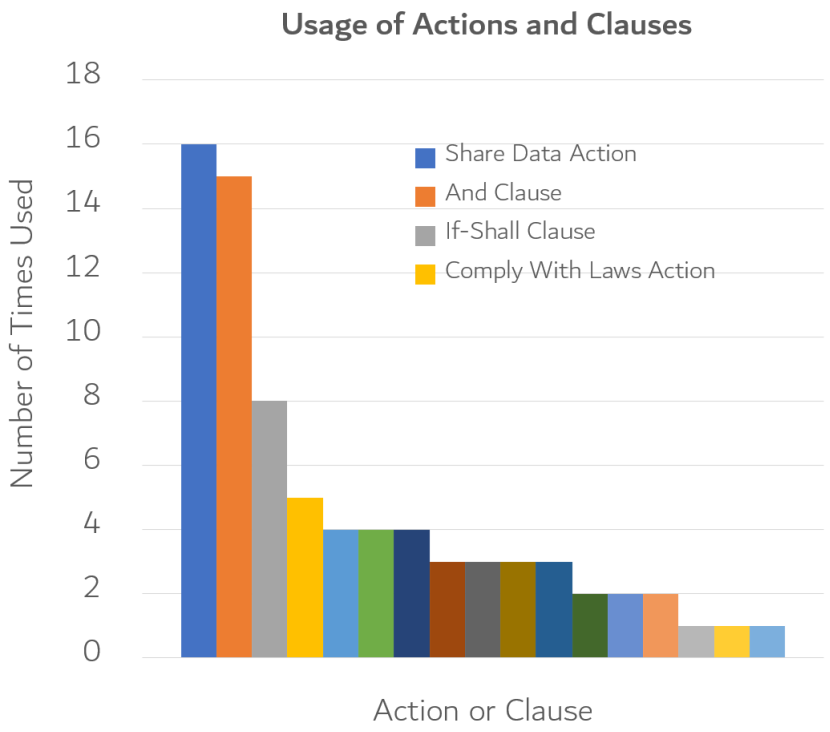

Figure 8-1: Frequency of Clause and Action reuse from the built-in repository when evaluating on the 77 actions and clauses our system is able to represent.

each clause that does not have to be written from scratch will save the end-user vast amounts of time and effort compared to drafting legal contracts by hand.

8.3 Formal Verification

Using the same two data sharing agreements, we formally verified 203 individual proofs concerning these actions and clauses. The Z3 Prover achieved a 100% proof validation success rate, and required fewer than two seconds to check all proofs sequentially when running on a low-power dual-core tablet computer. The KEVM verification also passed all proofs in around the same time frame.

These results show that the logical implementations of each clause we evaluated and of the generated legal agreements as a whole are valid. Timing wise, the proofs ran extremely fast, and will not be an issue even for extremely lengthy legal agreements.

8.4 Blockchain Deployment and Adjudication

To evaluate our system's blockchain deployment and adjudication capabilities, we manually tested using Remix and MetaMask as described in Section 7.1. We tested by deploying on both the Rinkeby testnet and the live Ethereum blockchain. Deploying a contract on Rinkeby took on average ten seconds to achieve the necessary six node confirmations, whereas deploying on the real Ethereum blockchain took on the magnitude of minutes depending on the length of the contract, the network load, and the miner incentive fee paid. The transaction cost was a few U.S. dollars per deployment, which is negligible compared to the legal cost savings that users of our system will gain.

To test querying the deployed contract, we successfully used Remix and MetaMask to interact with the deployed contract. Some successful queries performed were the checking of clause states, checking of action states, and seeing which arbiters were listed on a contract. Querying a contract should not change a contract's state at all, so after each query the contract was checked to see if it was still fulfilled.

To evaluate adjudication, we manually submitted six breach requests, three for each sample contract that we tested on. The breach requests were set to payment requests of 0 Ether tokens. We then had arbiter accounts linked via MetaMask vote on the action state variables contained in these breach requests. The three different breach requests tested the cases of having no arbiter quorum, quorum but no majority for changing the validity of the contract, and the case where a breach was decided to have actually occurred. In the first two cases, the breach votes resulted in no changes to the contract state as a whole and no payment was made to the breach requester's account. In the last case, the contract was successfully marked as breached by evaluating to `False` after tallying votes, and a payment transaction could be seen incoming to the breach requester's Ethereum account.

Because each contract query and transaction on the blockchain requires a small fee to be paid, it was hard to scale tests for many different contract queries and breach votes. For future work, exhaustive tests should be run on the Rinkeby testnet, but

the manual tests we performed show that the deployment and adjudication processes function as intended with no observed errors. Each transaction took on the magnitude of minutes to return a result, so there is no issue in terms of delays in either contract deployment or adjudication voting.

Chapter 9

Future Work

In terms of future work, there are many improvements aesthetically, functionally, and evaluation-wise that we hope to add to our project.

On the aesthetics side, we wish to add drag-and-drop functionality to the front-end web interface. This will improve the ease-of-use of our system even further than the current point-and-click implementation. Users would be able to simply drag-and-drop clauses into the contract draft, as well as rearrange the order of clauses in the contract with simple mouse movements. Users will also be able to drag parties and other frequent parameter values to the selected clauses to fill in the parameter placeholders. Improving ease-of-use is an important goal of our system as it will increase its adoption in the legal field. A higher adoption rate will not only mark a higher success of our system, but also improve its functionality because the public clause repository is crowd-sourced from users.

In terms of functionality, some modifications to our Python higher-level language can introduce support for more types of clauses and increased flexibility of existing clauses. While reading through sample legal contracts, we noticed that a few of them contained clauses where a party of the contract could designate a proxy to perform an action should the original party not be able to. For example, in a goods exchange contract, the party receiving the goods might specify a third-party to handle delivery of the goods should the primary party be unable to receive them. Our current Python representation of clauses does not support this. Another feature we wish to add is

the ability for party parameters in clauses to represent roles instead of only specific parties. For example, in a healthcare data sharing contract, if there are multiple hospitals listed as parties on the contract, a single clause should be able to reference all the hospitals collectively rather than listing each of them separately. The list of hospitals should also be able to change during the lifetime of the contract. Both of these improvements to the contract drafting process will allow our system to support additional clauses, increasing the robustness of our system and its adoption potential by lawyers.

In terms of functionality on the blockchain, each smart contract that our system currently deploys only tracks its expiration time and the current agreement between parties and arbiters on the truth value of each Action that the contract contains. For future work, the smart contracts can be improved to perform automatic fulfillment of some contract clauses. For example, in a contract requiring periodic payments, future iterations of our smart contracts might automatically request Ethereum blockchain-based payments for each payment installment. For legal contracts relating to trust funds or escrowing money, a smart contract modelling this legal agreement might require the parties on the blockchain to send these escrow funds to the contract address, after which the smart contract can manage the funds until they are supposed to be distributed. Again, these improvements will add functionality to our system, hopefully making lawyers more willing to adopt our framework.

Evaluation-wise, the true test of our system's success will be its usability and helpfulness to our target audience of legal contract writers. For future work, we hope to user test our system with legal professionals, and ask them to assess whether our system can help speed up and ease their contract drafting process. We also hope to showcase the formal verification and adjudication features our system provides, to see how valued our contract managing system is. With user-test feedback, we can determine what additional features and improvements will be most suited for our system.

For this thesis, our scope was limited to data sharing contracts, as addressed in Section 1.1. For future work, we hope to evaluate our system on other types of

legal contracts to see whether its functionality coverage will remain the same. This additional evaluation can also reveal additional clause types that our system might need to support.

Chapter 10

Conclusion

The current state of technology in legal contract drafting lags far behind some of the fields that these contracts are meant to service. This thesis successfully produced a system to digitize paper-based legal contracts and deploy them onto the Ethereum blockchain, where they can be easily managed, audited, and adjudicated. Although we focused on data sharing contracts when evaluating our system, the framework we have developed is applicable to any type of legal contract.

By adopting this system, lawyers will easily be able to draft legal contracts using either a point-and-click front-end or an easy to learn Python library. Both of these methods are connected to a built-in crowd-sourced clause repository, promoting the efficiency of clause reuse between contracts. Once written, our system automatically handles the complete back-end process to verify, compile, and deploy the digital contract onto the Ethereum blockchain in the form of a smart contract. The translation of the contract language from its front-end natural language representation to Ethereum bytecode is guaranteed in accuracy by both Z3 and KEVM mathematical proofs in two separate steps of the compilation process.

