Modeling Smart Contracts with Probabilistic Logic Programming

Damiano Azzolini[1], Fabrizio Riguzzi[2], and Evelina Lamma[1]

[1] Dipartimento di Ingegneria – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[2] Dipartimento di Matematica e Informatica – University of Ferrara
Via Saragat 1, I-44122, Ferrara, Italy
[damiano.azzolini,fabrizio.riguzzi,evelina.lamma]@unife.it

Abstract. Smart contracts are computer programs that run in a distributed network, the blockchain. These contracts are used to regulate the interaction among parties in a fully decentralized way without the need of a trusted authority and, once deployed, are immutable. The immutability property requires that the programs should be deeply analyzed and tested, in order to ensure that they behave as expected and to avoid bugs and errors. In this paper, we present a method to translate smart contracts into probabilistic logic programs that can be used to analyse expected values of several smart contract's utility parameters and to get a quantitative idea on how smart contracts variables changes over time. Finally, we applied this method to study three real smart contracts deployed on the Ethereum blockchain.

Keywords: blockchain, probabilistic logic programming, smart contracts.

1 Introduction

The idea of the blockchain model dates back to 1990 [12] as a method to secure timestamping digital documents, but the interest around this technology grew only after the success of the paper by Nakamoto in 2008 [18]. Afterwards, the term *smart contract*, used to identify programs written in a quasi-Turing-complete programming language that run in a blockchain environment, gained traction, thanks to the possibility of enforcing contracts between two or more parties without the need of a central authority.

Smart contracts and blockchain technology are relevant to several research fields. In distributed systems the researchers study the interaction among peers in a fully decentralized environment with an unreliable network and create methods to ensure a decentralized consensus, even in case of dishonest parties. Formal methods are used to verify the behavior of smart contracts. Cryptography encompasses methods to ensure that all the participants can see the same data and that data have not been tampered with. Game theory and decision theory are used to model the behavior of the interacting parties to maximize the expected profit.

The execution of a smart contract is deterministic, i.e., it always yields the same result. However, because users can interact with it at their will, a probabilistic analysis is needed to predict its behavior. From an issuer's perspective, he may be interested in how many people interact with the contract and how profit values evolve. Similarly, from a user's perspective, he can be interested in the expected reward by interacting with a blockchain based game, such as gambling.

In this paper, we propose a method to translate smart contracts into probabilistic logic programs in order to compute the expected values of several smart contract's utility parameters. Furthermore, our approach can also be used to identify some coding errors. Several tools exist for identifying bugs but almost all of them provide a static analysis without computing a quantitative amount of the possible monetary loss. Moreover, translating a smart contract into a probabilistic logic language allows interaction with it in a practical way, without the need of a specialized checking tool for every smart contract language.

The paper is organized as follows: in Section 2 we briefly present blockchain technology and smart contracts. In Section 3 we introduce Probabilistic Logic Programming. In Section 4 we propose a method to translate smart contracts into probabilistic logic programs. In Section 5 we analyse three smart contracts that model real world applications and Section 6 concludes the paper.

2 Blockchain and Smart Contracts

Smart contracts were initially proposed in 1994 [26] as computer protocols to facilitate a self-enforcing agreements between two parties. Smart contracts received an increased attention only after the interest around blockchain technologies exploded, following the publication of [18]. Currently, the term smart contract is always used in the context of a blockchain environment.

In a nutshell, a blockchain is a decentralized, distributed, append only (usually) public ledger maintained by a set of peers, that records transactions between accounts. All the transactions are organized in a chain of blocks (hence the origin of the term blockchain) linked together using hash functions that guarantee the consistency and the immutability of the stored data. Each honest (full) node that follows a blockchain protocol stores his own updated copy of the ledger. In order to guarantee consistency of the data, i.e., that all the peers see the same copy of the ledger, a consensus must be reached: this is done by adopting a so-called *consensus algorithms*, such as Proof-of-Work (PoW).

