2018 IEEE Confs on Internet of Things, Green Computing and Communications, Cyber, Physical and Social Computing, Smart Data, Blockchain, Computer and Information Technology, Congress on Cybermatics

SolidWorx: A Resilient and Trustworthy Transactive Platform for Smart and Connected Communities

Scott Eisele
Vanderbilt University
Nashville, TN, USA
scott.r.eisele@vanderbilt.edu

Aron Laszka
University of Houston
Houston, TX, USA
alaszka@houston.edu

Anastasia Mavridou
Vanderbilt University
Nashville, TN, USA
anastasia.mavridou@vanderbilt.edu

Abhishek Dubey
Vanderbilt University
Nashville, TN, USA
abhishek.dubey@vanderbilt.edu

***Abstract*—Internet of Things and data sciences are fueling the development of innovative solutions for various applications in Smart and Connected Communities (SCC). These applications provide participants with the capability to exchange not only data but also resources, which raises the concerns of integrity, trust, and above all the need for fair and optimal solutions to the problem of resource allocation. This exchange of information and resources leads to a problem where the stakeholders of the system may have limited trust in each other. Thus, collaboratively reaching consensus on when, how, and who should access certain resources becomes problematic. This paper presents *SolidWorx*, a blockchain-based platform that provides key mechanisms required for arbitrating resource consumption across different SCC applications in a domain-agnostic manner. For example, it introduces and implements a hybrid-solver pattern, where complex optimization computation is handled off-blockchain while solution validation is performed by a smart contract. To ensure correctness, the smart contract of *SolidWorx* is generated and verified using a model-based approach.**

1. Introduction

Smart and connected communities (SCC) as a research area lies at the intersection of social science, machine learning, cyber-physical systems, civil infrastructures, and data sciences. This research area is enabled by the rapid and transformational changes driven by innovations in smart sensors, such as cameras and air quality monitors, which are now embedded in almost every physical device and system that we use, ranging from watches and smartphones to automobiles, homes, roads, and workplaces. The effects of these innovations can be seen in a number of diverse domains, including transportation, energy, emergency response, and health care, to name a few.

At its core, a smart and connected community is a multi-agent system where agents may enter or leave the system for different reasons. Agents may act on behalf of service owners, managing access to services and ensuring that contracts are fulfilled. Agents can also act on behalf of service consumers, locating services, entering contracts, as well as receiving and presenting results. For example, agents may coordinate carpooling services. Another example of such coordination exists in transactive energy systems [1], where homeowners in a community exchange excess energy. Consequently, these agents are required to engage in interactions, negotiate with each other, enter agreements, and make proactive run-time decisions—individually and collectively—while responding to changing circumstances.

This exchange of information and resources leads to a problem where the stakeholders of the system may have limited trust in each other. Thus, collaboratively reaching consensus on when, how, and who should access certain resources becomes problematic. However, instead of solving these problems in a domain specific manner, we present *SolidWorx* and show how this platform can provide key design patterns to implement mechanisms for arbitrating resource consumption across different SCC applications.

Blockchains may form a key component of SCC platforms because they enable participants to reach a consensus on the value of any state variable in the system, without relying on a trusted third party or trusting each other. Distributed consensus not only solves the trust issue, but also provides fault-tolerance since consensus is always reached on the correct state as long as the number of faulty nodes is below a threshold. Further, blockchains can also enable performing computation in a distributed and trustworthy manner in the form of smart contracts. However, while the distributed integrity of a blockchain ledger presents unique opportunities, it also introduces new assurance challenges that must be addressed before protocols and implementations can live up to their potential. For instance, Ethereum smart contracts deployed in practice are riddled with bugs and security vulnerabilities. Thus, we use a correct-by-construction design toolchain, called FSolidM [2], to design and implement the smart-contract code of *SolidWorx*.

The outline of this paper is as follows. We formulate a resource-allocation problem for SCC in Section 2, describing two concrete applications of the platform in Section 2.2 and presenting extensions to the basic problem formulation in Section 2.3. We describe our solution architecture in Section 3, which consists of off-blockchain solvers (Section 3.2) and a smart contract (Section 3.3), providing a brief analysis in Section 3.4. In Section 4, we evaluate *SolidWorx* using two case studies, a carpooling assignment (Section 4.1) and an energy trading system (Section 4.2).

978-1-5386-7975-3/18/$31.00 ©2018 IEEE
DOI 10.1109/Cybermatics_2018.2018.00221

TABLE 1: List of Symbols

Symbol	Description
P	set of resource providers
C	set of resource consumers
T	set of resource types
OP	set of providing offers
OC	set of consumption offers
o_P	resource provider who posted offer $\boldsymbol{o} \in OP$
o_C	resource consumer who posted offer $\boldsymbol{o} \in OC$
$o_Q(t)$	amount of resources of type $t \in T$ provided or requested by offer $\boldsymbol{o}$
$o_V(t)$	unit reservation price of offer $\boldsymbol{o}$ for resource type $t \in T$
a_{OP}	providing offer from which assignment $\boldsymbol{a}$ allocates resources
a_{OC}	consuming offer to which assignment $\boldsymbol{a}$ allocates resources
a_Q	amount of resources allocated by assignment $\boldsymbol{a}$
a_T	type of resources allocated by assignment $\boldsymbol{a}$
a_V	unit price for the resources allocated by assignment $\boldsymbol{a}$

Finally, we discuss the architecture of *SolidWorx* in the context of related research in Section 5, and we provide concluding remarks in Section 6.

2. Problem Formulation

We first introduce a base formulation of an abstract resource allocation problem (Section 2.1), which captures the core functionality of a transactive platform for SCC. Then, we describe two examples of applying this formulation to solving practical problems in SCC (Section 2.2). We conclude the section by introducing various extensions to the base problem formulation, in the form of alternative objectives and additional constraints (Section 2.3). A list of the key symbols used in the resource allocation problem can be found in Table 1.

2.1. Resource Allocation Problem

In essence, the objective of the transactive platform is to allocate resources from users who provide resources to users who consume them. The sets of *resource providers* and *resource consumers* are denoted by P and C, respectively. Note that a user may act both as a resource provider and as a resource consumer, in which case the user is a member of both P and C. Resources that are provided or consumed belong to a set of *resource types*, which are denoted by T. A resource type is an abstract concept, which captures not only the inherent characteristics of a resource, but all aspects related to providing or consuming resources. For example, a resource type could correspond to energy production or consumption in a specific time interval, or it could correspond to a ride between certain locations at a certain time.