After contract deployment, users can easily use the front-end to query the smart contract at any time for details on the agreement conditions or to check whether the contract still holds. During the lifetime of a contract, if any party suspects that another party has breached the contract, they can request predefined arbiters to vote on whether the breach has occurred and a proposed monetary settlement. Arbiters

can then conduct both on-chain and off-chain investigations to determine the existence of a breach. Once the voting period passes, the smart contract governing the legal agreement will compile the arbiter votes and automatically issue monetary penalties should a breach been determined to have occurred.

Evaluating our system on real-world legal contracts, we found that the contribution of a crowd-sourced clause repository drastically sped up the contract drafting process, as many clauses are shared between contracts governing the same field. This observation held no matter if the user developed contracts using the front-end web interface or our provided Python library. Testing the smart contracts on the Rinkeby Ethereum testnet, we found that there was minimal delay between when the user submitted a legal contract and when the smart contract became confirmed on the blockchain. The adjudication process was tested using manual voting on the Ethereum testnet. Future iterations of this project will involve automatic and exhaustive testing of the voting process.

Overall, the system we developed in this thesis provides lawyers with a fresh, more technologically advanced, and more efficient way to draft, track, manage, and adjudicate legal contracts. Adoption of it will drastically decrease the cost and time needed for creating the important legal contracts that govern almost all formal interactions in today's interconnected world.

Appendix A

Example Legal Contracts

This appendix section contains a subset of example legal contracts we used in developing and evaluating our system. They are presented in PDF form in their original text.

A.1 Simple Data Sharing Agreement

The following is a simple, one page data sharing agreement between The Rhode Island Department of Health and The Providence Plan. This was the very first contract our system was evaluated on, and the first clauses that we tried to model with both our front-end interface and Python library.

Sample Data-Sharing and Usage Agreement

Rhode Island Department of Health and the Providence Plan

This agreement establishes the terms and conditions under which the Rhode Island Department of Health (RIDOH) and The Providence Plan (TPP) can acquire and use data from the other party. Either party may be a provider of data to the other, or a recipient of data from the other.

1. The confidentiality of data pertaining to individuals will be protected as follows:
   a. The data recipient will not release the names of individuals, or information that could be linked to an individual, nor will the recipient present the results of data analysis (including maps) in any manner that would reveal the identity of individuals.
   b. The data recipient will not release individual addresses, nor will the recipient present the results of data analysis (including maps) in any manner that would reveal individual addresses.
   c. Both parties shall comply with all Federal and State laws and regulations governing the confidentiality of the information that is the subject of this Agreement.
2. The data recipient will not release data to a third party without prior approval from the data provider.
3. The data recipient will not share, publish, or otherwise release any findings or conclusions derived from analysis of data obtained from the data provider without prior approval from the data provider.
4. Data transferred pursuant to the terms of this Agreement shall be utilized solely for the purposes set forth in the "Partnership Agreement".
5. All data transferred to TPP by RIDOH shall remain the property of RIDOH and shall be returned to RIDOH upon termination of the Agreements.
6. Any third party granted access to data, as permitted under condition #2, above, shall be subject to the terms and conditions of this agreement. Acceptance of these terms must be provided in writing by the third party before data will be released.

IN WITNESS WHEREOF, both the Rhode Island Department of Health, through its duly authorized representative, and The Providence Plan, through its duly authorized representative, have hereunto executed this Data Sharing Agreement as of the last date below written.

_____

Medical Director, Division of Family Health
Rhode Island Department of Health

Date: _____

_____

Executive Director
The Providence Plan

Date: _____

Source: www2.urban.org/nnip/ds_sample.html

A.2 Complex Data Sharing Agreement

The following is a more complex, multi-page data sharing agreement between the Camden Coalition of Healthcare Providers, the Center for Family Services, and a customizable third party. Only the first five pages are replicated, as these contain the clauses of the contract. The remaining pages of the contract are reserved for signatures and appendices.

Master Data Sharing Agreement

This Master Data Sharing Agreement ("Agreement") is entered by and among the **Camden Coalition of Healthcare Providers** ("Camden Coalition"), located at 800 Cooper St. 7th Floor, Camden, NJ 08102, **Center for Family Services** ("CFS") located at 584 Benson St., Camden, NJ 08103, and _____, that provides Solutions and support to the Camden Promise Neighborhood, and is located at _____, collectively "Parties". This Agreement shall be effective as of May 1, 2017 ("Effective Date").

1. **PURPOSE AND INTENDED USE OF DATA SHARING.** The purpose of this Agreement is to facilitate the submission of data to the Camden Coalition for the creation, use, and maintenance of a system of integrated social, health, and educational data concerning citizens of Camden City, Southern New Jersey, and the broader state of New Jersey in order to obtain a more complete understanding of the service needs, service gaps, and impact of services ("Camden ARISE"). The data will be used for the following purposes:

    a. For inclusion in the Camden Promise Neighborhood case management system, which is a component of Camden ARISE and is used by Camden Promise Neighborhood Solutions ("Solutions") to coordinate, manage, track, and report on the services provided by all or some of the other Camden Promise Neighborhood Solutions to individuals and families. [NAME] agrees to allow the disclosure of personally identifiable information to the entities shown in **Exhibit A** to this Agreement provided that (i) appropriate consent or authorization, if required for use, has been obtained from the individual or the individual's parent or guardian; and (ii) a role-based access control is assigned as specified in **Exhibit A.**

    b. For research and evaluation purposes to study and report on the impact of services provided by Solutions and other organizations contributing data for inclusion in Camden ARISE ("Data Contributor(s)") to citizens of Camden City and Southern New Jersey on individuals and families in the area and to study and report on factors related to service provision, assessment of need, and topics relevant to innovating new approaches to benefit the citizens of Camden City and Southern New Jersey.

2. **DEFINITIONS**

    a. Camden ARISE - A system of integrated social, health, and educational data concerning citizens of Camden City, Southern New Jersey, and the broader state of New Jersey
    b. Camden Promise Neighborhood – A project led by CFS which aims to create a comprehensive pipeline of services and a cradle through college to career path leading to positive change for the children and families
    c. Confidential Information – the [NAME] Data Set, Primary Data Set(s), the Case Management Data Set, Camden ARISE data or any part thereof
    d. Data Contributor(s) – Organization(s) that provide data for inclusion in Camden ARISE.
    e. Case Management Data Set – A data set comprised of data from all Solutions.

f. Solutions– Organizations which form a part of the Camden Promise Neighborhood and who (1) have active data sharing agreements with the Camden Coalition and; (2) contribute data for use in the Camden Promise Neighborhood case management system and Camden ARISE. The Parties anticipate that these organizations/programs will include ***insert names of partner organizations here***. The Solutions that form a part of the Camden Promise Neighborhood may change during the term of this Agreement. The Coalition will maintain an active list of Solutions which will be made available upon request.
g. [NAME] Data Set – Data set comprised only of data provided by [NAME]
h. Primary Data Sets – Data sets provided by Solutions other than [NAME]

3. TERM AND TERMINATION.

a. **Term.** This Agreement shall be in effect for five years from the Effective Date and thereafter shall renew for one year terms until terminated in accordance with 2b.

b. **Termination.** This Agreement may be terminated by any party with thirty (30) days written notice to the other parties. In the event of the termination of the Agreement, the Parties shall, upon request, (1) delete all data containing individually identifying information obtained under this Agreement; and (2) certify in writing within ten (10) business days that all copies of the data stored on cloud-based or local servers, backup servers, backup media, or other media have been permanently erased or destroyed.

4. DESCRIPTION OF DATA

a. [NAME] Data Set. [NAME] shall share with CFS and the Camden Coalition data according to the specifications set forth in Exhibit B. The data shared shall be limited to the data elements mutually agreed upon by the parties. The designated representative of each party will agree on specific data elements and data and record and file formats.

b. Agreements with Solutions. Solutions will contribute and be given access to the Promise Neighborhood Case Management System and Camden ARISE after executing data sharing agreements with the Camden Coalition that contain substantially similar provisions as those contained in this Agreement.

c. Other Data Sources Eligible for Linkage.

i. Each Solution will contribute a data set that shall be made part of the Promise Neighborhood Case Management System. The [NAME] Data Set will be linked with Primary Data Sets from all Solutions that choose to participate in the Case Management System to create a Case Management Data Set. The Primary Data Set from each Solution and/or the Case Management Data Set may be linked with data from Data Contributors that is available in Camden ARISE.

ii. A data dictionary containing a list of all data elements in the [NAME], Primary, and Case Management Data Set will be available to Solutions and Data Contributors upon request.

d. Adding to the [NAME] Data Set. Subject to applicable law, and provided there is mutual agreement of the Parties to this Agreement, content of the [NAME] Data Set may also include other records mutually agreed upon by the Camden Coalition, CFS and [NAME] to be necessary and appropriate for the proper execution of this Master Data Sharing Agreement or any approved Data Use Agreement executed under this Master Data Sharing Agreement.

