Compact Multiparty Verification of Simple Computations

Omri Ross[1] and Johannes Rude Jensen[2]

[1] University of Copenhagen
[2] Copenhagen Business School
Omri@firmo.network; Johannes@firmo.network

Abstract.
We present a compact model for blind multiparty verification of compilation results. By employing a simple incentive scheme, we construct a mechanism, staking a deposit value on the correctness of compiled and deployed byte code. A blind committee of peers evaluate the authenticity of the deployed byte code by re-computing the task, hashing the source and target code into checksums, and submitting bids to the contract. If the evaluation round reveals inconsistencies in the checksums provided by the peers, the contract can be rejected and the deposit shared amongst contenders.

Keywords: Domain Specific Languages, Computational Verification, Blockchain Technology.

1 Introduction

With the proliferation of blockchain and peer-to-peer technologies [2, 5], an increasing emphasis has been placed on verification of both programming languages and compilers. This is manifested in a growing demand for formally verifiable solutions, strictly typed or purely functional programming languages. [6,9,11,12,20,24,25,37] This development is, arguably, for good reason. The Ethereum blockchain, a smart contract platform currently supporting a plethora of digital assets, has been subject to multiple episodes resulting in the loss of funds. [33,40,41] The Ethereum blockchain executes smart-contracts in the Ethereum Virtual Machine, (EVM) a replicated stack machine maintained by nodes in the network. Programmers predominately write smart contracts in the higher level scripting language: Solidity, a JavaScript inspired contract language compiling directly to EVM. [46] Due to the complexity of Solidity, episodes involving coordinated or accidental loss of funds are often connected to delicate aspects of the EVM semantics. [6, 33,40,41]
Atzei et al. [1] defines a taxonomy of exploits and vulnerabilities observable in smart contracts, written in Solidity. Luu et al. allocates a set of known vulnerabilities in 8,883 of 19,366 smart contracts deployed on the Ethereum blockchain, utilizing the open source symbolic execution tool, Oyente [28,32]. As is commonly known, the

immutable characteristics of the deterministic environment afforded by blockchain technologies, does not permit debugging. Even minor errors, exploits or vulnerabilities can result in severe losses of funds, thus the adage: "*Code is law"*. At the surface, the continuous stream of issues in smart contract technologies, may seem puzzling. After all, smart contracts are simple but high risk computations. So, why is it so difficult to produce correct code? Amongst many plausible causes, a common culprit is the complexity of native scripting languages available today. The DAO hack [40,41] and the Parity multi-sig Wallet incident [33], are well-documented cases, both resulting in significant loss of value. Both cases involved fairly simple computations, expressed in complex programming languages. Due to the openness of the Ethereum blockchain, any user can call the public methods defined in a contract. [6] Writing and verifying contracts shielded from any potential abuse, has proven to be a difficult task. Fortunately, low fault tolerance in code execution, is not a problem isolated to the domain of blockchain technology. The issue is commonly known in a variety of fields, concerned with safety and mission critical software development. Consider the notoriously prohibitive safety requirements in the aerospace engineering industry as an example. [47] In mission critical software development, the use of domain specific languages (DSL) is commonly known as good practice. While open-source projects and exhaustive literature on domain specific languages is widely available, [12,15,20,24,25,26] the broad application of DSL in financial peer-to-peer systems remains absent. One can think of multiple reasons for the lack of language plurality in the blockchain space. A dominant cause, we submit, is the bottleneck occurring in the compilation and deployment of domain specific languages. When a community tasked with the development maintenance of a programming language is small, [43] verifying the integrity of the compilation process, can prove a cumbersome process. Though language plurality may serve the growing community of users well, the long term implication of the current situation, could be the concentration of open-source participation in just a few select programming languages. Developing a domain specific language requires ressoruces and talent, bearing little probability of any revenue. We perceive this partially, as a problem of trust. How can the end-user verify that her counterparty has compiled the correct contract, to the blockchain? Unless a verifiably correct implementation of a compiler is available online, the user will be required to read and verify assembly code by hand. By proposing a compact solution for the verification of correct compilation in a multitude of programming languages, we aim to encourage language pluralism in blockchain and distributed ledger technologies. Verifying the authenticity of a compiled and deployed contract by imposing simple games for rational agents may, in turn, create a platform of trust for communities or individuals involved in developing domain specific languages. The monetization of such a platform may contribute much needed resources, towards the maintenance and development of open-source niche languages for a variety of use-cases.

1.1 Honest Deployment

We introduce the problem of honest deployment with the following parable: Alice and Bob are non-technical end-users or financial service providers, looking to deploy a series of specific financial contracts to the blockchain. Eve has created a strictly typed, formally verified, domain specific language, perfectly suitable for writing these contracts. Eve is able to write compile and securely deploy any contract specified by Alice and Bob. Now, how can Alice and Bob verify that Eve is compiling and deploying the correct contract?

