Developing Secure Bitcoin Contracts with BitML

Nicola Atzei
University of Cagliari
Italy
atzeinicola@gmail.com

Massimo Bartoletti
University of Cagliari
Italy
bart@unica.it

Stefano Lande
University of Cagliari
Italy
lande@unica.it

Nobuko Yoshida
Imperial College London
UK
n.yoshida@imperial.ac.uk

Roberto Zunino
University of Trento
Italy
roberto.zunino@unitn.it

ABSTRACT

We present a toolchain for developing and verifying smart contracts that can be executed on Bitcoin. The toolchain is based on BitML, a recent domain-specific language for smart contracts with a computationally sound embedding into Bitcoin. Our toolchain automatically verifies relevant properties of contracts, among which *liquidity*, ensuring that funds do not remain frozen within a contract forever. A compiler is provided to translate BitML contracts into sets of standard Bitcoin transactions: executing a contract corresponds to appending these transactions to the blockchain. We assess our toolchain through a benchmark of representative contracts.

Demo Video URL: https://youtu.be/bxx3bM5Pm6c

CCS CONCEPTS

• **Software and its engineering → Software verification**; • **Security and privacy → Distributed systems security**.

KEYWORDS

Bitcoin; smart contracts; verification

ACM Reference Format:
Nicola Atzei, Massimo Bartoletti, Stefano Lande, Nobuko Yoshida, and Roberto Zunino. 2019. Developing Secure Bitcoin Contracts with BitML. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3338906.3341173

1 INTRODUCTION

In the last five years much outstanding research has been devoted to showing how to exploit Bitcoin to execute *smart contracts* — computer protocols which allow for exchanging cryptocurrency according to complex pre-agreed rules [4–7, 10, 12, 15, 27–30, 33]. Despite the wide variety of use cases witnessed by these works, no tool support has been provided yet to facilitate the development of Bitcoin contracts. Today, this task requires to devise complex protocols which, besides using the standard cryptographic primitives, can read and append transactions on the Bitcoin blockchain. Creating a new protocol requires a significant effort to establish its correctness and security: this is an error-prone task, usually performed manually.Crafting the transactions used by these protocols is burdensome as well, since it requires to struggle with low-level, poorly documented features of Bitcoin.

In this paper we consider BitML, a recent high-level language for smart contracts, featuring a computationally sound embedding into Bitcoin [13], and a sound and complete verification technique of relevant trace properties [14]. BitML can express many of the smart contracts in the literature [8, 11], and execute them by appending suitable transactions to the Bitcoin blockchain. The computational soundness of the embedding guarantees that security properties at the level of the BitML semantics are preserved at the level of Bitcoin transactions, even in the presence of adversaries. Still, BitML lives in a theoretical limbo, as no tool support exists yet to develop contracts and deploy them on the Bitcoin blockchain.

Contributions. We develop a toolchain for writing and verifying BitML contracts, and for deploying them on Bitcoin. More specifically, our main contributions can be summarised as follows:

(1) A BitML embedding in Racket [22], which allows for programming BitML contracts within the DrRacket IDE.
(2) A security analyzer which can check arbitrary LTL properties of BitML contracts. In particular, the analysis can decide *liquidity*, a landmark property of smart contracts requiring that the funds within a contract do not remain frozen forever.
(3) A compiler from BitML contracts to standard Bitcoin transactions. The computational soundness result in [13] ensures that attacks to compiled contracts are also observable at the BitML level. Therefore, the properties verified by our security analyzer also hold for compiled contracts.
(4) A collection of BitML contracts, which we use as a benchmark to evaluate our toolchain. This collection contains some of the most complex contracts ever developed for Bitcoin, e.g. financial services, auctions, timed commitments, lotteries, and a variety of other gambling games. We use our benchmarks to discuss the expressiveness and the limitations of Bitcoin as a smart contracts platform.