Each provider $p \in P$ may post a set of *providing offers*. Each providing offer $\boldsymbol{o}$ is a tuple $\boldsymbol{o} = \langle o_P, o_Q, o_V \rangle$, where $o_P \in P$ is the provider who posted the offer, $o_Q \in T \mapsto \mathbb{N}$ is the amount of resources offered from each type (i.e., $o_Q(t)$ is the amount of resources offered from type $t \in T$), and $o_V \in T \mapsto \mathbb{N}$ is the unit reservation price asked for each resource type (i.e., $o_V(t)$ is the value asked for providing a unit resource of type $t \in T$). Each offer $\boldsymbol{o} = \langle o_P, o_Q, o_V \rangle$ defines a set of alternatives: provider o_p offers to provide either $o_Q(t_1)$ resources of type $t_1 \in T$ or $o_Q(t_2)$ resources of type $t_2 \in T$, but not at the same time. However, convex linear combinations, such as providing $\lfloor \alpha \cdot o_Q(t_1) \rfloor$ resources of type $t_1 \in T$ and $\lfloor (1-\alpha) \cdot o_Q(t_2) \rfloor$ resources of type $t_2 \in T$ at the same time (where $\alpha \in [0,1]$), are allowed. For example, an offer $\boldsymbol{o}$ providing $o_Q(t_1)$ units of energy in time interval t_1 or $o_Q(t_2)$ units of energy in time interval t_2 may provide $\lfloor 0.5 \cdot o_Q(t_1) \rfloor$ energy in time interval t_1 and $\lfloor 0.5 \cdot o_Q(t_2) \rfloor$ energy in time interval t_2. The set of all offers posted by all the providers is denoted by OP.

Each consumer $c \in C$ posts a set of *consumption offers*. Each consumption offer $\boldsymbol{o}$ is a tuple $\boldsymbol{o} = \langle o_C, o_Q, o_V \rangle$, where $o_C \in C$ is the consumer who posted the offer, $o_Q \in T \mapsto \mathbb{N}$ is the amount of resources requested from each type (i.e., $o_Q(t)$ is the amount of resources requested from type $t \in T$), and $o_V \in T \mapsto \mathbb{N}$ is the unit reservation price offered for each resource type (i.e., $o_V(t)$ is the value offered for a unit resource of type $t \in T$). Similar to providing offers, consumption offers also define a set of alternatives. The set of all offers posted by all the consumers is denoted by OC.

A *resource allocation* A is a set of resource assignments. Each resource assignment $\boldsymbol{a} \in A$ is a tuple $\boldsymbol{a} = \langle a_{OP}, a_{OC}, a_Q, a_T, a_V \rangle$, where $a_{OP} \in OP$ is a providing offer posted by a provider, $a_{OC} \in OC$ is a consumption offer posted by a consumer, $a_Q \in \mathbb{N}$ and $a_T \in T$ are the amount and type of resources allocated from offer a_{OP} to a_{OC}, and $a_V \in \mathbb{N}$ is the unit price for the assignment.

A resource allocation A is *feasible* if

$$\forall \boldsymbol{o} \in OP: \sum_{t \in T} \sum_{\substack{\boldsymbol{a} \in A: \\ a_{OP} = \boldsymbol{o} \wedge a_T = t}} \frac{a_Q}{o_Q(t)} \leq 1 \tag{1}$$

$$\forall \boldsymbol{o} \in OC: \sum_{t \in T} \sum_{\substack{\boldsymbol{a} \in A: \\ a_{OC} = \boldsymbol{o} \wedge a_T = t}} \frac{a_Q}{o_Q(t)} \leq 1 \tag{2}$$

$$\forall \boldsymbol{a} \in A: \ (a_{OP})_V(a_T) \leq a_V \tag{3}$$

$$\forall \boldsymbol{a} \in A: \ (a_{OC})_V(a_T) \geq a_V. \tag{4}$$

In other words, a resource allocation is feasible if the resources assigned from each providing offer (or consuming offer) is a convex linear combination of the offered (or requested) resources, and if the value in each assignment is higher than (or lower than) the reservation price of the providing offer (or consuming offer).

The objective of the base formulation of the *resource allocation problem* is to maximize the amount of resources assigned from providers to consumers. We define the base formulation of the problem as follows.

Definition 1 (Resource Allocation Problem)**.** Given sets of providing and consumption offers OP and OC, find a feasible resource allocation A that attains the maximum

$$\max_{A:\ A \text{ is feasible}} \sum_{\boldsymbol{a} \in A} a_Q. \tag{5}$$

2.2. Example Applications

To illustrate how the Resource Allocation Problem (RAP) may be applied in smart and connected communi-

ties, we now describe two example problems that can be expressed using RAP.

2.2.1. Energy Futures Market

We consider a residential energy-futures market in a transactive microgrid. In this application, resource consumers model residential energy consumers (i.e., households), while resource providers model the subset of consumers who have energy providing capabilities (e.g., solar panels, batteries). We divide time into fixed-length intervals (e.g., 15 minutes), and let each resource type correspond to providing or consuming a unit amount of power (e.g., 1 W) in a particular time interval.

Based on their predicted energy supply and demand, residential consumers (or smart homes acting on their behalf) post offers to provide or consume energy in future time intervals. For instance, a provider may predict that it will be able to generate a certain amount of power π using its solar panel during time intervals $t_1, t_2, \ldots, t_N \in T$. Then, it will submit a *set of N offers*: for each time interval $t \in \{t_1, \ldots, t_N\}$ in which energy may be produced, it posts an offer specifying

$$o_Q(t) = \begin{cases} \pi & \text{if } t = t \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

Alternatively, the provider may have a fully charged battery, which could be discharged in any of the next N intervals $t_1, t_2 \ldots, t_N$. Let π denote the amount of power that could be provided if the battery was fully discharged in a single time interval. Then, the provider will submit a *single offer* specifying

$$o_Q(t) = \begin{cases} \pi & \text{if } t \in \{t_1, t_2, \ldots, t_N\} \\ 0 & \text{otherwise.} \end{cases} \tag{7}$$

The reservation prices of the offers should consider the energy prices of the utility company (i.e., the alternative to local trading) and the cost of providing energy (e.g., cost of battery depreciation due to charging and discharging).