Blockchain is the underlying technology of several decentralized platforms, such as Bitcoin and Ethereum. Ethereum is a decentralized transaction-based state machine [31] that executes smart contracts written in a quasi-Turing-complete bytecode language. Programmers usually develop smart contracts using a programming language called Solidity which is translated, by a compiler, into bytecode for execution. In this paper, when we write smart contract we mean a smart contract deployed on Ethereum. However, our method can be extended to general smart contracts of which source code is available. In case of Ethereum,

source code is not stored in the blockchain (only a hash of the code is stored) but there are several platforms that allows developers to upload the code of a contract and verify it, such as Etherscan[3]. This website checks if the compilation of the uploaded code matches the bytecode stored into the blockchain. If so, the smart contract is considered verified and the source code is made public.

3 Probabilistic Logic Programming

A wide variety of domains can be represented using Probabilistic Logic Programming (PLP) languages under the distribution semantics [22,25]. A program in a language adopting the distribution semantics defines a probability distribution over normal logic programs called *instances* or *worlds*. Each normal program is assumed to have a total well-founded model [27]. Then, the distribution is extended to queries and the probability of a query is obtained by marginalizing the joint distribution of the query and the programs. A PLP language under the distribution semantics with a general syntax is that of *Logic Programs with Annotated Disjunctions* (LPADs) [29]. In the following part we present the semantics of LPADs for the case of no function symbols, if function symbols are allowed see [21].

Heads of clauses in LPADs are disjunctions in which each atom is annotated with a probability. Consider an LPAD T with n clauses: $T = \{C_1, \ldots, C_n\}$. Each clause C_i takes the form: $h_{i1} : \Pi_{i1}; \ldots; h_{iv_i} : \Pi_{iv_i} :- b_{i1}, \ldots, b_{iu_i}$, where $h_{i1}, \ldots, h_{iv_i}$ are logical atoms, $b_{i1}, \ldots, b_{iu_i}$ are logical literals and $\Pi_{i1}, \ldots, \Pi_{iv_i}$ are real numbers in the interval $[0, 1]$ that sum to 1. $b_{i1}, \ldots, b_{iu_i}$ is indicated with $body(C_i)$. Note that, if $v_i = 1$ the clause corresponds to a non-disjunctive clause. We also allow clauses where $\sum_{k=1}^{v_i} \Pi_{ik} < 1$: in this case the head of the annotated disjunctive clause implicitly contains an extra atom *null* that does not appear in the body of any clause and whose annotation is $1 - \sum_{k=1}^{v_i} \Pi_{ik}$. We define *substitution* θ a function mapping variables to terms. Usually θ has the form $\theta = \{X_1/t_1, ..., X_k/t_k\}$ meaning that each variable X_i is substituted by the term t_i. Applying a substitution θ to a LPAD T means replacing all the occurrences of each variable X_j in ϕ by the corresponding term t_j. For an exhaustive treatment of PLPs see [22].

Given an LPAD T, the main task is to perform *inference*. There are two types of inference: exact inference and approximate inference. Both exact and approximate inference are implemented into the suite *cplint* [1,23].

3.1 Exact Inference

The main goal of exact inference is to solve tasks in an exact way, without approximation. Various approaches have been presented for performing inference on LPADs, such as PITA [24]. Starting from an LPAD, PITA performs inference using knowledge compilation [10] to Binary Decision Diagrams (BDD).

[3] `https://etherscan.io/`

The exact inference task is in general #P-complete [16] so it is not tractable for certain domains. In these cases, *approximate* inference is needed. We will not use exact inference in this paper and so we will not cover this topic further in detail.

3.2 Approximate Inference

In cplint, approximate inference is performed using Monte Carlo algorithms [5, 20]. Using these algorithms, the possible worlds are sampled and the query is tested in the samples. The estimated probability of the query is then given by the fraction of the sampled worlds where the query succeeds.

Consider a simple example were a coin is toss with uncertainty on its fairness. Our goal is to find the probability that it lands head or tail. A probabilistic logic program to model this scenario may be:

```
heads(Coin):0.5; tails(Coin):0.5 :-
  toss(Coin),not(biased(Coin)).
heads(Coin):0.6; tails(Coin):0.4 :-
  toss(Coin),biased(Coin).
fair(Coin):0.9; biased(Coin):0.1.
toss(Coin).
```

The program states: if we toss a coin that is not biased, then it lands heads or tails with the same probability 0.5. If we toss a coin that is biased, then it lands heads with probability 0.6 and tails with probability 0.4. We express our uncertainty on the bias of the coin supposing that it is fair with probability 0.9 and biased with probability 0.1. Finally, we say that the coin is certainly tossed.