The architecture of our toolchain is displayed in Figure 1. The development workflow is the following: (a) write the BitML contract, and specify the required properties. Optionally, specify some

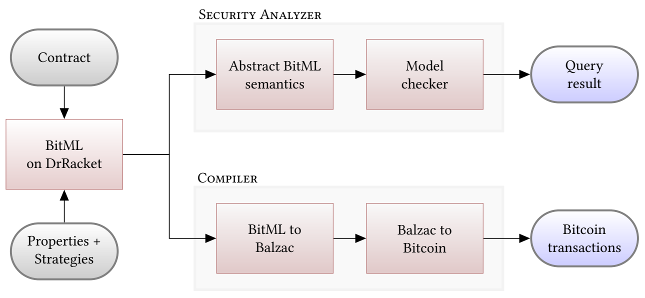

Figure 1: Toolchain architecture.

constraints on the participants' strategies, e.g. to partially define the behaviour of the honest participants; (b) verify that the contract satisfies the required properties through the security analyzer; (c) compile the contract to Bitcoin transactions; (d) execute the contract, by appending these transactions to the Bitcoin blockchain according to the chosen strategy. We remark that the last step can be performed on the Bitcoin main network, without requiring any extensions or customizations. Our toolchain is open-source[1], as well as the contracts in our benchmark. A tutorial is available online[2], including references to our experiments on the Bitcoin testnet.

2 DESIGNING BITML CONTRACTS

BitML contracts allow two or more participants to exchange their bitcoins (₿) according to a given logic. A contract consists of two parts: a precondition, describing requirements that participants must fulfil to stipulate the contract, and a process, which specifies the execution logic of the contract. Here, rather than providing the syntax and semantics of BitML (see [13] for a formalization), we illustrate it through a simple but paradigmatic example, the *mutual timed commitment* contract [6]. This contract involves two participants (named below A and B) each one choosing a secret and depositing a certain amount of cryptocurrency (say, 1₿). The goal of the contract is to ensure that each participant will either learn the other participant's secret, or otherwise receive the other participant's deposit as a compensation. The contract gives some time to the participants to reveal their secrets. If a participant reveals her secret in time, then she can get her deposit back; otherwise, after the time is up, the other participant can withdraw that deposit.

In our tool, we can specify this contract as follows:

```
(participant "A" "029c...cced") ; A's public key
(participant "B" "022c...af30") ; B's public key

(contract
  (pre
    (deposit "A" 1 "1a34...6f38@0") ; tx output (1BTC)
    (secret  "A" a "628f...de71")   ; hash of A's secret
    (deposit "B" 1 "19e7...85ff@2") ; tx output (1BTC)
    (secret  "B" b "9d48...bb35"))  ; hash of B's secret

  (choice
    (reveal (a) (choice
                  (reveal (b) (split
                                (1 -> (withdraw "A"))
                                (1 -> (withdraw "B"))))
                  (after 100050 (withdraw "A"))))
    (after 100000 (withdraw "B")))
```

[1] https://github.com/bitml-lang
[2] https://blockchain.unica.it/bitml

The first two lines create aliases for the participant names, specifying their public keys. The contract preconditions are in the `pre` part: each participant must specify the identifier of a transaction output, and the hash of the chosen secret. The transaction output must be unspent, must contain the required 1₿, and must be redeemable using the participant's private key. The hash is used during the contract execution: when the participant provides a value, claiming that it is the chosen secret, the hash of this value is required to be equal to the one in the precondition.

The contract logic is specified after the preconditions. The top-level `choice` defines two alternative branches of the contract. The first branch can only be taken if A reveals her secret (named a); when this happens, the contract continues with the innermost `choice`. The second branch can only be taken after a timeout, specified as the block at height 100000, and it allows B to redeem all the funds deposited within the contract (i.e., 2₿) by executing `withdraw "B"`. So, to avoid losing her deposit, A is incentivized to reveal her secret in time. Similarly, the innermost `choice` is used to incentivize B to reveal his secret before the block at height 100050. If B reveals, then the `split` subcontract is executed: this divides the balance of the contract in two parts of 1₿ each, allowing the participants to withdraw their deposits back.

The language is defined exploiting the Racket macro system, which is used to rewrite BitML syntactic constructs to Racket code. This approach benefits from the Racket language ecosystem, and allows us to write BitML contracts in the DrRacket IDE. Indeed, our toolchain integrates within the DrRacket IDE the contract editor, the security analyzer and the BitML compiler. The implementation of BitML in Racket extends the idealized version of BitML in [13] to make the language usable in practice. For instance, it introduces special deposits of type `fee`, which are automatically spread over all the transactions obtained by the compiler. We also implement static checks for a number of errors that could prevent the correct execution of contracts, e.g. committing secrets with the same hash, double spending a transaction output, etc.