2.2.2. Carpooling Assignment

We consider the problem of assigning carpooling riders to drivers with empty seats in their cars. In this application, resource consumers model riders, while resource providers model drivers. We again divide time into fixed-length intervals, and we divide the space of pick-up locations into a set of areas (e.g., city blocks). Then, we let a resource type correspond to a ride from a particular area in a particular time interval to a particular area. A unit of a resource is a single seat for a ride.

A provider (i.e., driver) who has π empty seats in its car will post a providing offer. Let $\Pi \subseteq T$ denote the set of combinations of pick-up and drop-off areas and pick-up times that are feasible for the provider. Then, the provider's offer specifies

$$o_Q(t) = \begin{cases} \pi & \text{if } t \in \Pi \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

Similarly, a consumer (i.e., rider) who needs 1 seat will post a consuming offer, specifying

$$o_Q(t) = \begin{cases} 1 & \text{if } t \in \Pi \\ 0 & \text{otherwise,} \end{cases} \tag{9}$$

where Π is the set of combinations (i.e., pick-up and drop-off areas and pick-up times) that are feasible for the rider.

2.3. Problem Formulation Extensions

The Resource Allocation Problem that we introduced in Section 2.1 can capture a wide range of real-world problems. However, some problems may not be easily expressed using the constraints (Equations (1) to (4)) and the objective (Equation (5)) of the base problem formulation. For this reason, here we introduce a set of alternative objective formulations and additional constraints for resource allocation.

2.3.1. Objectives

We first introduce alternative objective formulations, which quantify the utility of a resource allocation based on alternative goals.

Resource Type Preferences: Equation (5) assumes that exchanging a unit of any resource type is equally beneficial. In some practical scenarios, exchanging certain resource types may be more beneficial than exchanging others. For each resource type $t \in T$, let β_t denote the utility derived from exchanging a unit of resources of type t. Then, the utility of a resource allocation A can be expressed as

$$\sum_{\boldsymbol{a} \in A} \beta_{(a_T)} \cdot a_Q. \tag{10}$$

Provider and Consumer Benefit: The reservation price $o_V(t)$ of a providing offer $\boldsymbol{o}$ means that provider o_P is indifferent to (i.e., derives zero benefit from) exchanging resources of type t at unit price $o_V(t)$. Hence, the unit benefit derived by the provider from exchanging at a higher price $a_V \geq o_V(t)$ is equal to $a_V - o_V(t)$. Similarly, the unit benefit derived by a consumer, who posted an offer $\boldsymbol{o}$, from exchanging resources of type t at price a_V is equal to $o_V(t) - a_V$. Therefore, the total benefit created by an assignment $\boldsymbol{a}$ for provider a_{OP} and consumer a_{OC} is

$$\begin{aligned} &a_Q \cdot [a_V - (a_{OP})_V (a_T)] + a_Q \cdot [(a_{OC})_V (a_T) - a_V] \\ =&a_Q \cdot [(a_{OC})_V (a_T) - (a_{OP})_V (a_T)], \end{aligned} \tag{11}$$

and the total benefit created by a resource allocation A for all the providers and consumers is

$$\sum_{\boldsymbol{a} \in A} a_Q \cdot [(a_{OC})_V (a_T) - (a_{OP})_V (a_T)]. \tag{12}$$

2.3.2. Constraints

Next, we introduce additional feasibility constraints that may be imposed on the resource allocations.

Price Constraints: A regulator (e.g., utility company in a transactive energy platform) may impose constraints on the prices at which resources may be exchanged (e.g., based on bulk-market prices). If the minimum and maximum

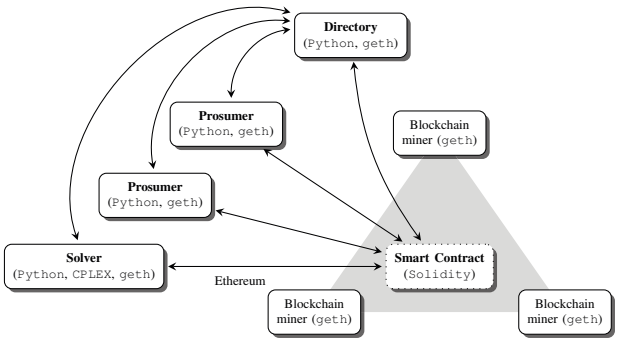

Figure 1: Implementation view of the *SolidWorx*. A private Ethereum network (used for testing purposes) is the decentralized computation platform running the smart contracts.

unit prices for resource type $t \in T$ are min_t and max_t, respectively, then we can express price constraints as

$$\forall \boldsymbol{a} \in A: \; min_{(a_T)} \leq a_V \leq max_{(a_T)}. \tag{13}$$

Pairwise Constraints: Due to physical constraints, exchanging resources of certain types between certain pairs of prosumers may be impossible. If the set of prosumer pairs that may exchange resources of type $t \in T$ is denoted by $E_t \subseteq P \times C$, we can express pairwise constraints as

$$\forall \boldsymbol{a} \in A : (a_{OP}, a_{OC}) \in E_{(a_T)}. \tag{14}$$

Real-Valued Offers and Allocations: Finally, we may also relax some of the constraints of the base formulation. In particular, we may allow real-valued quantities in offers and allocations (i.e., $o_Q : T \mapsto \mathbb{R}_+$ and $a_Q \in \mathbb{R}_+$) as well as real-valued prices (i.e., $o_V : T \mapsto \mathbb{R}_+$ and $a_V \in \mathbb{R}_+$).

3. *SolidWorx*: A Decentralized Transaction Management Platform

Now, we describe the *SolidWorx* platform, which (1) allows prosumers[1] to post offers and (2) can find a solution to the resource allocation problem in an efficient and trustworthy manner. SolidWorx follows the actor-based architecture, which was proposed initially in [3], and which has been accepted as a standard model for building distributed applications. The key aspect of an actor-based system are interfaces with well defined execution models [4]. The following subsections will describe the transaction management platform in more detail. Here, we provide a brief overview.