5. CUSTODIAL RESPONSIBILITY AND DATA STEWARDSHIP.

a. The Camden Coalition and CFS will be joint Custodians of the raw and linked data sets and will be responsible for the observance of all conditions for use and for establishment and maintenance of security arrangements as specified in this Agreement to prevent unauthorized use.

b. Unless otherwise stated or modified in this Agreement, the Camden Coalition and CFS shall manage, link, and store data as specified in **Exhibit C** to this Agreement.

c. [NAME] will not use Confidential Information for any purpose other than the purposes specified in this agreement. The Camden Coalition and CFS will fully cooperate with [NAME] in the event that an adult individual or the parent or guardian of a minor under 18 years old requests the opportunity to review his/her personally identifiable information disclosed to the Camden Coalition and/or CFS by [NAME] or wishes to revoke their consent to data sharing with the Camden Coalition and/or CFS. [NAME] will notify the Camden Coalition and CFS in the event it obtains written consent for data sharing with the Camden Coalition and CFS, a revocation of consent to share data with the Camden Coalition and CFS, or a request to review personally identifiable information stored by the Camden Coalition and CFS from an adult or parent/guardian of a minor under 18 years old.

d. [NAME] will not release any data it receives as a result of its participation in this Agreement to any third parties not specifically authorized to have access to such data under this Agreement.

6. ROLES AND RESPONSIBILITIES. The Camden Coalition and CFS agree to provide appropriate staff support to execute its data stewardship, data management, custodial responsibilities, and analysis under this Agreement. The Camden Coalition and CFS agree to provide notice to [NAME] within thirty (30) days when additional Solutions or Data Contributors join Camden ARISE. [NAME] agrees to provide appropriate staff support to create and transmit the [NAME] Data Set to the Camden Promise Neighborhood case

management system, which is a component of Camden ARISE, as specified in **Exhibit C** to this Agreement. The Camden Coalition and CFS will notify [NAME] and all other Solutions' staff member(s) of changes to any of the roles listed below within 30 days of receiving notice of the change. Notification via electronic mail to [NAME] and all other Solutions' staff member(s) will be sufficient.

a. The following Camden Coalition staff member(s) is/are assigned to roles related to the proper management, processing, and distribution of the data under this Agreement:

   Role Name, Title, and Organization Contact information

b. The following CFS staff member(s) is/are assigned to roles related to the proper management, processing, and distribution of the data under this Agreement:

   Role Name, Title, and Organization Contact information

c. The following [NAME] staff member(s) is/are assigned to roles related to the proper management, processing, and distribution of the data under this Agreement:

   Role Name, Title, and Organization Contact information

d. Principal Investigator(s) or Lead Data Analyst(s) are individuals conducting research and evaluation, who will be vetted and approved through a governance structure developed by the Camden Coalition, CFS and all Solutions. All Principal Investigators or Lead Data Analyst(s) are Users, as defined in Exhibit C. Principal Investigators or Lead Data Analyst(s) will abide by any protocols and procedures established by the governance structure developed by the Camden Coalition, CFS and all Solutions. In addition, Principal Investigator(s) or Lead Data Analyst(s) may involve one or more researchers or student research assistants, working under the close supervision of the Principal Investigator(s) or Lead Data Analyst(s), to assist in a support role with various tasks under this Agreement and any approved data use agreements executed under this Agreement. Principal Investigator(s) will require any of its researcher(s) and/or student research assistants that create, receive, maintain, or transmit data to provide reasonable assurance, evidenced by written contract, that researcher(s) and/or student research assistants will comply with the same privacy and security obligations as Principal Investigator(s) with respect to the data.

7. PERMISSIBLE DATA USE, LINKING AND SHARING UNDER THIS AGREEMENT. All data shared as part of this Agreement and/or any related data use agreements remains the property of the supplying Solution. This Agreement represents and warrants further that data covered under this Agreement shall not be disclosed, released, revealed, showed, sold, rented, leased, or loaned to any person or organization except (1) to other Solutions or Data Contributors; (2) as specified herein; (3) as approved in an executed data use agreement; (4) as otherwise authorized in writing by [NAME] or; (5) as required by law. Access to the data covered by this Agreement shall be limited to the minimum number of individuals necessary to achieve the purpose stated in this section and to those individuals on a need-to-know basis only.

a. Authorized Linkage and Data Transfers for Program and Site Management. Access to

limited identifiable individual-level data will be restricted to a tightly controlled data stream of "need to know" users at end service points and carefully selected organizational administrators (as specified in Exhibits A and C to this Agreement). Only records with a signed consent or authorization agreement will be transmitted for this purpose.

b. Authorized Linkage and Data Transfers for Research and Evaluation. Uses of Confidential Information for research and evaluation shall be limited to the Principal Investigator(s)/Lead Data Analyst(s) with whom a signed data use agreement exists. Only de-identified data shall be released for this purpose. The Camden Coalition will manage all identifiable data contained with the Case Management Data Set as well as other Camden ARISE data. CFS will support the Camden Coalition in the management of the identifiable, linked data, as part of the broader data administration needs of the Camden Promise Neighborhood initiative. The Camden Coalition will establish external data sharing protocols with approval from the Solutions or Data Contributor(s). A data governance board, which will be established by the Camden Coalition, CFS, and the Solutions, will decide upon the process for de-identification. Data use agreements with Principal Investigator(s)/Lead Data Analyst(s) will include information about the de-identification process.

8. **NO WARRANTY FOR DATA OR LINKAGE QUALITY.** Both the accuracy of record linkage and the utility of administrative data for research and analytical purposes are dependent on the quality and consistency of the source data. Although the Camden Coalition and CFS will use reasonable efforts to promote accurate record linkage and the creation of appropriate data sets for analysis, no warranty is made as to the achievement of any particular match rate nor as to the ultimate accuracy or utility of any data contributed under this Agreement.

9. **PUBLICATION AND DISSEMINATION OF RESULTS.** The Camden Coalition and CFS shall provide [NAME] copies of written reports, analysis, or visuals produced or derived in whole or in part from [NAME]'s data prior to public dissemination. [NAME] will also be provided with final copies of these publications. Copies shall be submitted to the [NAME]'s primary contact for the administration of this Agreement as specified in Section 6 to this Agreement.

10. **MODIFICATION.** The Parties may amend this Agreement by mutual consent, in writing, at any time. This Agreement may be terminated by either party with thirty (30) days written notice.

11. **SIGNATURES.** By the signatures of their duly authorized representatives below, the Camden Coalition, CFS, and [NAME] agree to all of the provisions of this Master Data Sharing Agreement.

[Signatures on following page]

Appendix B

Select Source Code Files

This appendix section contains select source code files that are instrumental to the functioning and understanding of our project.