3 VERIFYING BITML CONTRACTS

The tool verifies various forms of *liquidity*, requiring that no funds (or funds up-to a certain amount) are frozen forever within a contract[3]. Further, the tool can verify arbitrary LTL formulae, where state predicates can specify, e.g., the funds owned by participants, the provided authorizations, and the revealed secrets. By default, the tool verifies the required property against *all* possible behaviours of each participant: for instance, if a contract contains `reveal a`, the verifier considers both the case where the secret is revealed and the one where it is not. Authorizations are handled similarly, by considering both cases. However, in most cases, a participant wishes to verify a contract with respect to a given behaviour for herself, making no assumptions on the other participants' behaviour (unless some other participants are considered trusted, in which case it would make sense to fix a behaviour also for them). For instance, a participant A may want to give her authorization to perform a given branch only after participant B has revealed his secret. The tool allows for constraining the behaviour of participants, specifying

[3] A paradigmatic case of non-liquid contract was the Ethereum Parity Wallet [1]. An attacker managed to kill a library called by the wallet, irreversibly freezing ~160$M$$.

the conditions upon which secrets are revealed and authorizations are provided. Actions which can be performed by everyone, like `withdraw` and `split`, cannot be constrained.

For instance, we can verify that the mutual timed commitment contract is liquid whatever strategies are chosen by participants. The query `check-liquid` correctly answers true, since: (i) if `A` does not reveal, then anyone (after the block at height 100000) can perform `withdraw "B"`, which transfers the whole contract balance to `B`; (ii) if `A` reveals but `B` does not reveal, then anyone (after the block at height 100050) can perform `withdraw "A"`, which transfers the whole contract balance to `A`; (iii) if both `A` and `B` reveal, then anyone can perform `split`, which transfers the balance in equal parts to `A` and `B`.

Note that if we remove the `after` branch at line 16, the contract is no longer liquid. However, it becomes liquid when `A`'s strategy is to reveal the secret. We can verify that this holds through the query `check-liquid (strategy "A" (do-reveal a))`. Liquidity is lost again if `A` chooses to reveal only after `B` has revealed, i.e. when her strategy is `"A" (do-reveal a) if ("B" (do-reveal b))`.

Besides liquidity, we can check specific LTL properties of contracts through the command `check-query`. E.g., in the mutual timed commitment we can verify that, after `A` reveals, she will eventually get back at least her 1₿ deposit. In LTL, this property is formalised as the following formula, where 10^8 satoshi = 1₿:

```
[](a revealed => <>A has-deposit>= 100000000 satoshi)
```

We also verify that if `A` reveals the secret, then eventually either `B` reveals, or `A` will get `B`'s deposit, too. The LTL query is the following:

```
[](a revealed =>
<>(b revealed \/ A has-deposit>= 200000000 satoshi))
```

Our verification technique is based on model-checking the state space of BitML contracts. Since this state space is infinite, before running the model-checker we reduce it to a finite-state one, by exploiting the abstraction in [14]. This abstraction resolves the three sources of infiniteness of the concrete semantics of BitML: the passing of time, the advertisement/stipulation of contracts, and the off-contract bitcoin transfers. To obtain a finite-state system, the abstraction: (i) quotients time in a finite number of time intervals, (ii) disables the advertisement of new contracts, and (iii) limits the off-contract operations to those for transferring funds to contracts and for destroying them. This abstraction is shown in [14] to enjoy a strict correspondence with the concrete BitML semantics: namely, each concrete step of the contract under analysis is mimicked by an abstract step, and vice versa.

Our tool implements the abstract BitML semantics in Maude, a model-checking framework based on rewriting logic [18]. Maude is particularly convenient for this purpose: we use its equational logic to express structural equivalence between BitML terms, and its conditional rewriting rules to encode the abstract semantics of BitML. In this way, we naturally obtain an executable abstract semantics of BitML. Once a BitML contract in translated in Maude, we use the Maude LTL model-checker [21] to verify the required security properties, under the strategies specified by the user. The various forms of liquidity are also translated to corresponding LTL formulae. The computational soundness of the BitML compiler guarantees that the properties verified by the model checker are preserved when executing the contract on Bitcoin.