Figure 1 shows the key actors of our transaction management platform, while Figure 2 describes the data flow between these actors. A directory actor provides a mechanism to look up connection endpoints, including the address of a deployed smart contract. In our previous work, we described how to create a decentralized directory service using distributed hash tables [5]. Therefore, we do not describe the implementation of this service any further in this paper. Prosumer actors (i.e., resource providers and consumers) post offers to the platform via functions provided by a

1. An actor or an agent that can both provide and consume resources.

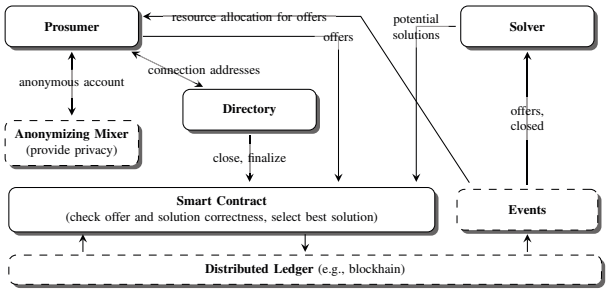

Figure 2: Data flow between actors of *SolidWorx*.

smart contract. These functions check the correctness of each offer and then store it within the smart contract. An optional mixer service can be used to obfuscate the identity of the prosumers [6]. By generating new anonymous addresses at random periodically, prosumers can prevent other entities from linking the anonymous addresses to their actual identities [7], [6], thereby keeping their activities private. Solver actors, which are pre-configured with constraints and an objective function, can listen to smart-contract events, which provide the solvers with information about offers. Solvers run at a pre-configured intervals, compute a resource allocation, and submit the solution allocation to the smart contract. The directory, acting as a service director, can then finalize a solution by invoking a smart-contract function, which chooses the best solution from all the allocations that have been submitted. Once a solution has been finalized, the prosumers are notified using smart-contract events.

3.1. Scheduling

Once the system is deployed, providers and consumers will need to use it repeatedly for finding optimal resource allocations. For example, riders and drivers want to find optimal carpooling assignments every day, while users in an energy futures market want to find optimal energy trades every, e.g., 20 minutes. Consequently, the platform has to gather offers and solve the resource allocation problem at regular time intervals. Each one of these cycles is divided into two phases. First, an *offering phase*, in which providers and consumers can post new offers or cancel their existing offers (e.g., if they wish to change their offer based on changes in the market). Second, a *solving phase*, in which the resource allocation problem is solved for the posted (but not cancelled) offers. At the end of the second phase, the assignments between providers and consumers are finalized based on the solution. Then, a new cycle begins with an offering phase.

3.2. Hybrid Solver Architecture

The Resource Allocation Problem described in Section 2 can be solved by formulating it as an (integer) linear program (LP): feasibility constraints (Equations (1) to (4)) and constraint extensions (Section 2.3.2) can all be formulated as linear inequalities, while the objective function (Equation (5)) as well as the alternative objectives (Section 2.3.1) can be formulated as linear functions. Arguably, we could

set up a solver actor that would receive offers from prosumers, formulate a linear program, and use a state-of-the-art LP-solver (e.g., CPLEX [8]) to find an optimal solution. However, a simple N-to-1 architecture with N prosumers and 1 solver would suffer from the following problems:

- *Lack of trust in solver nodes:* Prosumers would need to trust that the solver is acting selflessly and is providing correct and optimal solutions.
- *Vulnerability of the transaction management platform:* A single solver would be a single point of failure. If it were faulty or compromised, the entire platform would be faulty or compromised.
- *Data storage:* For the sake of auditability, information about past offers and allocations should remain available even in case of node failures.

A decentralized ledger with distributed information storage and consensus provided by blockchain solutions, such as Ethereum, is an obvious choice for ensuring the auditability of all events and providing distributed trust. However, computation is relatively expensive on blockchain-based distributed platforms[2], solving the trading problem using a blockchain-based smart contract would not be scalable in practice. In light of this, we propose a *hybrid implementation approach*, which combines the trustworthiness of blockchain-based smart contracts with the efficiency of more traditional computational platforms.

Thus, the key idea of our hybrid approach is to (1) use a high-performance computer to solve the computationally expensive linear program *off-blockchain* and then (2) use a smart contract to record the solution *on the blockchain*. To implement this hybrid approach securely and reliably, we must address the following issues.

- Computation that is performed off-blockchain does not satisfy the auditability and security requirements that smart contracts do. Thus, the results of any off-blockchain computation must be verified by the smart contract before recording them on the blockchain.
- Due to network disruptions and other errors (including deliberate denial-of-service attacks), the off-blockchain solver might fail to provide the smart contract with a solution on time (i.e., before assignments are supposed to be finalized). Thus, the smart contract must not be strongly coupled to the solver.
- For the sake of reliability, the smart contract should accept solutions from multiple off-blockchain sources; however, these sources might provide different solutions. Thus, the smart contract must be able to choose from multiple solutions (some of which may come from a compromised computer).

3.3. Smart Contract

We implement a smart contract that can (1) verify whether a solution is feasible and (2) compute the value of the objective function for a feasible solution. Compared to finding an optimal solution, these operations are computationally inexpensive, and they can easily be performed on a blockchain-based decentralized platform. Thus, we implement a smart contract that provides the following functionality:

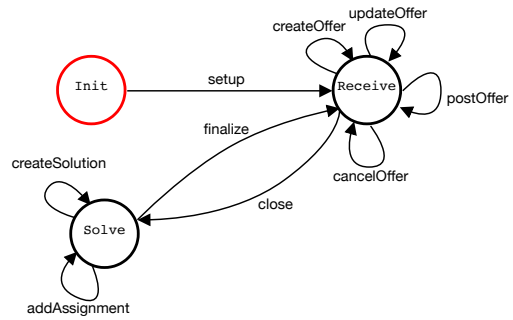

Figure 3: FSolidM model of the *SolidWorx* smart contract.

- Solutions may be submitted to the contract at any time during the solving phase. The contract verifies the feasibility of each submitted solution, and if the solution is feasible (i.e., if it satisfies the constraint Equations (1) to (4)), then it computes the value of the objective function (i.e., Equation (5)). The contract always keeps track of the best feasible solution submitted so far, which we call the *candidate solution*.
- At the end of the solving phase, the contract finalizes resource assignments for the cycle based on the candidate solution. If no solution has been submitted to the contract, then an empty allocation is used as a solution, which is always feasible but attains zero objective.

This simple functionality achieves a high level of security and reliability. Firstly, it is clear that an adversary cannot force the contract to finalize assignments based on an incorrect (i.e., infeasible) solution since such a solution would be rejected. Similarly, an adversary cannot force the contract to choose an inferior solution instead of a superior one. In sum, the only action available to the adversary is proposing a superior feasible solution, which would actually improve the transactive management platform.