Similar challenges have mobilized attention, in times where out-sourcing large computational tasks has become a viable business model for corporations with large server capacities. Suppliers and consumers of *infrastructure-as-a-service* (IaaS) share issues, well aligned with the problem addressed here. [35,39] In most cases, consumers do not have the capacity, know-how or time to re-execute computations. On the other end of the trade, service providers lack the means to demonstrate the validity of their services to their clients or end-users. [44] The question is thus; how can consumers verify computations, without having to re-execute them locally? As suggested by Walfish and Blumberg [45] we can formulate the problem as the following: The *client* delivering an input denoted by x, provides the *worker* computing the task with a specification, denoted by p. The worker computes an output y, now having to provide the client with succinct proof that $y = p(x)$. A few obvious restrictions apply to a solution for problems of this nature. First, verifying the output y has to be less computationally costly than computing $p(x)$, if the model is to remain relevant. Second, p has to result in a deterministic output, unless very sophisticated and costly verification methods are available. Third, the worker must deliver tangible evidence that malicious behavior would, with high probability, be more expensive than the potential gains of attempting an attack.

1.2 Existing Solutions

We consider currently existing solutions to similar problems in computational verification and associated theory. Due to the generally applicable nature of problems in the verification of computations, the literature on the subject is rich. Theoretical and practical implementation predominantly subscribe to Micali's notion of a *Computationally Sound* proof (CS proof). Amongst the required properties of a CS proof, Micali lists convenience, feasibility, reasonable complexity, universality, transferability and confidence in the soundness of the proof. [31] Research on the practical implementation of probabilistic proof schemes has made great advancements in meeting these requirements. Most recently, practical implementations of proofs in quantum computing has generated new interest in the field. [14] Yet, the general range of practical implementations often prioritize highly specific applications for single classes of computations. Existing implementations are typically composed of hybrid abstractions of *interactive proofs* and *probabilistically checkable proofs* [7,17,18,19,30,35,36,38,39]. Interactive proofs rely on the exchange of a series of messages, determining the authenticity of a proof certificate. PCP proofs generally proceed by checking random selection of bits in a correctly formatted proof. Here, the prover provides the verifier with an encoded transcript of the operation, in the attempt at convincing the verifier that the computation has been correctly executed.[1] A common denominator in these and similar operations, is the overhead and setup costs associated with the required computations. [44,45] While an extensive cost-profile is permissible in certain scenarios, the aforementioned use-case does not tolerate extraneous costs. As is evident, a cost profile for Alice and Bob, relies more on the probability of Eve acting benevolently, than on the costs associated with a series of trivial compilations and the deployments of contract code. Here, we note the importance of externalities in the verification of computations. The extant literature predominantly approaches verification as a bilateral process between two publically known agents: *prover* and *verifier*. In the conventional verification paradigm, the agent's identity is public knowledge, while communications are kept private. Public blockchain technology reverses this tendency with the introduction of *openness* as a fundamental property. On a public blockchain, communication between agents in the form of transactions or computations are public. To provide a level of anonymity, identities are kept pseudonymous, discernable only by public keys or addresses. [16] Consequently, the environment presented by public blockchain technologies does not permit tacit notions of reputation in the verification process.[2] A malicious agent is perfectly capable of switching both public keys and addresses at any point, discarding any association with a previous address. This property poses a difference to the general literature on outsourced computations, building on a set of implicit assumptions on the location and identity of agents and infrastructure. When utilizing cloud services, we trust that questionable performance will, eventually, impede financial penalties on the service provider. Whether in the form of lost revenue or contractual penalties, we confide in the public association between brand and output. A practical verification scheme must replicate the relationship between performance and penalty, in the presence of strategic agents with pseudonymous identities.

Teutsch et al. and Reitweißner [42,29,22] addresses this problem in detail. With the TrueBit system, users can submit a computation to the network of *Solvers* and *Challengers.* Extending the aforementioned terminology, Solvers compute *p(x)* and pose a simple proof that $y = p(x)$. Challengers may dispute the solution by computing the same task with a conflicting result. When a challenger disputes a solvers results, a subroutine, known as the 'verification game' is initiated [42]. A verification game is, essentially, a search problem, in which Solver and Challenger proceed by configuring the disputed computation in ranges. Each range is represented by a Merkle tree, containing the entire state. By indexing the Merkle root for each range on the blockchain, the Challenger can allocate the discrepancy between the two computations and pass this information on to the Solver. This loop is repeated in several iterations, until the range in which Solver and Challenger computes conflicting results is well defined. At this point, the disputed range is executed on-chain. The dispute is resolved by distributing a staked deposit value to the winning party. The TrueBit system offers a novel

[1] We recommend Walfish et al. (2015) rich survey of exciting practical implementations for an overview of the field.