4 COMPILING BITML TO BITCOIN

Our compiler operates in two phases: first, it translates BitML contracts into Balzac[4], an abstraction layer over Bitcoin transactions based on the formal model of [9]; then, it translates Balzac transactions into standard Bitcoin transactions. The compiler from BitML to Balzac implements the algorithm in [13], extending it with transaction fees. In particular, the compiler guarantees that each transaction contains enough fees to be publishable in the blockchain. The compiler from Balzac to Bitcoin produces *standard* Bitcoin transactions [3]: this is crucial since non-standard ones are discarded by the Bitcoin network. To this aim, Balzac produces standard output scripts of the form "Pay to Public Key Hash" (P2PKH) or "Pay to Script Hash" (P2SH). P2PKH is used for encoding signature verification (e.g., to redeem the deposit obtained by a `withdraw`), while P2SH is used for complex redeeming conditions (e.g., to check that the revealed secret matches the committed hash). Since Bitcoin requires that all the values pushed by standard scripts fit within 520 bytes, our compiler checks that this constraint is satisfied for each generated script. Balzac outputs serialized raw transactions, which can be directly broadcast to the Bitcoin network.

5 EVALUATION

To evaluate our toolchain, we use a benchmark of representative use cases, including financial contracts [17, 38], auctions, lotteries [7, 33] and gambling games[5]. For each contract in the benchmark, we display in Table 1 the number N of involved participants, the number T of transactions obtained by the compiler, and the verification time V for checking liquidity[6]. The participants' strategies are constrained only as needed to ensure liquidity: in most cases, we do not put any constraints at all. For the contracts which involve predicates on secrets (e.g., all the lotteries), in principle one would need to check liquidity against all the possible choices of secrets. To make verification feasible, since each contract only checks a finite set of predicates, we partition the infinite choices of secrets into a finite set of regions, and sample one choice from each region. In this way, the liquidity check is performed a finite number of times, ensuring that the verifier explores every reachable state of the contract. For instance, in the 4-players lottery we explore 3^4 regions, which explains the 67 hours needed to verify its liquidity.[7]

The only work against which we can compare the performance of our tool is [5], which models Bitcoin contracts in Uppaal, a model-checking framework based on Timed Automata. The most complex contract modelled in [5] is the mutual timed commitment with 2 participants: this requires $\sim$ 30s to be verified in Uppaal, while our tool verifies the same property in $<$ 100ms. This speedup is due to the higher abstraction level of BitML over [5], which operates at the (lower) level of Bitcoin transactions.

One of the main difficulties that we have encountered in developing contracts is that some complex BitML specifications can not be compiled to Bitcoin, because Bitcoin has a 520-byte limit on the size of each value pushed to the evaluation stack [2]. In some cases, we

[4]https://github.com/balzac-lang/balzac
[5]https://github.com/bitml-lang/bitml-compiler/tree/master/examples/benchmarks
[6]For uniformity, in the performance evaluation we focus on liquidity We carry out our experiments on a PC with a Intel Core i7-7800X CPU @ 3.50GHz, and 64GB of RAM.
[7]Another feature which significantly affects the verification time is the fact that we are considering *all* the possible strategies of *all* the participants.

Contract	***N***	***T***	***V***
Mutual timed commitment	2	15	83ms
Mutual timed commitment	3	34	103ms
Mutual timed commitment	4	75	454ms
Mutual timed commitment	5	164	13s
Escrow (early fees)	3	12	8s
Escrow (late fees)	3	11	3.4s
Zero Coupon Bond	3	8	86ms
Coupon Bond	3	18	1.3s
Future(C)	3	$5 + T_C$	80ms + V_C
Option(C, D)	3	$14 + T_C + T_D$	90ms + $V_C + V_D$
Lottery ($O(N^2)$ collateral)	2	15	427ms
Lottery (0 collateral)	2	8	142ms
Lottery (0 collateral)	4	587	67h
Rock-Paper-Scissors	2	23	781ms
Morra game	2	40	674ms
Shell game	2	23	27s
Auction (2 turns)	2	42	3.3s

Table 1: Benchmarks for the BitML toolchain.

managed to massage the BitML contract so to make its compilation respect the 520-byte constraint. For instance, a common pattern that easily violates the 520-byte constraint is the following:

```
(choice (revealif (b) (pred (p0)) (C0))
        (revealif (b) (pred (p1)) (C1))
        (after T      (C2)))
```