To ensure that the smart-contract code is correct-by-construction [9], we use the formal design environment FSolidM [2] to design and generate the Solidity code of the smart contract. FSolidM allows designing Ethereum smart contracts as Labelled Transition Systems (LTS) with formal semantics. Each LTS can then be given to the NuSMV model checker [10] to verify liveness, deadlock-freedom, and safety properties, which can capture important security concerns.

In Figure 3, we present the LTS representation of the transactive-platform smart contract, designed with FSolidM. The contract has three states:[3]

2. Further, Solidity, the preferred high-level language for Ethereum, currently lacks built-in support for certain features that would facilitate the implementation of an LP solver, such as floating-point arithmetics.

3. Generated smart-contract code is not included in the paper because of space constraints. However, interested readers can view the code at `https://github.com/visor-vu/transaction-management-platform`

- `Init`, in which the contract has been deployed but not been initialized. Before the contract can be used, it must be initialized (i.e., numerical parameters must be set up).
- `Receive`, which corresponds to the *offering* phase of a cycle (see Section 3.1). In this state, prosumers may post (or cancel) their offers.
- `Solve`, which corresponds to the *solving* phase of a cycle (see Section 3.1). In this state, solvers may submit solutions (i.e., resource allocations) based on the posted (but not cancelled) offers.

In FSolidM, smart-contract functions are modeled as LTS transitions. Note that by design, each function may be executed only if the contract is in the origin state of the corresponding transition. Our smart contract has the following transitions (after the name of each transition, we list the function parameters):

- from state `Init`:
  - `setup(uint64 numTypes, uint64 precision, uint64 maxQuantity, uint64 lengthReceive, uint64 lengthSolve)`: initializes a contract with numerical parameter values, setting up the number of resource types, the arithmetic precision for calculations, the maximum quantity that may be offered, and the time length of the offering and solving phases; upon execution, the contract transitions to state `Receive`.
- from state `Receive`:
  - `createOffer(bool providing, uint64 misc))`: creates a blank offer (belonging to the prosumer invoking this transition) within the smart contract; parameter `providing` is true for providers and false for consumers, parameter `misc` is an arbitrary value that prosumers may use for their own purposes (e.g., to distinguish between their own offers); emits an `OfferCreated` event.
  - `updateOffer(uint64 ID, uint64 resourceType, uint64 quantity, uint64 value)`: sets quantity and value for a resource type in an existing offer (identified by the `ID` given in the `OfferCreated` event); may be invoked only by the entity that created the offer, and only if the offer exists but has not been posted yet; emits an `OfferUpdated` event.
  - `postOffer(uint64 ID)`: posts an existing offer, enabling solvers to include this offer in a solution; may be invoked only by the entity that created the offer; emits an `OfferPosted` event.
  - `cancelOffer(uint64 ID)`: cancels (i.e., "un-posts") an offer, forbidding solvers from including this offer in a solution; may be invoked only by the entity that created the offer; emits an `OfferCanceled` event.
  - `close()`: protected by a guard condition on time, which prevents the execution of this transition before the offering phase of the current cycle ends; transitions to state `Solve`; emits a `Closed` event.
- from state `Solve`:
  - `createSolution(uint64 misc)`: creates a new, empty solution (i.e., resource allocation) within the smart contract; parameter `misc` is an arbitrary value that solvers may use for their own purposes (e.g., to distinguish between their own solutions); emits a `SolutionCreated` event.
  - `addAssignment(uint64 ID, uint64 providingOfferID, uint64 consumingOfferID, uint64 resourceType, uint64 quantity, uint64 value)`: adds a resource assignment to an existing solution (identified by the `ID` given in the `SolutionCreated` event); may be invoked only by the entity that created the solution; checks a number of constraints ensuring that the solution remains valid if this assignment is added; emits an `AssignmentAdded` event.
  - `finalize()`: selects the best solution and finalizes it by emitting an `AssignmentFinalized` event for each assignment in the solution; protected by a guard condition on time, which prevents the execution of this transition before the solving phase of the current cycle ends; transitions to state `Receive`.

Notice that posting an offer or submitting a solution requires at least three or two function calls, respectively. The reason for dividing these operations into multiple function calls is to ensure that the computational cost of each function call is constant:

- `createOffer`, `postOffer`, `cancelOffer`, and `createSolution` are obviously constant-cost.
- `updateOffer` adds a single resource type to an offer.
- `addAssignment` simply updates the sum amounts on the left-hand sides of Equations (1) and (2) for a single providing and a single consuming offer, respectively; and then it updates the sum in Equation (5).

With variable-cost functions, posting a complex offer or submitting a complex solution could be infeasible due to large computational costs, which could exceed the gas limit.[4]

A typical sequence of function calls and events in *SolidWorx* is shown in Figure 4.

3.4. Analysis

The computational cost of every smart-contract function is constant (i.e., $O(1)$) except for `finalize`, whose cost is an affine function of the size of the solution (i.e., $O(|A|)$). Note that the cost of `finalize` depends on the size of the solution A only because it emits an event for every assignment $\boldsymbol{a} \in A$. These could be omitted for the sake of maximizing performance since the assignments have already been recorded in the blockchain anyway. The number of function calls required for posting an offer depends on the number of resource types with non-zero quantity in the offer. If there are n such resource types, then $n + 2$ calls are required (`createOffer`, n `updateOffer`, and `postOffer`). The number of function calls required for

4. In Ethereum, each transaction is allowed to consume only a limited amount of gas, which corresponds to the computational and storage cost of executing the transaction.

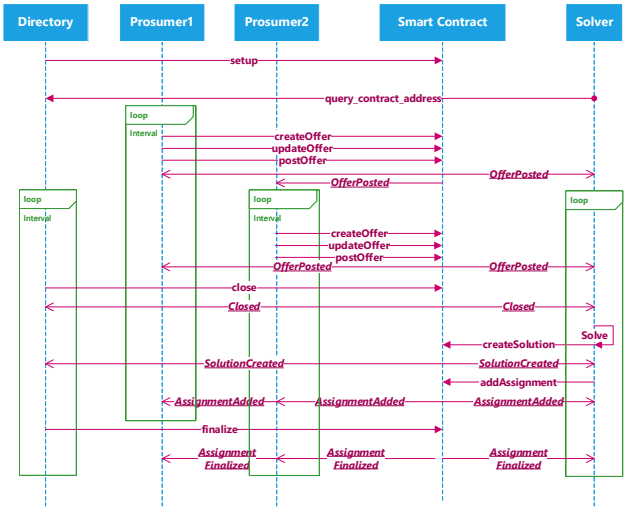

Figure 4: A possible sequence of operations in *SolidWorx*. Underlined text denotes events emitted by the smart contract. Some events, such as `OfferUpdated`, are omitted for simplicity.

submitting a solution A is $1 + |A|$ (`createSolution` and $|A|$ `addAssignment`).