B.1 Python Representation of a Legal Contract

The following is an example of a contract written in Python using our Python library. It is a contract between two fake entities named "Department of Health" and "Research University," with the sole arbiter being the United States Federal Court. The clauses represented in Python are based on the clauses in the simple data sharing agreement from Appendix A.1. Import statements have been removed for brevity.

```
doh = Party(name="Department of Health",
            address="0x2B5634C42055806a59e9107ED44D43c426E58258")
ru = Party("Research University",
           address="0x37396B8A6242DD2ab08Db4160dD93Bb9580bdBA5")
parties = [doh, ru]
arbiters = [
    Arbiter(name="United States Federal Court",
            address="0x37396B8A6242DD2ab08Db4160dD93Bb9580bdBA5")]
third_party = "any third party"
the_public = "the public"

register_serializable(Party, "party")
register_serializable(Arbiter, "arbiter")
register_serializable(IfMayClause, "if_may_clause")
register_serializable(AndClause, "and_clause")
register_serializable(NegatedClause, "negated_clause")
register_serializable(IfShallClause, "if_shall_clause")
register_serializable(OrClause, "or_clause")
register_serializable(OwnDataAction, "own_data_action")
register_serializable(GrantApprovalAction, "grant_approval_action")
register_serializable(ShareDataAction, "share_data_action")
register_serializable(ReturnDataAction, "return_data_action")
register_serializable(ComplyWithLawsAction, "comply_with_laws_action")
register_serializable(TerminationAction, "termination_action")
register_serializable(ExecuteAgreement, "execute_agreement_action")

clause1 = AndClause(
    NegatedClause(OrClause(
        ShareDataAction(
            parties=parties,
            arbiters=arbiters,
            user=ru,
            data_action="release the names of individuals or
             ↪  information that could be linked to an individual",
            data_audience=the_public),
        ShareDataAction(
            parties=parties,
            arbiters=arbiters,
            user=ru,
            data_action="release results of data analysis (including
             ↪  maps) that would reveal the identity of individuals",
            data_audience=the_public),
    )),
    NegatedClause(OrClause(
        ShareDataAction(
```

```
            parties=parties,
            arbiters=arbiters,
            user=ru,
            data_action="release individual addresses",
            data_audience=the_public),
        ShareDataAction(
            parties=parties,
            arbiters=arbiters,
            user=ru,
            data_action="release results of data analysis (including
             ↪ maps) that would reveal individual addresses",
            data_audience=the_public),
    )),
    AndClause(
        ComplyWithLawsAction(ru, "Federal", arbiters, parties,
         ↪ default=True),
        ComplyWithLawsAction(ru, "State", arbiters, parties,
         ↪ default=True),
        ComplyWithLawsAction(doh, "Federal", arbiters, parties,
         ↪ default=True),
        ComplyWithLawsAction(doh, "State", arbiters, parties,
         ↪ default=True)
    )
)
third_party_use_action = ShareDataAction(
    parties=parties,
    arbiters=arbiters,
    user=ru,
    data_action="releases",
    data_audience=third_party)

clause2 = IfMayClause(
    if_clause=GrantApprovalAction(
        parties=parties,
        arbiters=arbiters,
        granter=doh,
        grantee=ru,
        approval_type="third party release"),
    may_clause=third_party_use_action
)

clause3_if_clause = GrantApprovalAction(
    parties=parties,
    arbiters=arbiters,
    granter=doh,
```

```
    grantee=ru,
    approval_type="public release")
clause3_may_clause_shares = ShareDataAction(
    parties=parties,
    arbiters=arbiters,
    user=ru,
    data_action="shares analyses or findings",
    data_audience=the_public)
clause3_may_clause_publishes = ShareDataAction(
    parties=parties,
    arbiters=arbiters,
    user=ru,
    data_action="publishes analyses or findings",
    data_audience=the_public)
clause3_may_clause_release = ShareDataAction(
    parties=parties,
    arbiters=arbiters,
    user=ru,
    data_action="otherwise releases analyses or findings",
    data_audience=the_public)
clause3 = IfMayClause(
    if_clause=clause3_if_clause,
    may_clause=OrClause(
        clause3_may_clause_shares,
        clause3_may_clause_publishes,
        clause3_may_clause_release
    )
)

clause4 = NegatedClause(Action(
    parties=parties,
    arbiters=arbiters,
    name="misuse",
    description="Recipient uses data for purposes other than agreed
     ↪  upon"))

clause5 = AndClause(
    OwnDataAction(doh, "data", arbiters=arbiters, default=True),
    IfShallClause(
        if_clause=TerminationAction(
            actor=None, arbiters=arbiters, parties=parties),
        shall_clause=ReturnDataAction(ru, doh, item="data",
         ↪  arbiters=arbiters, parties=parties))
)
```

```
clause6 = IfShallClause(
    if_clause=third_party_use_action,
    shall_clause=ExecuteAgreement(Party(third_party,
     ↪  address="0x37396B8A6242DD2ab08Db4160dD93Bb9580bdBA5"),
                                    "this agreement in writing",
                                     ↪  arbiters=arbiters,
                                     ↪  parties=parties)
)

document = Document((clause1, clause2, clause3, clause5, clause6))
```

B.2 JSON Representation of a Legal Contract

The following is an excerpt from an example of a contract represented using our JSON specification. The contract used to generate this JSON file was the simple data sharing agreement from Appendix A.1. The "`actions`" and "`clauses`" lists have been truncated for brevity.

```
{
    "parties": [
        {
            "name": "ridh",
            "long_name": "Rhode Island Department of Health",
            "address": "0x2B5634C42055806a59e9107ED44D43c426E58258"
        },
        {
            "name": "pp",
            "long_name": "The Providence Plan",
            "address": "0x37396B8A6242DD2ab08Db4160dD93Bb9580bdBA5"
        }
    ],
    "arbiters": [
        {
            "name": "aha",
            "long_name": "American Public Health Association",
            "address": "0xBaa6c254dD5498A608186E9De1057E21128a4836"
        },
        {
            "name": "uscourts",
            "long_name": "The Federal Court for the District of Rhode
            ↪  Island",
            "address": "0x0c31411a63419b0a5ebed2d4e3e6cce1e9ac819d"
        },
        {
            "name": "cdc",
            "long_name": "Centers for Disease Control and Prevention",
            "address": "0x689C56AEf474Df92D44A1B70850f808488F9769C"
        }
    ],
    "actions": [
        {
            "name": "name_release",
            "description": "Recipient releases individual names
            ↪  publicly",
            "parties": [
                "ridh",
                "pp"
            ],
            "arbiters": [
                "aha",
                "uscourts",
                "cdc"
            ],
```

```
            "default": false,
            "arbiter_quorum": 3
        },
        {
            "name": "pii_release",
            "description": "Recipient releases individual information
            ↪  publicly",
            "parties": [
                "ridh",
                "pp"
            ],
            "arbiters": [
                "aha",
                "uscourts",
                "cdc"
            ],
            "default": false,
            "arbiter_quorum": 3
        }
    ],
    "party_votes": {
        "provider_public_release": false,
        "recipient_public_release": {
            "ridh": false,
            "pp": true
        }
    },
    "arbiter_votes": {
        "recipient_public_release": {}
    },
    "clauses": [
        {
            "uri": "if_may_clause",
            "checksum": null,
            "data": {
                "if_clause": {
                    "uri": "action",
                    "checksum": null,
                    "data": {
                        "name": "provider_third_party"
                    }
                },
                "may_clause": {
                    "uri": "action",
                    "checksum": null,
```

```
                "data": {
                    "name": "recipient_third_party"
                }
            }
        }
    },
    {
        "uri": "if_may_clause",
        "checksum": null,
        "data": {
            "if_clause": {
                "uri": "action",
                "checksum": null,
                "data": {
                    "name": "provider_public_release"
                }
            },
            "may_clause": {
                "uri": "or_clause",
                "checksum": null,
                "data": {
                    "clauses": [
                        {
                            "uri": "action",
                            "checksum": null,
                            "data": {
                                "name": "recipient_public_release"
                            }
                        },
                        {
                            "uri": "action",
                            "checksum": null,
                            "data": {
                                "name": "recipient_publish"
                            }
                        },
                        {
                            "uri": "action",
                            "checksum": null,
                            "data": {
                                "name":
                                 ↪  "recipient_otherwise_publish"
                            }
                        }
                    ]
```

```
                    }
                }
            }
        }
    ]
}
```

B.3 Python to Vyper Compiler

The following is the code file used to compile contracts written in our provided Python library to Vyper code that can then be further compiled into Ethereum bytecode for deployment on the Ethereum blockchain.