The `choice` is compiled into a transaction whose redeem script encodes the disjunction of *three* logical conditions, corresponding to the three branches of the `choice`. Depending on the predicates `p0` and `p1`, and on the number of participants in the contract, this script may violate the 520-byte constraint. A workaround is to rewrite the pattern above into the following one:

```
(choice (revealif (b) (pred (p0)) (C0))
        (after T (tau (choice
                       (revealif (b) (pred (p1)) (C1))
                       (after T1      (C2))))))
```

In this case the compilation includes two transactions, corresponding to the two `choice`s. The scripts of these transactions encode the disjunction of *two* logical conditions, corresponding to the two branches of the `choice`s. Using this workaround we have managed to compile the 4-players lottery into standard transactions, at the price of increasing the number of transactions (587 for the standard version *vs.* 138 for the nonstandard one). Similar techniques (e.g. simplification of predicates) allowed us to compile all the contracts in Table 1 into standard Bitcoin transactions.

In general, the 520-byte constraint intrinsically limits the expressiveness of Bitcoin contracts: for instance, since public keys are 33 bytes long, a contract which needs to simultaneously verify 15 signatures can not be implemented using standard transactions.

6 CONCLUSIONS

Although our benchmarks witness a rich variety of contracts expressible in BitML, there is room for improvement. BitML is not *Bitcoin-complete*, i.e. some contracts executable in Bitcoin are not expressible in BitML. The main sources of this incompleteness are three: (i) all the transactions obtained by the compiler must be signed *before* stipulation by all the involved participants (only the signatures for authorizations can be provided at run-time); (ii) all transaction fields must be taken into account when computing signatures, while partial signatures (e.g. those obtained through `SIGHASH_ANYONECANPAY` and `SIGHASH_SINGLE`) are not used; (iii) off-chain interactions are limited to revealing secrets and providing authorizations. The first constraint is required to ensure that honest participants can always perform, at the Bitcoin level, the moves enabled in the corresponding BitML contract, regardless of the behaviour of the others. In this respect, BitML follows the standard assumption that participants are *non-cooperative*, i.e. at any moment after stipulation they can stop interacting (unlike TypeCoin [19], which assumes cooperation, allowing dishonest participants to make a contract deadlock). Yet, cooperation can be incentivized, by punishing misbehaviour with penalties, like e.g. in the timed commitment of Section 2. As a consequence of the design choices above, contracts with a dynamically-defined set of players (e.g., crowdfunding), or an unbounded number of iterations (e.g., micro-payment channels), are not expressible in BitML.

The limitations of BitML (and of Bitcoin) could be overcome in various ways. For instance, using Bitcoin "as-is", it would be possible to relax constraint (iii) above, so to allow e.g. zero-knowledge off-chain protocols. This would enable to extend BitML with primitives to express *contingent payments* contracts, where participants trade solutions of a class of NP problems [10, 32]. Similarly, by relaxing constraint (i), we could extend BitML to enable dynamic stipulation of subcontracts, requiring that all the involved participants provide their signatures at run-time. This would allow to model e.g. micro-payment channels in BitML. Together with the use of `SIGHASH_ANYONECANPAY` (relaxing constraint (ii)), this would also allow for modelling crowdfunding contracts. As before, this extension could be implemented without modifying Bitcoin.

Other extensions of BitML would require extensions of Bitcoin. For instance, *covenants* [34, 36] would allow for implementing arbitrary finite-state machines. Controlled *input malleability* would allow to efficiently implement tournaments in multi-player gambling games, like e.g. lotteries [12]. This can also be achieved through a new opcode that checks if the redeeming transaction belongs to a given set [33]. Contingent payments without zero-knowledge proofs can be achieved by exploiting a new opcode that checks the validity of key pairs [20]. A new opcode which checks signatures for arbitrary messages would allow for expressing general fair multiparty computations [29]. Further, fair and robust multiparty computations can be achieved using more complex transactions [26]. A more radical approach would be to replace the Bitcoin scripting language with a more expressive one, like e.g. Simplicity [35].

Compared with the tools for analysing Ethereum contracts [16, 23–25, 31, 37, 39, 40], whose precision is subject to the limitations derived by the Turing-completeness of the underlying languages, our toolchain features a sound and complete verification technique.