To compute the probability that the coin lands head, using the module MCINTYRE from cplint. For example, we can sample `heads(Coin)` a certain number of times and compute the probability that it's biased using `?- mc_sample(heads(coin),1000,P)`. It's also possible to compute expectations by sampling using `mc_expectation/4` to get, for instance, the number of consecutive toss landing head. cplint also allows the definition of probability densities using `A:Density:- Body`. For instance, `g(X): gaussian(X,0, 1)` states that argument `X` of `g(X)` follows a Gaussian distribution with mean `0` and variance `1`.

4 Modelling Smart Contracts with Probabilistic Logic Programming

Probabilistic Logic Programming [11] has been successfully applied in many different fields where probability has a key role, such as natural language processing, link prediction in social networks, model checking and also Bitcoin protocol [3, 4, 19]. While smart contracts are, by definition, deterministic (i.e., the execution of a smart contract's function must always yield the same result), the

interaction among users of the same smart contract can be seen as probabilistic because the involved parties are not under the control of the issuer.

From a smart contract issuer's perspective, he can be interested in the expected value of certain profit variables, such as the collected fees (which can be a fixed fraction of the user input value), the expected number of tokens sold in a certain amount of time or the tokens distribution after several transactions among users. From the opposite perspective, the user's perspective, he could be interested in the expected reward from participating in a smart contract game, such as gambling, which is, at the moment of writing, among the smart contract categories with most interactions (according to `DappRadar`[4]).

These values can be computed by translating smart contracts into probabilistic logic programs. This process is composed of two steps: the smart contract is translated into a Logic Program and then probabilistic facts are added to turn it into a PLP. The translation of a smart contract function into a logic programming predicate is straightforward. Here we use SWI-Prolog [30] implementation of the programming language Prolog. Every function can be translated into a predicate with the same name and the same number of arguments. Moreover, if needed in the logic flows of the function, the members of the globally available `msg` object, such as `msg.value` or `msg.sender`, can be added to the arguments list of the predicate. Since a Prolog programs does not allow the `return` keyword, all the values that are returned from a Solidity function must be added as further arguments of the predicate.

Listing 1.1 shows a simple example of a smart contract written in Solidity simulating a bank. The function `constructor` is executed only at the creation of the contract: it sets the owner of the contract and issues 1000 tokens to the creator. Each user has the possibility to call `transfer`. This function accepts two parameters, the address of the receiver and the amount the user wants to transfer to the receiver. First, it verifies that the user has enough funds to perform the operation and that the sender is different from the receiver. Both conditions are evaluated using `require`, a built-in function that checks the condition passed as input and throws an exception if it is not met. Then, the balances of both parties are updated accordingly.

To simulate the storage reserved to a smart contract, it is possible to add two further arguments to the predicate arguments list, one for the input list and one for the output list: using a user-defined predicate called `find/3`, we retrieve the balance of the sender (identified with `Sender`) from `BalanceList` and we store it into `BalanceSender`. Similarly, we retrieve the balance of the receiver. Then, we perform the checks corresponding to `require` in listing 1.1. Finally, we update the balances and update the list (generating another list with the old balances replaced with the new one for both parties, since Prolog lists do not allow modifications) using another user-defined predicate called `update/3`.

Once the contract is translated into SWI-Prolog code, we can add some probabilistic facts. For instance, we may suppose that the transferred amount from a user to another user (this transfer can be seen also as placing a bet, by

[4] `https://dappradar.com/`

```
contract simpleBank {
    address owner;
    mapping(address => uint) balances;

    constructor() public {
        owner = msg.sender;
        balances[msg.sender] = 1000;
    }

    function transfer(address receiver, uint amt) public {
        require(balances[msg.sender] >= amt);
        require(msg.sender != receiver);
        balances[msg.sender] -= amt;
        balances[receiver] += amt;
    }
}
```

Listing 1.1. Example of Solidity smart contract.