3.4.1. Verification

For the specification of safety and liveness properties, we use Computation Tree Logic (CTL) [11]. CTL formulas specify properties of execution trees generated by transitions systems. The formulas are built from atomic predicates that represent transitions and statements of the transition system (i.e., smart contract), using several operators, such as `AX`, `AF`, `AG` (unary) and, $\mathtt{A}[\cdot\,\mathtt{U}\,\cdot]$, $\mathtt{A}[\cdot\,\mathtt{W}\,\cdot]$ (binary). Each operator consists of a quantifier on the branches of the tree and a temporal modality, which together define when in the execution the operand sub-formulas must hold. The intuition behind the letters is the following: the branch quantifier is `A` (for "All") and the temporal modalities are `X` (for "neXt"), `F` (for "some time in the Future"), `G` (for "Globally"), `U` (for "Until") and `W` (for "Weak until"). A property is satisfied if it holds in the initial state of the transition systems. For instance, the formula $\mathtt{A}[p\,\mathtt{W}\,q]$ specifies that in *all execution branches* the predicate p must hold *up to the first state* (not including this latter) where the predicate q holds. Since we used the weak until operator `W`, if q never holds, p must hold forever. As soon as q holds in one state of an execution branch, p does not need to hold anymore, even if q does not hold. On the contrary, the formula $\mathtt{AG}\,\mathtt{A}[p\,\mathtt{W}\,q]$ specifies that the subformula $\mathtt{A}[p\,\mathtt{W}\,q]$ must hold in *all branches at all times*. Thus, p must hold whenever q does not hold, i.e., $\mathtt{AG}\,\mathtt{A}[p\,\mathtt{W}\,q] = \mathtt{AG}\,(p \vee q)$.

We verified correctness of behavioral semantics with the NuSMV model checker [10], by verifying the following properties:

- *deadlock-freedom*, which ensures that the contract cannot enter a state in which progress is impossible;
- *"if `close` happens, then `postOffer` or `cancelOffer` can happen only after `finalize`"*, translated to CTL as: `AG(close)` $\rightarrow$ `AX A` `[¬(postOffer ∧ cancelOffer) W finalize]`, which ensures that prosumers cannot post or cancel their offers once the solvers have started working;
- *"`OfferPosted(ID)` can happen only if `(ID < offers.length) && !offers[ID].posted && (offers[ID].owner == msg.sender)`"*, translated to CTL as: `A[¬OfferPosted(ID) W (ID < offers.length) && !offers[ID].posted && (offers[ID].owner == msg.sender)]`, which ensures that an offer can be posted only if it has been created (but not yet posted) and only by its creator;
- *"`OfferCanceled(ID)` can happen only if `(ID < offers.length) && offers[ID].posted && (offers[ID].owner == msg.sender)`"*, translated to CTL as: `A[¬OfferCanceled(ID) W (ID < offers.length) && offers[ID].posted && (offers[ID].owner == msg.sender)]`, which ensures that an offer can be canceled only if it has been posted and only by the poster;
- *"if `finalize` happens, then `createSolution` can happen only after `close`"*, translated to CTL as: `AG(finalize)` $\rightarrow$ `AX A [¬createSolution W close]`, which ensures that solutions can be posted only in the solving phase.

4. Case Studies

To evaluate our platform, we present two case studies, based on the energy trading and carpooling problems (Section 2.2), with numerical results. The computational results for the carpool example were obtained on a virtual machine configured with 16 GB of RAM and 4 cores of a i7-6700HQ processor. The energy market example results were obtained on a virtual machine configured with 8GB of RAM and 2 cores of an i7-6700HQ processor. For these experiments, we used our private Ethereum blockchain network [12].

4.1. Carpooling Problem

In this section, we describe a simulated carpooling scenario. The problem of carpooling assignment was introduced earlier in Section 2.2.2. Here, we model a carpool prosumer as an actor that specifies 1) whether it is providing or requesting a ride, 2) the number of seats being offered/requested, 3) a residence, 4) a destination, 5) a time interval during which the ride is available/required, 6) and a radius specifying how far out of their way they are willing to travel. To setup the carpooling problem, we need to identify these parameters and encode them as offers.

Residences were generated by sampling from real-trip distribution data of Vanderbilt University. Destinations were chosen uniformly at random for each prosumer from the 5 garages around Vanderbilt University. Other parameters were also chosen randomly: number of seats from the range of 1 to 3, prosumer type from producer or consumer, time interval from 15-minute intervals between 7:00 and

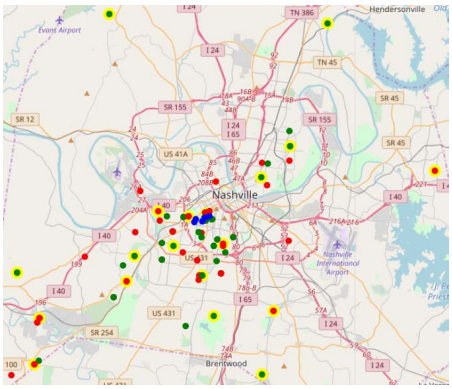

Figure 5: Green and red dots mark the 75 residences (anonymized and resampled). Blue dots are destinations on campus. We used K-Means to identify 20 central locations (yellow dots) for pickup.

9:30AM. The "out of the way" metric was chosen to be half of the distance between the residence and the destination. For a provider, the center of the pick-up circle is the midway point between the residence and the destination, and for a consumer, the center is the residence.

Since each prosumer has a distinct residence, encoding it as a unique resource type would mean that every prosumer would need to have the address of every other prosumer to determine if they are in their pick-up range. Instead, we specify *pick-up points* which are public locations were carpoolers can meet. Each prosumer can determine which pick-up points are within their out-of-the-way radius and list those points in their offer. To encode these values, we assign an ID to each pickup point and destination. Finally, we encode each 15-minute interval using a timestamp.