ACKNOWLEDGMENTS

Work partially supported by MIUR PON 2018 "Distributed Ledgers for Secure Open Communities" ARS01_00587; by R.A.S. projects "Sardcoin" and "Smart collaborative engineering"; by EPSRC projects EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1 and EP/N028201. Stefano Lande is supported by P.O.R. F.S.E. 2014-2020.

REFERENCES

[1] 2017. A Postmortem on the Parity Multi-Sig Library Self-Destruct. https://goo.gl/Kw3gXi.
[2] 2019. Bitcoin script size limit. https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki#520-byte-limitation-on-serialized-script-size.
[3] 2019. Bitcoin standard transactions. https://bitcoin.org/en/developer-guide#standard-transactions.
[4] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. 2014. Fair Two-Party Computations via Bitcoin Deposits. In *Financial Cryptography Workshops (LNCS)*, Vol. 8438. Springer, 105–121. https://doi.org/10.1007/978-3-662-44774-1_8
[5] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Łukasz Mazurek. 2014. Modeling Bitcoin contracts by timed automata. In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS) (LNCS)*, Vol. 8711. Springer, 7–22. https://doi.org/10.1007/978-3-319-10512-3_2
[6] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. 2014. Secure Multiparty Computations on Bitcoin. In *IEEE S & P*. 443–458. https://doi.org/10.1109/SP.2014.35 First appeared on Cryptology ePrint Archive, http://eprint.iacr.org/2013/784.
[7] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. 2016. Secure multiparty computations on Bitcoin. *Commun. ACM* 59, 4 (2016), 76–84. https://doi.org/10.1145/2896386
[8] Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli, Stefano Lande, and Roberto Zunino. 2018. SoK: unraveling Bitcoin smart contracts. In *POST (LNCS)*, Vol. 10804. Springer, 217–242. https://doi.org/10.1007/978-3-319-89722-6
[9] Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. 2018. A formal model of Bitcoin transactions. In *Financial Cryptography and Data Security (LNCS)*, Vol. 10957. Springer. https://doi.org/10.1007/978-3-662-58387-6
[10] Waclaw Banasik, Stefan Dziembowski, and Daniel Malinowski. 2016. Efficient Zero-Knowledge Contingent Payments in Cryptocurrencies Without Scripts. In *ESORICS (LNCS)*, Vol. 9879. Springer, 261–280. https://doi.org/10.1007/978-3-319-45741-3_14
[11] Massimo Bartoletti, Tiziana Cimoli, and Roberto Zunino. 2018. Fun with Bitcoin Smart Contracts. In *ISoLA (LNCS)*, Vol. 11247. Springer, 432–449. https://doi.org/10.1007/978-3-030-03427-6_32
[12] Massimo Bartoletti and Roberto Zunino. 2017. Constant-deposit multiparty lotteries on Bitcoin. In *Financial Cryptography Workshops (LNCS)*, Vol. 10323. Springer. https://doi.org/10.1007/978-3-319-70278-0
[13] Massimo Bartoletti and Roberto Zunino. 2018. BitML: a calculus for Bitcoin smart contracts. In *ACM CCS*. https://doi.org/10.1145/3243734.3243795
[14] Massimo Bartoletti and Roberto Zunino. 2019. Verifying liquidity of Bitcoin contracts. In *POST (LNCS)*, Vol. 11426. Springer, 222–247. https://doi.org/10.1007/978-3-030-17138-4_10
[15] Iddo Bentov and Ranjit Kumaresan. 2014. How to Use Bitcoin to Design Fair Protocols. In *CRYPTO (LNCS)*, Vol. 8617. Springer, 421–439. https://doi.org/10.1007/978-3-662-44381-1_24
[16] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. 2016. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS '16)*. ACM, 91–96. https://doi.org/10.1145/2993600.2993611
[17] Alex Biryukov, Dmitry Khovratovich, and Sergei Tikhomirov. 2017. Findel: Secure Derivative Contracts for Ethereum. In *Financial Cryptography Workshops (LNCS)*, Vol. 10323. Springer, 453–467. https://doi.org/10.1007/978-3-319-70278-0_28
[18] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. 2002. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.* 285, 2 (2002), 187–243. https://doi.org/10.1016/S0304-3975(01)00359-0