transferring funds to the address where the smart contract is stored) is uniformly distributed between 0.5 and 2 Ether. This uncertainty can be expressed with `amount(A):uniform(A,0.5,2.0)`. The complete code is shown listing 1.2.

```
amount(A):uniform(A,0.5,2.0).
transfer(Receiver,Amt,Sender,BalanceList,NewBalanceList):-
    find(Sender,BalanceList,BalanceSender),
    find(Receiver,BalanceList,BalanceReceiver),
    amount(Amt),
    BalanceSender >= Amt,
    Sender \= Receiver,
    NewBalanceS is BalanceSender - Amt,
    NewBalanceR is BalanceReceiver + Amt,
    update(BalanceList,Sender,NewBalanceS,NewBalanceList1),
    update(BalanceList1,Receiver,NewBalanceR,NewBalanceList).
```

Listing 1.2. Example of smart contract translated into a probabilistic logic program.

The parameter `Amt` is now sampled from the uniform distribution specified above. Finally, to query the program to get, for instance, the expected transferred value, we can use `mc_expectation/4`. The following section shows how we applied this method to three smart contracts deployed and publicly accessible on the Ethereum mainnet. The usage of a PLP language makes it possible to test the smart contract without deploying it into a test net. Moreover, a logical language for smart contracts makes the contract simpler and easier to debug.

5 Experiments

To conduct our experiments we modelled three smart contracts written in Solidity taken from Etherscan. In the experiments, we analyzed a smart con-

tract for transferring tokens, one for a Ponzi scheme and one of a gambling game. All the experiments are conducted on a cluster[5] with Intel® Xeon® E5-2630v3 running at 2.40 GHz. The execution time is computed using the built-in SWI-Prolog predicate `statistics/2` and the memory usage is the value `maxresident` computed using GNU Time[6]. For each experiment, we used the predicate `mc_expectation/4`, available in cplint, with 1000 samples.

5.1 Transfer

In this example we model a scenario where `N` users trade (burn or transfer) a certain amount of tokens. The functions `burn` and `transfer` are taken from the code stored at the address `0xB8c77482e45F1F44dE1745F52C74426C631bDD52` on the Ethereum mainnet. Each user starts with 100 tokens. We want to know, for example, how many transfers are needed to produce a situation where a single user has more than 180 tokens. Those two values have been chosen just to demonstrate the process. Transfers among users are done in a random way with the transferred amount uniformly distributed between 1 and 10. Moreover, we include a small probability (5%) that, instead of trading tokens, the user will burn tokens. In Table 1 we show the relation between number of users, execution time of the experiments, memory usage and expected number of transactions. As expected, the number of transfers needed to create a situation in which a user has more than a certain number (180 for this experiment) of tokens increases as the number of users increases.

Consider now the transfer function shown in listing 1.1. The second line checks that the sender of the tokens is different from the receiver. However, in some real-world examples [7], due to coding errors, this check is not present, causing the generation of unexpected extra tokens. The method proposed in this paper is also suitable to spot bugs and coding errors of this type. For modelling this situation, we run the previous experiment but this time we compute the expected value of all circulating tokens, represented as the sum of the balances of all the participants. In every run, the program chooses two random (possibly the same) users from the user's list and performs a transfer of tokens between them. As expected, the number of circulating tokens exceeds the initial amount. The results are presented in Table 2.

5.2 Ponzi Scheme

The boost of blockchain adoption in the last years caused a massive number of smart contracts being issued. In [7] and [28] the authors identify and analyze a huge number of Ponzi schemes deployed in the Bitcoin blockchain. In [6] the analysis was extended to the Ethereum blockchain.

[5] `http://www.fe.infn.it/coka/doku.php?id=start`
[6] `https://www.gnu.org/software/time/`
[7] `https://gist.github.com/loiluu/0363070e1bada977f6192c8e78348438`

Table 1. Details for the Transfer experiment (Subsection 5.1).

# of users	Time (s)	Memory (Mb)	Expected Value
5	1.643	52.744	192.724
25	8.717	102.924	421.516
50	23.431	155.636	661.514
75	44.172	202.428	874.234
100	71.367	246.136	1071.335
125	101.48	285.208	1249.186
150	140.116	327.488	1441.242

Table 2. Details for the Transfer experiment (Subsection 5.1) with bug.