Note that although the syntax highlighting might look incorrect, the dark red text is a lengthy multi-line string that this code is writing to a new file containing the Vyper code.

```
from typing import TYPE_CHECKING, TextIO, List, Set, Dict
from datetime import timedelta

from action import Action
from clauses.and_clause import AndClause
from .vyper import Vyper, VYPER_PREFIX

if TYPE_CHECKING:
    from document import Document


# contract_duration: used to set self.expiry
# pre_voting_duration: after a dispute is created, the minimum amount
 ↪  of time before arbiters can vote
# min_evidence_submission_duration: how long, before voting opens, the
 ↪  parties will be able to submit evidence.
#    As such, no actions can be added to a dispute during this period
# voting_duration: how long, after the pre_voting_duration, the
 ↪  arbiters have to vote. No new evidence will be
#    considered during this period
def py2vy(document: 'Document', f: TextIO, contract_duration:
 ↪  timedelta, pre_voting_duration: timedelta,
          min_evidence_submission_duration: timedelta,
           ↪  voting_duration: timedelta) -> None:
    actions: Dict[Action, int] = document.actions_to_int
    clauses = document.clauses

    f.write(f"""#!/bin/vyper
struct DisputedAction:
    in_dispute: bool  # need a flag to differentiate an action not in
 ↪  dispute from one that is and simply has no votes
    voteCount: int128
    votesTrue: int128
    votesFalse: int128

struct Action:
    state: bool
    arbiter_quorum: int128
    action_hash: bytes32
    num_parties: int128  # One more than the maximum (meaningful)
 ↪  index of ordered_parties

struct Dispute:
    creation_time: timestamp
```

```
action_parties: map(int128, map(address, bool))  # first key = action,
 ↪  second = party, True if party is valid
action_ordered_parties: map(int128, map(int128, address))   # first
 ↪  key = action, second = index, value is the party
action_arbiters: map(int128, map(address, bool))  # first key =
 ↪  action; True if party is valid; false otherwise
action_party_votes: map(int128, map(address, bool))  # first key =
 ↪  action; defaults to default

dispute_counter: int128
disputes: map(int128, Dispute)  # key is dispute number, value is the
 ↪  deadline
disputes_disputed_actions: map(int128, map(int128, DisputedAction))
# first key = dispute number, second key = action number, third key =
 ↪  arbiter, value is true voted
disputes_disputed_actions_voters: map(int128, map(int128, map(address,
 ↪  bool)))

expiry: public(timestamp)

""")
    for i, clause in enumerate(clauses):
        actions_in_clause: Set[Action] = set()
        stack = [clause]
        while len(stack) > 0:
            current = stack.pop()
            for child in current.get_children():
                if isinstance(child, Action):
                    if child not in actions:
                        actions[child] = len(actions)
                    actions_in_clause.add(child)
                else:
                    stack.append(child)
    f.write(f"""
actions: Action[{len(actions)}]""")
    f.write(f"""

@public
def __init__():
    self.expiry = block.timestamp +
 ↪  {int(contract_duration.seconds)}""")

    for action, i in actions.items():
        f.write(f"""
```

```
    self.actions[{i}] = Action({{state: {action.default}, # name:
↪   {action.name
        }; description:
↪   {action.description.format(parties=action.parties) if
↪   action.description is not None else None}
                                arbiter_quorum:
↪   {action.arbiter_quorum},
                                action_hash: 0x{"0" * 64},  # TODO
                                num_parties:
↪   {len(action.parties)}}})""")
        for j, party in enumerate(action.parties):
            assert party.address is not None, f"party {party} must
             ↪  have a valid address"
            f.write(f"""
    self.action_parties[{i}][{party.address}] = True
    self.action_ordered_parties[{i}][{j}] = {party.address}
    self.action_party_votes[{i}][{party.address}] = {action.default}
                                                """)
        for arbiter in action.arbiters:
            assert arbiter.address is not None, f"arbiter {arbiter}
             ↪  must have a valid address"
            f.write(f"""
    self.action_arbiters[{i}][{arbiter.address}] = True""")

    f.write(f"""
    self.dispute_counter = 0""")

    var_names: List[str] = []
    clause_to_var_name: Dict[Vyper, str] = {}
    lines: List[str] = []
    contract_clause = AndClause(*document.clauses)
    contact_clause_name = "c"
    for line in contract_clause.generate_vyper(actions):
        var_name = line.var_name.replace(VYPER_PREFIX,
         ↪  contact_clause_name)
        var_value = line.var_value.replace(VYPER_PREFIX,
         ↪  contact_clause_name)
        var_names.append(var_name)
        line_eval = line.evaluable
        assert isinstance(line_eval, Vyper)
        clause_to_var_name[line_eval] = var_name
        lines.append(f"{var_name}: bool = {var_value}")
    # reverse the ordering, since the output is in reverse ordering
    var_names.reverse()
```

```
    clause_var_to_i: Dict[str, int] = {}  # One clause / variable per
    ↪ line
    for var_name in var_names:
        clause_var_to_i[var_name] = len(clause_var_to_i)
    f.write(f"""

@public
def computeContract() -> bool[{len(lines)}]:
    assert block.timestamp <= self.expiry
    """)
    for line_str in lines:
        f.write(f"""
    {line_str}""")
    f.write(f"""

    return [""")
    for var_name in var_names:
        f.write(f"""
    {var_name},""")
    f.write(f"""
    ]
        """)

    f.write(f"""

# First step of the dispute process:
# Create a dispute. Anyone can do this. This just creates a deadline
↪ and assigns an ID
@public
def createDispute():
    self.disputes[self.dispute_counter].creation_time =
↪ block.timestamp
    self.dispute_counter += 1

# Second step of the dispute process:
# Bind actions to the dispute. Any party can do this
@public
def disputeAction(dispute_id: int128, action_id: int128):
    assert self.action_parties[action_id][msg.sender]  # assert that
↪ the originator is a party
    assert 0 <= dispute_id and dispute_id < self.dispute_counter  #
↪ assert the dispute exists
    dispute: Dispute = self.disputes[dispute_id]
    assert  block.timestamp +
↪ {int(min_evidence_submission_duration.seconds)
```

```
        } < dispute.creation_time +
↪ {int(pre_voting_duration.seconds)} # assert there is sufficient
↪ time
    assert 0 <= action_id and action_id < {len(actions)}  # assert
↪ that the action exists

    self.disputes_disputed_actions[dispute_id][action_id].in_dispute =
↪ True

# Third step is voting
@public
def vote(dispute_id: int128, action_id: int128, side: bool):
    assert 0 <= dispute_id and dispute_id < self.dispute_counter  #
↪ assert dispute_id is valid
    dispute: Dispute = self.disputes[dispute_id]
    assert dispute.creation_time + {int(pre_voting_duration.seconds) +
                                    int(voting_duration.seconds)} <=
↪ block.timestamp  # assert voting is open
    assert block.timestamp < dispute.creation_time +
↪ {int(pre_voting_duration.seconds) +

↪ int(voting_duration.seconds)}  # assert voting is not yet closed
    assert 0 <= action_id and action_id < {len(actions)}  # assert
↪ action id is valid
    disputed_action: DisputedAction =
↪ self.disputes_disputed_actions[dispute_id][action_id]
    assert disputed_action.in_dispute  # ensure that a party asked the
↪ arbiters to rule on this one
    assert self.action_arbiters[action_id][msg.sender]  # assert
↪ caller is an arbiter
    disputed_action.voteCount += 1
    if side:
        disputed_action.votesTrue += 1
    else:
        disputed_action.votesFalse += 1

# Fourth step is closing the dispute and registering the states
@public
def closeDispute(dispute_id: int128):
    assert 0 <= dispute_id and dispute_id < self.dispute_counter  #
↪ assert dispute_id is valid
    dispute: Dispute = self.disputes[dispute_id]
    assert dispute.creation_time + {int(pre_voting_duration.seconds) +
                                    int(voting_duration.seconds)} <=
↪ block.timestamp  # assert voting has closed
```

```
    # TODO would it be better to store exactly which actions are in
↪   this dispute?
    for action_id in range({len(actions)}):
        disputed_action: DisputedAction =
↪   self.disputes_disputed_actions[dispute_id][action_id]
        if not disputed_action.in_dispute:
            continue
        if not disputed_action.voteCount >=
↪   self.actions[action_id].arbiter_quorum:
            continue
        # TODO check that the parties do not agree
        if disputed_action.votesTrue > disputed_action.votesFalse:
            self.actions[action_id].state = True
        else:
            self.actions[action_id].state = False
""")
```