An offer consists of a collection of alternative resource types, each with a quantity and value. We encode a resource type, which is a combination of a time interval, a pick-up point, and a destination, as a 64-bit unsigned integer. For example, if timestamp is 1523621700, pick-up location ID is 15, and destination ID is 3, then the resource type is 1523621700153. A complete offer may look as follows:

$$\{\texttt{True},\ \{1523623500173:2, 1523623500153:2,\\ 1523624400153:2, 1523624400173:2\},\\ \{1523623500173:10, 1523623500153:10,\\ 1523624400153:10, 1523624400173:10\}\ \}.$$

In this offer, the prosumer is offering rides (`True` for providing), has two pick-up locations in range (17 and 15), drives to destination 3, is available in two time intervals, offers 2 seats, and asks for value 10 in exchange for a ride.

In our experiment, we selected 75 prosumers for the carpool service simulation. The red and green points in Figure 5 are the locations of the consumers and producers randomly sampled from the anonymized distribution data of employees of Vanderbilt University. The yellow points were selected as pick up locations using K-Means clustering choosing 20 clusters. The blue points are 5 garages around Vanderbilt campus where employees typically park.

Figure 6: Each bar is a 15 minute interval. Each color in a bar is an offer that is valid during that interval. The height of each color is the number of seats offered. If the color appears during another interval that means it could be matched in any one of them, but no more than one.

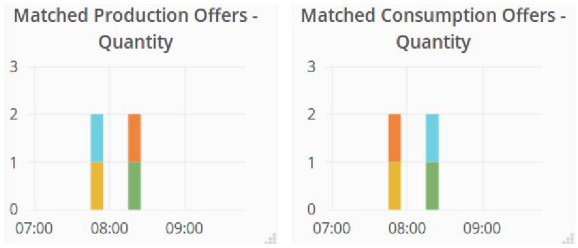

Figure 7: 4 production and 4 consumption offers that were matched. The blue and yellow production offers are matched with the orange and yellow consumption offers. The height of each color is the number of seats in that offer that were matched.

Figure 6 shows all the offers posted to the platform. Figure 7 shows the production and consumption offers that were matched. The running time of the solver was 23 ms, while the time between the request for finalization and emission of `AssignmentFinalized` events was 29 s.

4.2. Energy Trading Problem

To show the versatility of our transaction management platform, we now apply it to the problem of energy trading within a microgrid, which we introduced in Section 2.2.1. In this example, a prosumer is modeled as an actor with an energy generation and consumption profile for the near future. In practice, the generation profile would be typically derived from predictions based on the weather, energy generation capabilities, and the amount of battery storage available. The

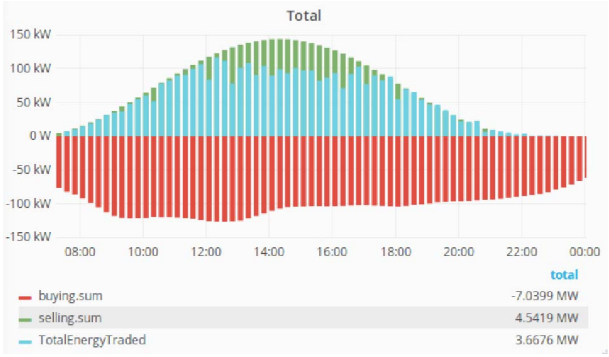

Figure 8: Total energy production capacity (green) and energy demand (red) for each interval, as well as the total energy traded in each interval (blue).

Figure 9: A failure scenario with failure at 8:15AM. The solver can submit new solutions as time progresses; the most recent solution is the color that is on the top of the stack for an interval.

consumption profile would be derived from flexible power loads, like washers and electric vehicles.

To represent future generation or consumption at a certain time, resource types encode timestamps for 15-minute intervals, during which the power will be generated or consumed. As an example, consider a battery that has 500 Wh energy, which could be discharged any time between 9AM and 10AM. This can be represented by an offer having resource types 900, 915, 930, and 945, specifying a quantity of 500 Wh for each.

For our simulation, the prosumer energy profiles are load traces recorded by Siemens during a day from a microgrid in Germany, containing 102 homes (5 producers and 97 consumers). Since the dataset does not include prices, we assume reservation prices to be uniform in our experiments, and focus on studying the amount of energy traded and the performance of the system.

Figure 8 shows the total production and consumption across this microgrid, as well as the total energy traded per interval using our platform. The horizontal axis shows the starting time for each of the 96 intervals.

In another simulation, we exercise the hybrid solver architecture by running multiple solvers, and after some time, cause one to fail. This result is shown in Figure 9. The narrow vertical red line indicates when solver 1 fails at 8:15AM. Up until that point, solver 1 submitted the green, yellow, light blue, orange solutions, with the final solution being red.On the other hand we see that solver 2 continues to provide solutions for later intervals.

5. Related Research

Online Information Management Platforms: Smart and connected community systems are designed to collect, process, transmit, and analyze data. In this context, data collection usually happens at the edge because that is where edge devices with sensors are deployed to monitor surrounding environments. *SolidWorx* does not suggest a specific data collection methodology. Rather, it follows an actor-driven design pattern where "prosumer" actors can integrate their own agents into *SolidWorx* by using the provided APIs. Another concern of these platforms is the cost of processing. Traditionally, this problem was solved using scalable cloud resources in-house [13]. However, *SolidWorx* enables a decentralized ecosystem, where components of the platform can run directly on edge nodes, which is one of the reasons why we designed it to be asynchronous in nature.

To an extent, the information architecture of *SolidWorx* can be compared to dataflow engines [14], [15], [16]. All of these existing dataflow engines use some form of a computation graph, comprising computation nodes and dataflow edges. These engines are designed for batch-processing and/or stream-processing high volumes of data in resource intensive nodes, and do not necessarily provide additional "platform services" like trust management or solver nodes.

Integration with Blockchains: *SolidWorx* integrates a blockchain because it enables the digital representation of resources, such as energy and financial assets, and their secure transfer from one party to another. Further, blockchains constitute an immutable, complete, and fully auditable record of all transactions that have ever occurred in the system. This is in line with the increased interest and commercial adoption of blockchains [17], which has yielded market capitalization surpassing $75 billion USD [18] for Bitcoin and $36 billion USD for Ethereum [19]. Prior work has also considered the security and privacy of IoT and Blockchain integrations [20], [21], [22].

The biggest challenge in these integrated systems comes from computational-complexity limitations and from the complexity of the consensus algorithms. In particular, their transaction-confirmation time is relatively long and variable, primarily due to the widely-used proof-of-work algorithm. Further, blockchain-based computation is relatively expensive, which is the main reason why we separated finding a solution and validating the solution into two separate components in *SolidWorx*.