# of users	Time (s)	Memory (Mb)	Initial Amount	Final Amount
5	0.755	33.324	500	627.262
25	4.967	95.996	2500	2592.483
50	12.834	154.196	5000	5077.37
75	23.828	204.036	7500	7568.241
100	37.663	253.204	10000	10064.181
125	53.451	294.352	125000	12561.246
150	72.883	344.072	150000	15058.114

A Ponzi scheme is a financial fraud that promises a high return of the investment: the profit increases as long as the number of people involved in the schema increases. However, this type of business model, and in particular pyramid schemes, a variant of Ponzi scheme, quickly become unsustainable due to the need of constant increase in participants to be profitable. Probability models in these cases are fundamental for analyzing the expected reward (payoff) and avoid being cheated.

For this experiment, we collect the code for a well-known pyramid schema called Rubixi, stored at `0xe82719202e5965Cf5D9B6673B7503a3b92DE20be`. This code is often used to show a critical vulnerability of smart contracts which allow anyone to become the owner of the contract and withdraw the collected fees [2]. In this experiment we ignore this problem as it is out of the scope of this example. The logic of Rubixi is simple: a user can send some Ether, at least 1, to the contract through the *fallback* function. When receiving of the amount, the contract collects the Ether, adds the new participant to the participants list and redistributes the accumulated value to the other participants if certain conditions are met. Several considerations can be done for this contract. For instance: what is the amount of collected fees in a certain amount of time? What is the number of participants we have to wait for receiving a payment? We modelled the contract in order to answer the second question supposing that users will send an amount of tokens to the contract uniformly distributed between 0.9 and 2, since we consider also a situation where a user sends an amount less than the minimum required. In this case the amount is only added to the collected fees but the participant is not considered. Our results are shown in Table 3.

Table 3. Details for the Ponzi Scheme experiment (Subsection 5.2). The last column relates the number of users to the amount of reward distributed. For instance, the fifth user entering the scheme has to wait 13 additional people to enter the schema before getting paid.

# of users	Time (s)	Memory (Mb)	# of users to wait
5	0.159	13.384	13.783
50	1.997	37.232	69.814
100	6.111	59.384	111.138
150	12.681	85.768	152.868
200	22.051	113.436	194.760
300	50.852	174.420	278.393
400	92.934	225.832	361.172

5.3 Gambling

According to the DappRadar website, in April 2020, 5 out of 10 most used dApps (web applications with the logic implemented on a smart contract) are gambling games. In this section we analyse a smart contract implementing a gambling platform stored at the address `0x999999C60566e0a78DF17F71886333E1dACE0BAE` of the Ethereum mainnet. It allows a player to bet on several games such as dice, roulette or poker. The outcome is computed considering several payout masks. Randomization is obtained using a `commit` value, externally generated, provided as input to the bet, combined with other values. The experiments were conducted simulating a player betting on the outcome of a single die an amount distributed with a Poisson distribution with several values of mean, and transaction fees uniformly distributed between 0.07 and 0.2 *Finney* (1 Finney = 10^{-3} Ether), according to the data taken from `BitInfoCharts`[8]. The results obtained are shown in Figure 1 and Table 4. As expected, the expected payout decreases in relation to the bet amount and the number of trials.

Table 4. Resource usage for the Gambling experiment (Subsection 5.3) with $\lambda = 150$.

# of trials	Time (s)	Memory (Mb)
5	0.517	9.344
50	4.890	11.016
100	8.757	18.924
150	12.938	28.22
200	17.439	27.076
300	27.011	143.896
400	37.437	201.996