B.4 Vyper Smart Contract

The following is the compiled Vyper contract based on the simple data sharing agreement from Appendix A.1 that is outputted by our Python to Vyper compiler. The initialization of Action objects in the Vyper code has been truncated for brevity.

```
struct DisputedAction:
    in_dispute: bool  # flag to differentiate an action not in a
     ↪  dispute from one that is and simply has no votes
    voteCount: int128
    votesTrue: int128
    votesFalse: int128

struct Action:
    state: bool
    arbiter_quorum: int128
    num_parties: int128  # One more than the maximum index of
     ↪  ordered_parties

struct Dispute:
    creation_time: timestamp

# first key = action, second = party, True if party is valid
action_parties: map(int128, map(address, bool))
# first key = action, second = index, value is the party
action_ordered_parties: map(int128, map(int128, address))
# first key = action; True if party is valid; false otherwise
action_arbiters: map(int128, map(address, bool))
# first key = action; defaults to default
action_party_votes: map(int128, map(address, bool))

dispute_counter: int128
disputes: map(int128, Dispute)  # key is dispute number, value is the
 ↪  deadline
disputes_disputed_actions: map(int128, map(int128, DisputedAction))
# first key = dispute number, second key = action number, third key =
 ↪  arbiter, value is true voted
disputes_disputed_actions_voters: map(int128, map(int128, map(address,
 ↪  bool)))
expiry: public(timestamp)
actions: Action[18]

@public
def __init__():
    self.expiry = block.timestamp + 0

    # description: Research University
     ↪  (0x37396B8A6242DD2ab08Db4160dD93Bb9580bdBA5) release the names
     ↪  of individuals or information that could be linked to an
     ↪  individual with the public
    self.actions[0] = Action({state: False,
```

```
                                arbiter_quorum: 1,
                                num_parties: 2})
    self.action_parties[0]
        [0x37396B8A6242DD2ab08Db4160dD93Bb9580bdBA5] = True
    self.action_ordered_parties[0][0] =
    ↪  0x37396B8A6242DD2ab08Db4160dD93Bb9580bdBA5
    self.action_party_votes[0]
        [0x37396B8A6242DD2ab08Db4160dD93Bb9580bdBA5] = False
    self.action_parties[0]
        [0x2B5634C42055806a59e9107ED44D43c426E58258] = True
    self.action_ordered_parties[0][1] =
    ↪  0x2B5634C42055806a59e9107ED44D43c426E58258
    self.action_party_votes[0]
        [0x2B5634C42055806a59e9107ED44D43c426E58258] = False
    self.action_arbiters[0]
        [0x37396B8A6242DD2ab08Db4160dD93Bb9580bdBA5] = True

    # description: Research University
    ↪  (0x37396B8A6242DD2ab08Db4160dD93Bb9580bdBA5) release results
    ↪  of data analysis (including maps) that would reveal the
    ↪  identity of individuals with the public
    self.actions[1] = Action({state: False,
                                arbiter_quorum: 1,
                                num_parties: 2})
    self.action_parties[1]
        [0x37396B8A6242DD2ab08Db4160dD93Bb9580bdBA5] = True
    self.action_ordered_parties[1][0] =
    ↪  0x37396B8A6242DD2ab08Db4160dD93Bb9580bdBA5
    self.action_party_votes[1]
        [0x37396B8A6242DD2ab08Db4160dD93Bb9580bdBA5] = False
    self.action_parties[1]
        [0x2B5634C42055806a59e9107ED44D43c426E58258] = True
    self.action_ordered_parties[1][1] =
    ↪  0x2B5634C42055806a59e9107ED44D43c426E58258
    self.action_party_votes[1]
        [0x2B5634C42055806a59e9107ED44D43c426E58258] = False
    self.action_arbiters[1]
        [0x37396B8A6242DD2ab08Db4160dD93Bb9580bdBA5] = True
    # Initialization of Actions truncated for brevity

    self.dispute_counter = 0

@public
def computeContract() -> bool[32]:
    assert block.timestamp <= self.expiry
```

```
# description: {parties[0]} release the names of individuals or
 ↪  information that could be linked to an individual with the
 ↪  public
c_and0_and0_negatedClause_or0: bool = self.actions[0].state
# description: {parties[0]} release results of data analysis
 ↪  (including maps) that would reveal the identity of individuals
 ↪  with the public
c_and0_and0_negatedClause_or1: bool = self.actions[1].state
c_and0_and0_negatedClause: bool = c_and0_and0_negatedClause_or0 or
 ↪  c_and0_and0_negatedClause_or1
c_and0_and0: bool = not c_and0_and0_negatedClause
# description: {parties[0]} release individual addresses with the
 ↪  public
c_and0_and1_negatedClause_or0: bool = self.actions[2].state
# description: {parties[0]} release results of data analysis
 ↪  (including maps) that would reveal individual addresses with
 ↪  the public
c_and0_and1_negatedClause_or1: bool = self.actions[3].state
c_and0_and1_negatedClause: bool = c_and0_and1_negatedClause_or0 or
 ↪  c_and0_and1_negatedClause_or1
c_and0_and1: bool = not c_and0_and1_negatedClause
# description: {parties[0]} complies with all Federal laws and
 ↪  regulations governing this agreement
c_and0_and2_and0: bool = self.actions[4].state
# description: {parties[0]} complies with all State laws and
 ↪  regulations governing this agreement
c_and0_and2_and1: bool = self.actions[5].state
# description: {parties[0]} complies with all Federal laws and
 ↪  regulations governing this agreement
c_and0_and2_and2: bool = self.actions[6].state
# description: {parties[0]} complies with all State laws and
 ↪  regulations governing this agreement
c_and0_and2_and3: bool = self.actions[7].state
c_and0_and2: bool = c_and0_and2_and0 and c_and0_and2_and1 and
 ↪  c_and0_and2_and2 and c_and0_and2_and3
c_and0: bool = c_and0_and0 and c_and0_and1 and c_and0_and2
# description: {parties[0]} grants approval for {parties[1]} to
 ↪  third party release
c_and1_ifClause: bool = self.actions[8].state
# description: {parties[0]} releases with any third party
c_and1_mayClause: bool = self.actions[9].state
c_and1: bool = c_and1_ifClause or not c_and1_mayClause
# description: {parties[0]} grants approval for {parties[1]} to
 ↪  public release
```

```
    c_and2_ifClause: bool = self.actions[10].state
    # description: {parties[0]} shares analyses or findings with the
     ↪  public
    c_and2_mayClause_or0: bool = self.actions[11].state
    # description: {parties[0]} publishes analyses or findings with
     ↪  the public
    c_and2_mayClause_or1: bool = self.actions[12].state
    # description: {parties[0]} otherwise releases analyses or
     ↪  findings with the public
    c_and2_mayClause_or2: bool = self.actions[13].state
    c_and2_mayClause: bool = c_and2_mayClause_or0 or
     ↪  c_and2_mayClause_or1 or c_and2_mayClause_or2
    c_and2: bool = c_and2_ifClause or not c_and2_mayClause
    # description: {parties[0]} owns data
    c_and3_and0: bool = self.actions[14].state
    # description: The agreement is terminated
    c_and3_and1_ifClause: bool = self.actions[15].state
    # description: {parties[0]} returns data to {parties[1]}
    c_and3_and1_shallClause: bool = self.actions[16].state
    c_and3_and1: bool = (not c_and3_and1_ifClause) or
     ↪  c_and3_and1_shallClause
    c_and3: bool = c_and3_and0 and c_and3_and1
    # description: {parties[0]} releases with any third party
    c_and4_ifClause: bool = self.actions[9].state
    # description: {parties[0]} executes this agreement in writing
    c_and4_shallClause: bool = self.actions[17].state
    c_and4: bool = (not c_and4_ifClause) or c_and4_shallClause
    c: bool = c_and0 and c_and1 and c_and2 and c_and3 and c_and4

    return [c, c_and4, c_and4_shallClause, c_and4_ifClause, c_and3,
     ↪  c_and3_and1, c_and3_and1_shallClause, c_and3_and1_ifClause,
     ↪  c_and3_and0, c_and2, c_and2_mayClause, c_and2_mayClause_or2,
     ↪  c_and2_mayClause_or1, c_and2_mayClause_or0, c_and2_ifClause,
     ↪  c_and1, c_and1_mayClause, c_and1_ifClause, c_and0,
     ↪  c_and0_and2, c_and0_and2_and3, c_and0_and2_and2,
     ↪  c_and0_and2_and1, c_and0_and2_and0, c_and0_and1,
     ↪  c_and0_and1_negatedClause, c_and0_and1_negatedClause_or1,
     ↪  c_and0_and1_negatedClause_or0, c_and0_and0,
     ↪  c_and0_and0_negatedClause, c_and0_and0_negatedClause_or1,
     ↪  c_and0_and0_negatedClause_or0]

# First step of the dispute process:
# Create a dispute. Anyone can do this. This just creates a deadline
 ↪  and assigns an ID
@public
```

```
def createDispute():
    self.disputes[self.dispute_counter].creation_time =
    ↪ block.timestamp
    self.dispute_counter += 1

# Second step of the dispute process:
# Bind actions to the dispute. Any party can do this.
@public
def disputeAction(dispute_id: int128, action_id: int128):
    # assert that the originator is a party
    assert self.action_parties[action_id][msg.sender]
    # assert the dispute exists
    assert 0 <= dispute_id and dispute_id < self.dispute_counter
    dispute: Dispute = self.disputes[dispute_id]
    assert block.timestamp + 43200 < dispute.creation_time + \
        0  # assert there is sufficient time
    assert 0 <= action_id and action_id < 18  # assert that the action
    ↪ exists
    self.disputes_disputed_actions[dispute_id][action_id].in_dispute =
    ↪ True

# Third step is voting
@public
def vote(dispute_id: int128, action_id: int128, side: bool):
    # assert dispute_id is valid
    assert 0 <= dispute_id and dispute_id < self.dispute_counter
    dispute: Dispute = self.disputes[dispute_id]
    assert dispute.creation_time + 43200 <= block.timestamp  # assert
    ↪ voting is open
    assert block.timestamp < dispute.creation_time + \
        43200  # assert voting is not yet closed
    assert 0 <= action_id and action_id < 18  # assert action id is
    ↪ valid
    disputed_action: DisputedAction =
    ↪ self.disputes_disputed_actions[dispute_id][action_id]
    # ensure that a party asked the arbiters to rule on this one
    assert disputed_action.in_dispute
    # assert caller is an arbiter
    assert self.action_arbiters[action_id][msg.sender]
    disputed_action.voteCount += 1
    if side:
        disputed_action.votesTrue += 1
    else:
        disputed_action.votesFalse += 1
```

```
# Fourth step is closing the dispute and registering the states
@public
def closeDispute(dispute_id: int128):
    # assert dispute_id is valid
    assert 0 <= dispute_id and dispute_id < self.dispute_counter
    dispute: Dispute = self.disputes[dispute_id]
    assert dispute.creation_time + 43200 <= block.timestamp  # assert
     ↪  voting has closed
    for action_id in range(18):
        disputed_action: DisputedAction =
         ↪  self.disputes_disputed_actions[dispute_id][action_id]
        if not disputed_action.in_dispute:
            continue
        if not disputed_action.voteCount >=
         ↪  self.actions[action_id].arbiter_quorum:
            continue
        if disputed_action.votesTrue > disputed_action.votesFalse:
            self.actions[action_id].state = True
        else:
            self.actions[action_id].state = False
```