Correctness of Smart Contracts: Both verification and automated vulnerability discovery are considered in the literature for identifying smart-contract vulnerabilities. For example, Hirai performs a formal verification of a smart contract that is used by the Ethereum Name Service [23]. However, this verification proves only one property and

it involves relatively large amount of manual analysis. In later work, Hirai defines the complete instruction set of the Ethereum Virtual Machine in Lem, a language that can be compiled for interactive theorem provers [24]. Using this definition, certain safety properties can be proven for existing contracts.

Bhargavan et al. outline a framework for verifying the safety and correctness of Ethereum smart contracts [25]. The framework is built on tools for translating Solidity and Ethereum Virtual Machine bytecode contracts into F^*, a functional programming language aimed at program verification. Using the F^* representations, the framework can verify the correctness of the Solidity-to-bytecode compilation as well as detect certain vulnerable patterns. Luu et al. provide a tool called OYENTE, which can analyze smart contracts and detect certain typical security vulnerabilities [26]. The main difference between prior work and the approach that we are using (i.e., verifying FSolidM models with NuSMV) is that the former can prevent a set of typical vulnerabilities, but they are not effective against vulnerabilities that are atypical or belong to types which have not been identified yet.

6. Conclusion

Smart and connected community applications require decentralized and scalable platforms due to the large number of participants and the lack of mutual trust between them. In this paper, we introduced a transactive platform for resource allocation, called *SolidWorx*. We first formulated a general problem that can be used to represent a variety of resource allocation problems in smart and connected communities. Then, we described an efficient and trustworthy platform based on a hybrid approach, which combines the efficiency of traditional computing environments with the trustworthiness of blockchain-based smart contracts. Finally, we demonstrated the applicability of our platform using two case studies based on real-world data.

Acknowledgement: This work was funded in part by a grant from Siemens, CT and in part by a grant from NSF under award number CNS-1647015. The views presented in this paper are those of the authors and do not reflect the opinion or endorsement of Siemens, CT and NSF.

References

[1] R. B. Melton, "Gridwise transactive energy framework," Pacific Northwest National Laboratory, Tech. Rep., 2013.

[2] A. Mavridou and A. Laszka, "Designing secure Ethereum smart contracts: A finite state machine based approach," in *Proceedings of the 22nd International Conference on Financial Cryptography and Data Security (FC)*, February 2018.

[3] G. A. Agha, "Actors: A model of concurrent computation in distributed systems," Massachusetts Institute of Technology, Artificial Intelligence Lab, Tech. Rep., 1985.

[4] A. Basu, B. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis, "Rigorous component-based system design using the bip framework," *IEEE Software*, vol. 28, no. 3, pp. 41–48, 2011.

[5] S. Eisele, I. Mardari, A. Dubey, and G. Karsai, "Riaps: Resilient information architecture platform for decentralized smart systems," in *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, May 2017, pp. 125–132.

[6] J. Bergquist, A. Laszka, M. Sturm, and A. Dubey, "On the design of communication and transaction anonymity in blockchain-based transactive microgrids," in *Proceedings of the 1st Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers (SERIAL)*. ACM, 2017, pp. 3:1–3:6.

[7] A. Laszka, A. Dubey, M. Walker, and D. Schmidt, "Providing privacy, safety and security in iot-based transactive energy systems using distributed ledgers," in *Proceedings of the 7th International Conference on the Internet of Things*, 2017.

[8] IBM ILOG CPLEX, "V12. 1: Users manual for CPLEX," *International Business Machines Corporation*, vol. 46, no. 53, p. 157, 2009.

[9] J. Sifakis, "Rigorous system design," *Foundations and Trends in Electronic Design Automation*, vol. 6, no. 4, pp. 293–362, 2013.

[10] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella, "Nusmv 2: An opensource tool for symbolic model checking," in *International Conference on Computer Aided Verification*. Springer, 2002, pp. 359–364.

[11] C. Baier, J.-P. Katoen, and K. G. Larsen, *Principles of model checking*. MIT press, 2008.

[12] H. Diedrich, *Ethereum: Blockchains, Digital Assets, Smart Contracts, Decentralised Autonomous Organisations*. CreateSpace Independent Publishing Platform, 2016. [Online]. Available: https://books.google.com/books?id=Y2YRvgAACAAJ

[13] D. C. Schmidt, J. White, and C. D. Gill, "Elastic infrastructure to support computing clouds for large-scale cyber-physical systems," in *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2014, pp. 56–63.

[14] Apache Software Foundation, "Apache Storm," http://storm.apache.org/.

[15] ——, "Apache Spark," http://spark.apache.org/.

[16] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. IEEE, 2010, pp. 170–177.

[17] M. Iansiti and K. Lakhani, "The truth about blockchain," https://hbr.org/2017/01/the-truth-about-blockchain, January 2017, (accessed on 08/30/2017).

[18] CoinMarketCap, "Bitcoin (BTC) price, charts, market cap, and other metrics," https://coinmarketcap.com/currencies/bitcoin/, August 2017, (accessed on 08/30/2017).

[19] ——, "Ethereum (ETH) $381.84 (3.83%)," https://coinmarketcap.com/currencies/ethereum/, August 2017, (accessed on 08/30/2017).

[20] A. Dorri, S. S. Kanhere, R. Jurdak, and P. Gauravaram, "Blockchain for iot security and privacy: The case study of a smart home," in *2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, 2017, pp. 618–623.

[21] A. Ouaddah, A. A. Elkalam, and A. A. Ouahman, "Towards a novel privacy-preserving access control model based on blockchain technology in IoT," in *Europe and MENA Cooperation Advances in Information and Communication Technologies*, 2017, pp. 523–533.

[22] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016.

[23] Y. Hirai, "Formal verification of deed contract in Ethereum name service," https://yoichihirai.com/deed.pdf, November 2016.

[24] ——, "Defining the Ethereum Virtual Machine for interactive theorem provers," in *1st Workshop on Trusted Smart Contracts, in conjunction with the 21st International Conference of Financial Cryptography and Data Security (FC)*, April 2017.

[25] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin, "Short paper: Formal verification of smart contracts," in *11th ACM Workshop on Programming Languages and Analysis for Security (PLAS), in conjunction with ACM CCS 2016*, October 2016, pp. 91–96.

[26] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 254–269.