[8] `https://bitinfocharts.com/ethereum/`

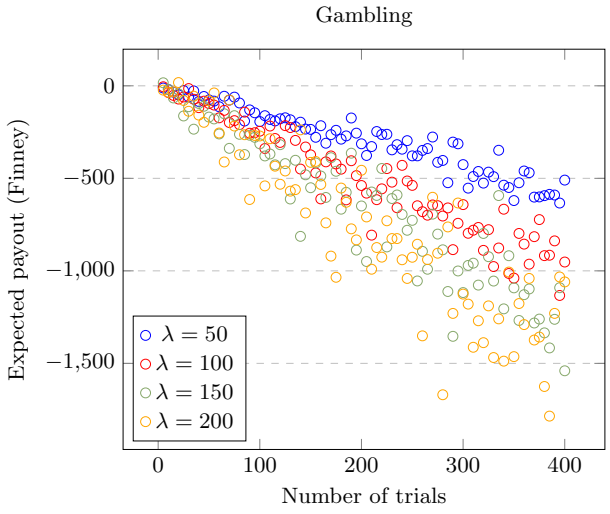

Fig. 1. Graph showing the expected payout of consecutive number of trials. λ represents the mean of the Poisson distribution.

6 Conclusions

In this paper we show that Probabilistic Logic Programming (PLP) can be applied to quantitatively model the behavior of smart contracts. We proposed an approach to translate smart contracts into probabilistic logic programs, independently from the contract language used, and we modelled three smart contracts taken from the Ethereum mainnet (i.e., Transfer, Ponzi Scheme and Gambling) in PLP.

Most of the literature about smart contracts is focused on vulnerability analysis and bug detection. There is a lot of work in the literature about automatic verification of smart contracts written using Solidity. For example, in [17] the authors used static analysis tools to automatically find bugs. Another approach can be found in [15] where symbolic model checking has been used. In [13] the Ethereum Virtual Machine has been defined in a language that can be compiled and understood by theorem provers in order to check some safety properties. All these methods perform well in finding bugs but do not return a quantitative impact of a bug in the contract (for instance, the number of tokens lost). An approach similar to the one proposed in this paper can be found in [8] where the authors developed their own framework to extract some utility values based on game theoretic considerations. Differently from them we focus on Probabilistic Logic Programming and for the experiments we rely on existing and well-studied tools instead of developing a new framework, taking advantage from the expressiveness of PLP and its underlying inference system. Moreover, our experiments can be performed using cplint on SWISH[9] [23] accessible through a web browser.

[9] `http://cplint.eu/`

A natural extension of our work could be developing a tool that automatically translates smart contracts into probabilistic logic programs to analyse further examples. Another idea is to extend the analysis including a decision theory approach to select a set of actions to perform in order to maximize a reward value. An interesting future work could be also to extend the probabilistic analysis to the blockchain technology, as done in [4], to model several scenarios, such as congestion of the network. Another future work is the design of a logic-based smart contract language or architecture, as suggested in [9] and [14] or probabilistic logic-based smart contract to directly allow probabilistic computation to be performed on the blockchain.

References

1. Alberti, M., Cota, G., Riguzzi, F., Zese, R.: Probabilistic logical inference on the web. In: Adorni, G., Cagnoni, S., Gori, M., Maratea, M. (eds.) AI*IA 2016 Advances in Artificial Intelligence. Lecture Notes in Computer Science, vol. 10037, pp. 351–363. Springer, Berlin (2016)
2. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: International Conference on Principles of Security and Trust. pp. 164–186. Springer (2017)
3. Azzolini, D., Riguzzi, F., Lamma, E.: Studying transaction fees in the bitcoin blockchain with probabilistic logic programming. Information 10(11), 335 (2019)
4. Azzolini, D., Riguzzi, F., Lamma, E., Bellodi, E., Zese, R.: Modeling bitcoin protocols with probabilistic logic programming. In: Bellodi, E., Schrijvers, T. (eds.) Proceedings of the 5th International Workshop on Probabilistic Logic Programming, PLP 2018, co-located with the 28th International Conference on Inductive Logic Programming (ILP 2018), Ferrara, Italy, September 1, 2018. CEUR Workshop Proceedings, vol. 2219, pp. 49–61. CEUR-WS.org (2018)
5. Azzolini, D., Riguzzi, F., Lamma, E., Masotti, F.: A comparison of MCMC sampling for probabilistic logic programming. In: Alviano, M., Greco, G., Scarcello, F. (eds.) Proceedings of the 18th Conference of the Italian Association for Artificial Intelligence (AI*IA2019), Rende, Italy 19-22 November 2019. Lecture Notes in Computer Science, Springer, Heidelberg, Germany (2019)
6. Bartoletti, M., Carta, S., Cimoli, T., Saia, R.: Dissecting ponzi schemes on ethereum: identification, analysis, and impact. arXiv preprint arXiv:1703.03779 (2017)
7. Bartoletti, M., Pes, B., Serusi, S.: Data mining for detecting bitcoin ponzi schemes. In: Crypto Valley Conference on Blockchain Technology, CVCBT 2018, Zug, Switzerland, June 20-22, 2018. pp. 75–84. IEEE (2018)
8. Chatterjee, K., Goharshady, A.K., Velner, Y.: Quantitative analysis of smart contracts. In: European Symposium on Programming. pp. 739–767. Springer, Cham (2018)
9. Ciatto, G., Calegari, R., Mariani, S., Denti, E., Omicini, A.: From the blockchain to logic programming and back: Research perspectives. In: Cossentino, M., Sabatucci, L., Seidita, V. (eds.) Proceedings of the 19th Workshop "From Objects to Agents", Palermo, Italy, June 28-29, 2018. CEUR Workshop Proceedings, vol. 2215, pp. 69–74. CEUR-WS.org (2018)
10. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. 17, 229–264 (2002)