B.5 SMT2 Proofs

The following is an excerpt from an automatically generated `.smt2` file containing proofs to be formally verified by Z3. These proofs are based on the simple data sharing agreement from Appendix A.1. The majority of this file has been truncated for brevity.

```
(declare-fun bool_to_int (Bool) Int)
(assert (= (bool_to_int false) 0))
(assert (= (bool_to_int true) 1))
(define-fun max ((x Int) (y Int)) Int (ite (< x y) y x))
(declare-const a_0 Bool) ; defining action as bool
(assert (not a_0))
(declare-const a_0_party_0_vote Bool)
(declare-const a_0_party_0_voter Bool)
(assert a_0_party_0_voter)
(declare-const a_0_party_1_vote Bool)
(declare-const a_0_party_1_voter Bool)
(assert a_0_party_1_voter)
(declare-const a_0_party_num_true_votes Int)
(assert (= a_0_party_num_true_votes (+ (bool_to_int a_0_party_0_vote)
↪  (bool_to_int a_0_party_1_vote))))
(declare-const a_0_party_num_false_votes Int)
(assert (= a_0_party_num_false_votes (+ (bool_to_int (not
↪  a_0_party_0_vote)) (bool_to_int (not a_0_party_1_vote)))))
(declare-const a_0_party_num_voters Int)
(assert (= a_0_party_num_voters (+ (bool_to_int a_0_party_0_voter)
↪  (bool_to_int a_0_party_1_voter))))
(declare-const a_0_party_resolution_true Bool)
(assert (= a_0_party_resolution_true (and (= a_0_party_num_voters 2)
↪  (> a_0_party_num_true_votes 0) (= a_0_party_num_false_votes 0))))
(declare-const a_0_party_resolution_false Bool)
(assert (= a_0_party_resolution_false (and (= a_0_party_num_voters 2)
↪  (> a_0_party_num_false_votes 0) (= a_0_party_num_true_votes 0))))
(declare-const a_0_arbiter_0_vote Bool)
(declare-const a_0_arbiter_0_voter Bool)
(assert a_0_arbiter_0_voter)
(declare-const a_0_arbiter_num_true_votes Int)
(assert (= a_0_arbiter_num_true_votes (+ (bool_to_int
↪  a_0_arbiter_0_vote))))
(declare-const a_0_arbiter_num_false_votes Int)
(assert (= a_0_arbiter_num_false_votes (+ (bool_to_int (not
↪  a_0_arbiter_0_vote)))))
(declare-const a_0_arbiter_num_voters Int)
(assert (= a_0_arbiter_num_voters (+ (bool_to_int
↪  a_0_arbiter_0_voter))))
(declare-const a_0_arbiters_meet_quorum Bool)
(assert (= a_0_arbiters_meet_quorum (or (>= a_0_arbiter_num_voters 1)
↪  (> (max a_0_arbiter_num_true_votes a_0_arbiter_num_false_votes)
↪  0))))
(declare-const a_0_arbiter_resolution_true Bool)
```

```
(assert (= a_0_arbiter_resolution_true (and a_0_arbiters_meet_quorum
↪  (> a_0_arbiter_num_true_votes a_0_arbiter_num_false_votes) (>
↪  a_0_arbiter_num_true_votes 0) (> a_0_party_num_voters
↪  a_0_party_num_false_votes))))
(declare-const a_0_arbiter_resolution_false Bool)
(assert (= a_0_arbiter_resolution_false (and a_0_arbiters_meet_quorum
↪  (> a_0_arbiter_num_false_votes a_0_arbiter_num_true_votes) (>
↪  a_0_arbiter_num_false_votes 0) (> a_0_party_num_voters
↪  a_0_party_num_true_votes))))
(assert (= (or a_0_party_resolution_true (and (not
↪  a_0_party_resolution_false) a_0_arbiter_resolution_true)) a_0))
(assert (= (or a_0_party_resolution_false (and (not
↪  a_0_party_resolution_true) a_0_arbiter_resolution_false)) (not
↪  a_0)))
(declare-const a_1 Bool) ; defining action as bool
(assert (not a_1))
(declare-const a_1_party_0_vote Bool)
(declare-const a_1_party_0_voter Bool)
(assert a_1_party_0_voter)
(declare-const a_1_party_1_vote Bool)
(declare-const a_1_party_1_voter Bool)
(assert a_1_party_1_voter)
(declare-const a_1_party_num_true_votes Int)
(assert (= a_1_party_num_true_votes (+ (bool_to_int a_1_party_0_vote)
↪  (bool_to_int a_1_party_1_vote))))
(declare-const a_1_party_num_false_votes Int)
(assert (= a_1_party_num_false_votes (+ (bool_to_int (not
↪  a_1_party_0_vote)) (bool_to_int (not a_1_party_1_vote)))))
(declare-const a_1_party_num_voters Int)
(assert (= a_1_party_num_voters (+ (bool_to_int a_1_party_0_voter)
↪  (bool_to_int a_1_party_1_voter))))
(declare-const a_1_party_resolution_true Bool)
(assert (= a_1_party_resolution_true (and (= a_1_party_num_voters 2)
↪  (> a_1_party_num_true_votes 0) (= a_1_party_num_false_votes 0))))
(declare-const a_1_party_resolution_false Bool)
(assert (= a_1_party_resolution_false (and (= a_1_party_num_voters 2)
↪  (> a_1_party_num_false_votes 0) (= a_1_party_num_true_votes 0))))
(declare-const a_1_arbiter_0_vote Bool)
(declare-const a_1_arbiter_0_voter Bool)
(assert a_1_arbiter_0_voter)
(declare-const a_1_arbiter_num_true_votes Int)
(assert (= a_1_arbiter_num_true_votes (+ (bool_to_int
↪  a_1_arbiter_0_vote))))
(declare-const a_1_arbiter_num_false_votes Int)
```

```
(assert (= a_1_arbiter_num_false_votes (+ (bool_to_int (not
↪  a_1_arbiter_0_vote)))))
(declare-const a_1_arbiter_num_voters Int)
(assert (= a_1_arbiter_num_voters (+ (bool_to_int
↪  a_1_arbiter_0_voter))))
(declare-const a_1_arbiters_meet_quorum Bool)
(assert (= a_1_arbiters_meet_quorum (or (>= a_1_arbiter_num_voters 1)
↪  (> (max a_1_arbiter_num_true_votes a_1_arbiter_num_false_votes)
↪  0))))
(declare-const a_1_arbiter_resolution_true Bool)
(assert (= a_1_arbiter_resolution_true (and a_1_arbiters_meet_quorum
↪  (> a_1_arbiter_num_true_votes a_1_arbiter_num_false_votes) (>
↪  a_1_arbiter_num_true_votes 0) (> a_1_party_num_voters
↪  a_1_party_num_false_votes))))
(declare-const a_1_arbiter_resolution_false Bool)
(assert (= a_1_arbiter_resolution_false (and a_1_arbiters_meet_quorum
↪  (> a_1_arbiter_num_false_votes a_1_arbiter_num_true_votes) (>
↪  a_1_arbiter_num_false_votes 0) (> a_1_party_num_voters
↪  a_1_party_num_true_votes))))
(assert (= (or a_1_party_resolution_true (and (not
↪  a_1_party_resolution_false) a_1_arbiter_resolution_true)) a_1))
(assert (= (or a_1_party_resolution_false (and (not
↪  a_1_party_resolution_true) a_1_arbiter_resolution_false)) (not
↪  a_1)))
(declare-const a_2 Bool) ; defining action as bool
(declare-const a_2_party_0_vote Bool)
(declare-const a_2_party_0_voter Bool)
(assert a_2_party_0_voter)
(declare-const a_2_party_1_vote Bool)
(declare-const a_2_party_1_voter Bool)
(assert a_2_party_1_voter)
(declare-const a_2_party_num_true_votes Int)
(assert (= a_2_party_num_true_votes (+ (bool_to_int a_2_party_0_vote)
↪  (bool_to_int a_2_party_1_vote))))
(declare-const a_2_party_num_false_votes Int)
(assert (= a_2_party_num_false_votes (+ (bool_to_int (not
↪  a_2_party_0_vote)) (bool_to_int (not a_2_party_1_vote)))))
(declare-const a_2_party_num_voters Int)
(assert (= a_2_party_num_voters (+ (bool_to_int a_2_party_0_voter)
↪  (bool_to_int a_2_party_1_voter))))
(declare-const a_2_party_resolution_true Bool)
(assert (= a_2_party_resolution_true (and (= a_2_party_num_voters 2)
↪  (> a_2_party_num_true_votes 0) (= a_2_party_num_false_votes 0))))
(declare-const a_2_party_resolution_false Bool)
```

```
(assert (= a_2_party_resolution_false (and (= a_2_party_num_voters 2)
↪  (> a_2_party_num_false_votes 0) (= a_2_party_num_true_votes 0))))
(declare-const a_2_arbiter_0_vote Bool)
(declare-const a_2_arbiter_0_voter Bool)
(assert a_2_arbiter_0_voter)
(declare-const a_2_arbiter_num_true_votes Int)
(assert (= a_2_arbiter_num_true_votes (+ (bool_to_int
↪  a_2_arbiter_0_vote))))
(declare-const a_2_arbiter_num_false_votes Int)
(assert (= a_2_arbiter_num_false_votes (+ (bool_to_int (not
↪  a_2_arbiter_0_vote)))))
(declare-const a_2_arbiter_num_voters Int)
(assert (= a_2_arbiter_num_voters (+ (bool_to_int
↪  a_2_arbiter_0_voter))))

(check-sat)
```