[19] Karl Crary and Michael J. Sullivan. 2015. Peer-to-peer affine commitment using Bitcoin. In *ACM Conf. on Programming Language Design and Implementation*. 479–488. https://doi.org/10.1145/2737924.2737997
[20] Sergi Delgado-Segura, Cristina Pérez-Solà, Guillermo Navarro-Arribas, and Jordi Herrera-Joancomartí. 2017. A fair protocol for data trading based on Bitcoin transactions. *Future Generation Computer Systems* (2017). https://doi.org/10.1016/j.future.2017.08.021
[21] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. 2002. The Maude LTL Model Checker. *Electr. Notes Theor. Comput. Sci.* 71 (2002), 162–187. https://doi.org/10.1016/S1571-0661(05)82534-4
[22] Matthew Flatt. 2012. Creating languages in Racket. *Commun. ACM* 55, 1 (2012), 48–56. https://doi.org/10.1145/2063176.2063195
[23] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. Foundations and Tools for the Static Analysis of Ethereum Smart Contracts. In *CAV (LNCS)*, Vol. 10981. Springer, 51–78. https://doi.org/10.1007/978-3-319-96145-3_4
[24] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *POST (LNCS)*, Vol. 10804. Springer, 243–269. https://doi.org/10.1007/978-3-319-89722-6_10
[25] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon M. Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Rosu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *IEEE Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, 204–217. https://doi.org/10.1109/CSF.2018.00022
[26] Aggelos Kiayias, Hong-Sheng Zhou, and Vassilis Zikas. 2016. Fair and Robust Multi-party Computation Using a Global Transaction Ledger. In *EUROCRYPT (LNCS)*, Vol. 9666. Springer, 705–734. https://doi.org/10.1007/978-3-662-49896-5_25
[27] Ranjit Kumaresan and Iddo Bentov. 2014. How to Use Bitcoin to Incentivize Correct Computations. In *ACM CCS*. 30–41. https://doi.org/10.1145/2660267.2660380
[28] Ranjit Kumaresan and Iddo Bentov. 2016. Amortizing Secure Computation with Penalties. In *ACM CCS*. 418–429. https://doi.org/10.1145/2976749.2978424
[29] Ranjit Kumaresan, Tal Moran, and Iddo Bentov. 2015. How to Use Bitcoin to Play Decentralized Poker. In *ACM CCS*. 195–206. https://doi.org/10.1145/2810103.2813712
[30] Ranjit Kumaresan, Vinod Vaikuntanathan, and Prashant Nalini Vasudevan. 2016. Improvements to Secure Computation with Penalties. In *ACM CCS*. 406–417. https://doi.org/10.1145/2976749.2978421
[31] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 254–269. https://doi.org/10.1145/2976749.2978309
[32] Gregory Maxwell. 2016. The first successful Zero-Knowledge Contingent Payment. (2016). https://bitcoincore.org/en/2016/02/26/zero-knowledge-contingent-payments-announcement/.
[33] Andrew Miller and Iddo Bentov. 2017. Zero-Collateral Lotteries in Bitcoin and Ethereum. In *EuroS&P Workshops*. 4–13. https://doi.org/10.1109/EuroSPW.2017.44
[34] Malte Möser, Ittay Eyal, and Emin Gün Sirer. 2016. Bitcoin covenants. In *Financial Cryptography Workshops (LNCS)*, Vol. 9604. Springer, 126–141. https://doi.org/10.1007/978-3-662-53357-4_9
[35] Russell O'Connor. 2017. Simplicity: A New Language for Blockchains. In *PLAS*. https://doi.org/10.1145/3139337.3139340
[36] Russell O'Connor and Marta Piekarska. 2017. Enhancing Bitcoin transactions with covenants. In *Financial Cryptography Workshops (LNCS)*, Vol. 10323. Springer. https://doi.org/10.1007/978-3-319-70278-0_12
[37] Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Rosu. 2018. A formal verification tool for Ethereum VM bytecode. In *ACM ESEC/SIGSOFT FSE*. 912–915. https://doi.org/10.1145/3236024.3264591
[38] Pablo Lamela Seijas and Simon J. Thompson. 2018. Marlowe: Financial Contracts on Blockchain. In *ISoLA (LNCS)*, Vol. 11247. Springer, 356–375. https://doi.org/10.1007/978-3-030-03427-6_27
[39] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018. Scilla: a Smart Contract Intermediate-Level LAnguage. *CoRR* abs/1801.00687 (2018).
[40] Petar Tsankov, Andrei Marian Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. *CoRR* abs/1806.01143 (2018).