11. De Raedt, L., Kimmig, A.: Probabilistic (logic) programming concepts. Mach. Learn. 100(1), 5–47 (2015)
12. Haber, S., Stornetta, W.S.: How to time-stamp a digital document. In: Conference on the Theory and Application of Cryptography. pp. 437–455. Springer (1990)
13. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: International Conference on Financial Cryptography and Data Security. pp. 520–535. Springer (2017)
14. Idelberger, F., Governatori, G., Riveret, R., Sartor, G.: Evaluation of logic-based smart contracts for blockchain systems. In: International Symposium on Rules and Rule Markup Languages for the Semantic Web. pp. 167–183. Springer (2016)
15. Kalra, S., Goel, S., Dhawan, M., Sharma, S.: Zeus: Analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018 (2018)
16. Koller, D., Friedman, N.: Probabilistic Graphical Models: Principles and Techniques. Adaptive computation and machine learning, MIT Press, Cambridge, MA (2009)
17. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. ACM (2016)
18. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
19. Nguembang Fadja, A., Riguzzi, F.: Probabilistic logic programming in action. In: Holzinger, A., Goebel, R., Ferri, M., Palade, V. (eds.) Towards Integrative Machine Learning and Knowledge Extraction, LNCS, vol. 10344. Springer (2017)
20. Riguzzi, F.: MCINTYRE: A Monte Carlo system for probabilistic logic programming. Fund. Inform. 124(4), 521–541 (2013)
21. Riguzzi, F.: The distribution semantics for normal programs with function symbols. Int. J. Approx. Reason. 77, 1 – 19 (October 2016)
22. Riguzzi, F.: Foundations of Probabilistic Logic Programming. River Publishers, Gistrup, Denmark (2018)
23. Riguzzi, F., Bellodi, E., Lamma, E., Zese, R., Cota, G.: Probabilistic logic programming on the web. Softw.-Pract. Exper. 46(10), 1381–1396 (10 2016)
24. Riguzzi, F., Swift, T.: The PITA system: Tabling and answer subsumption for reasoning under uncertainty. Theor. Pract. Log. Prog. 11(4–5), 433–449 (2011)
25. Sato, T.: A statistical learning method for logic programs with distribution semantics. In: Sterling, L. (ed.) ICLP 1995. pp. 715–729. MIT Press (1995)
26. Szabo, N.: Smart contracts (1994)
27. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM 38(3), 620–650 (1991)
28. Vasek, M., Moore, T.: Analyzing the bitcoin ponzi scheme ecosystem. In: International Conference on Financial Cryptography and Data Security. pp. 101–112. Springer (2018)
29. Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic Programs With Annotated Disjunctions. In: ICLP 2004. LNCS, vol. 3132, pp. 431–445. Springer (2004)
30. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theor. Pract. Log. Prog. 12(1-2), 67–96 (2012)
31. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper 151, 1–32 (2014)