Bibliography

1. Malin, B., Karp, D. & Scheuermann, R. H. Technical and Policy Approaches to Balancing Patient Privacy and Data Sharing in Clinical and Translational Research. *Journal of Investigative Medicine* **58,** 11–18. ISSN: 1081-5589 (2010).

2. Bharosa, N., Lee, J. & Janssen, M. Challenges and obstacles in sharing and coordinating information during multi-agency disaster response: Propositions from field exercises. *Information Systems Frontiers* **12,** 49–65. ISSN: 1572-9419 (Mar. 2010).

3. Shibata, N. *et al. A Method for Sharing Traffic Jam Information using Inter-Vehicle Communication* in *2006 3rd Annual International Conference on Mobile and Ubiquitous Systems - Workshops* (July 2006), 1–7. doi:10.1109/MOBIQW.2006.361760.

4. Pierro, M. D. What Is the Blockchain? *Computing in Science Engineering* **19,** 92–95. ISSN: 1521-9615 (2017).

5. Ammous, S. Blockchain Technology: What is it good for? *Available at SSRN 2832751* (2016).

6. Crosby, M., Pattanayak, P., Verma, S., Kalyanaraman, V., *et al.* Blockchain technology: Beyond bitcoin. *Applied Innovation* **2,** 71 (2016).

7. O'Dwyer, K. J. & Malone, D. Bitcoin mining and its energy footprint (2014).

8. Nakamoto, S. *et al.* Bitcoin: A peer-to-peer electronic cash system (2008).

9. Carlozo, L. What is blockchain? *Journal of Accountancy* **224,** 29 (2017).

10. Wood, G. *et al.* Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* **151,** 1–32 (2014).

11. *Solidity. The Contract-Oriented Programming Language* version 2a716ad. Ethereum. `https://github.com/ethereum/solidity`.

12. *Vyper. Pythonic Smart Contract Language for the EVM* version d72e998. Ethereum. `https://github.com/ethereum/vyper`.

13. Bjorner, N. & Jayaraman, K. *Checking cloud contracts in Microsoft Azure* in *International Conference on Distributed Computing and Internet Technology* (2015), 21–32.

14. De Moura, L. & Bjorner, N. *Z3: An efficient SMT solver* in *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), 337–340.

15. De Moura, L. M. & Bjorner, N. *Proofs and Refutations, and Z3.* in *LPAR Workshops* **418** (2008), 123–132.

16. *Z3. The Z3 Theorem Prover* version 9f1b8db. Microsoft. `https://github.com/Z3Prover/z3`.

17. Rosu, G. & Serbanuta, T. F. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming* **79,** 397–434 (2010).

18. Hirai, Y. *Defining the ethereum virtual machine for interactive theorem provers* in *International Conference on Financial Cryptography and Data Security* (2017), 520–535.

19. Mehar, M. I. *et al.* Understanding a revolutionary and flawed grand experiment in blockchain: The dao attack. *Journal of Cases on Information Technology (JCIT)* **21,** 19–32 (2019).

20. Knapp, S. Can LegalZoom Be the Answer to the Justice Gap. *Geo. J. Legal Ethics* **26,** 821 (2013).

21. Barton, B. H. Some early thoughts on liability standards for online providers of legal services. *Hofstra L. Rev.* **44,** 541 (2015).

22. McGinnis, J. O. & Pearce, R. G. The great disruption: How machine intelligence will transform the role of lawyers in the delivery of legal services. *Fordham L. Rev.* **82,** 3041 (2013).

23. *Introduction to Ergo* version 0.6.2. Accord Project. `https://docs.accordproject.org/docs/ergo.html`.

24. *Template Specification* version 0.7. Accord Project. `https://docs.accordproject.org/docs/accordproject-specification`.

25. Coblenz, M. *User-Centered Design of Principled Programming Languages* PhD thesis (University of British Columbia, 2018).

26. *Template Studio* version 0.10.0. Accord Project. `https://studio.accordproject.org/`.

27. Flood, M. D. & Goodenough, O. R. Contract as automaton: the computational representation of financial agreements. *Office of Financial Research Working Paper* (2015).

28. Daskalopulu, A. & Sergot, M. The representation of legal contracts. *AI & SOCIETY* **11,** 6–17 (1997).

29. Koepsell, D. & Covarrubias, V. G. The rise of big data and genetic privacy. *Ethics, Medicine and Public Health* **2,** 348–355 (2016).

30. Patel, V. A framework for secure and decentralized sharing of medical imaging data via blockchain consensus. *Health informatics journal,* 1460458218769699 (2018).

31. Azaria, A., Ekblaw, A., Vieira, T. & Lippman, A. *Medrec: Using blockchain for medical data access and permission management* in *2016 2nd International Conference on Open and Big Data (OBD)* (2016), 25–30.

32. Zyskind, G., Nathan, O. & Pentland, A. Enigma: Decentralized computation platform with guaranteed privacy. *arXiv preprint arXiv:1506.03471* (2015).

33. Zyskind, G., Nathan, O., *et al. Decentralizing privacy: Using blockchain to protect personal data* in *2015 IEEE Security and Privacy Workshops* (2015), 180–184.

34. Grabus, S. & Greenberg, J. *Toward a Metadata Framework for Sharing Sensitive and Closed Data: An Analysis of Data Sharing Agreement Attributes* in *Research Conference on Metadata and Semantics Research* (2017), 300–311.

35. Cheng, R. *et al.* Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts (1804).

36. Nugent, T., Upton, D. & Cimpoesu, M. Improving data transparency in clinical trials using blockchain smart contracts. *F1000Research* **5** (2016).

37. Foroglou, G. & Tsilidou, A.-L. *Further applications of the blockchain* in *12th Student Conference on Managerial Science and Technology* (2015).

38. Hildenbrandt, E. *et al. KEVM: A complete formal semantics of the ethereum virtual machine* in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)* (2018), 204–217.

39. Park, D., Zhang, Y., Saxena, M., Daian, P. & Rosu, G. *A formal verification tool for Ethereum VM Bytecode* in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018), 912–915.

40. Bhargavan, K. *et al. Formal verification of smart contracts: Short paper* in *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security* (2016), 91–96.

41. Amani, S., Begel, M., Bortin, M. & Staples, M. *Towards verifying ethereum smart contract bytecode in Isabelle/HOL* in *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (2018), 66–77.

42. Bjorner, N. *Taking satisfiability to the next level with Z3* in *International Joint Conference on Automated Reasoning* (2012), 1–8.

43. Dannen, C. *Introducing Ethereum and Solidity* (Springer, 2017).

44. *Rinkeby: Ethereum Testnet* Rinkeby. `https://rinkeby.io/`.

45. *Rinkeby: Authenticated Faucet* Rinkeby. `https://faucet.rinkeby.io/`.

46. Hu, Y.-C., Lee, T.-T., Chatzopoulos, D. & Hui, P. *Hierarchical interactions between ethereum smart contracts across testnets* in *Proceedings of the 1st Workshop on Cryptocurrencies and Blockchains for Distributed Systems* (2018), 7–12.

47. *Metamask. Brings Ethereum to your browser* version 6.3.0. Metamask. `https://metamask.io/`.

48. *Remix. Solidity IDE* Ethereum. `http://remix.ethereum.org/`.