[2] We address this issue further, in the discussion.

contribution to the extant literature, by introducing of financial incentives to the verification of computations. As both Solvers and Challengers are prompted to deposit a predefined value in the TrueBit contract, agents can be penalized and rewarded accordingly. Nevertheless, this implementation of the TrueBit system is not ideal for simple or trivial computations involving high-risk, such as the compilation of smart contracts written in domain specific languages.

2 A Verification Scheme for Simple Computations

In extension of the model presented and discussed by Teutsch and Reitweißner et al. [42] we present a verification scheme in the form of an extended game, with perfect information about other player's previous moves, but not their types. [23] As noted above, the verification of third party computations is a rich and complex issue, addressed at length in the literature. Following the work presented by Teutsch et al. [22 ,42] this implementation deviates from the general methodologies derived from IP or PCP based proof systems. Departing from these technically dense methodologies, we provide a simple incentive scheme for the peer verification of computational results. The mechanism is an indirect implementation with sequential messages, revealing information about other player's intention. The group of agents in each 'game' is a tuple of *n* participants, for this limited example N ={A,B,C}. Players have type $\Theta \equiv \{\theta^1, \ldots, \theta^2\}$ as elaborated below types are either benevolent or malicious/naive. In this model, types are only defined by weather or not the agent is able to verifiably compute *y = p(x).* For reasons of simplicity, this implementation only discriminates between two types of agents. We denote type θ_1 as the benevolent agent, correctly computing *y = p(x).* Any other agent is treated as a malicious, naive or otherwise compromised, denoted by type θ_2.

When Eve executes a compilation and deploys the contract code for Alice and Bob, she is required to *deposit* assets corresponding to a ratio of the contractual value-at-risk. The deposit is locked on-chain with the fees gathered by Alice and Bob. The source code and the compiled byte code is hashed into checksums and logged on chain. The source code is then distributed amongst seven randomly selected peers. Before Alice and Bob signs or executes the deployed contract, the code deployed by Eve is subject to a blind peer review. Five out of seven peers challenge the results by submitting their checksums. If Eve is proven to have computed the contract correctly, the contract is signed and released by Alice and Bob. This action transfers the deposit (*d*) and the fee (*f*) back to Eve. If the computation is disputed, the deposit is shared by consenting contenders. The following process details the implementation proposed in this paper. We advise the reader follow along, by viewing the state chart in appendix A. Appendix B provides a visualization of step one through four providing additional visual aid.

Step 1: Alice A and Bob B, presents a contract *x* and a set of specifications *p* to Eve C.

Step 1a: The required deposit *d* is calculated from the estimated value-at-risk, denoted v_u. The value at risk is multiplied by the deposit to contract value ratio *r*, expressed as:

$$d = (Av_u + Bv_u) * r$$

Step 1b: The service-fee *f* demanded by C, is stored with the deposit d on chain, for the remainder of the process.

Step 2: C presents a compilation result *y* to A and B as a transitional proof of $y = p(x)$.

Step 3: A, B, C independently produce hashed checksums of the source-code y_c and the output: x_c

Step 3a: If $(A, B(y_c, x_c)) \neq C(y_c, x_c)$ the contract is rejected.

Step 3b: If $A(y_c, x_c) \wedge B(y_c, x_c) = C(y_c, x_c)$ the contract is accepted, and C can deploy *y*

Step 3c: With deployment, A, B, C encode the hashed check sums (y_c, x_c) as input transaction data, together with the compiler pragma (version), denoted by *z.* Alternatively, simple functions such as *uintstoredData* on the Ethereum network can be applied. The only requirements for logging data on-chain is that the hashed checksums and the compiler pragma can be easily associated with the contract under scrutiny.

Step 4: Once the contract is deployed and the checksum and the compiler pragma (y_c, x_c, z) is logged on the blockchain, C must prove the correctness of *y* before A, B signs or executes the deployed bytecode, *y*.

Step 5: A, B, C distributes *x* to 7 randomly selected contenders on the network, denoted by: $T_1 .\ .\ .\ .\ T_7$.

Step 5a: Out of contenders $T_1 .\ .\ .\ .\ T_7$ at least 5 contenders should respond by challenging the validity of $y = p(x)$. A challenge is submitted by computing and hashing $p(x)$ with the correct compiler pragma (version) denoted by *z.* Once a reasonable amount of contenders has challenged the result, the tally can be concluded. If a majority of contesters return the equivalent SHA256 checksums such that:

$$\mathrm{T_n}\ (y_c,\ \ x_c)\ =\ (\mathrm{A,B}(y_c,\ \ x_c))$$

We can assume $\mathcal{C}$ to be *benevolent*, If:

$$\forall\,(\mathrm{T_n}(y_c, x_c))\ =\ (\mathrm{A,B}(y_c,\ \ x_c))\ \ \vdash \mathrm{C}{=}\,\theta_1$$

In this example, if ≧3 contesters return SHA256 checksums such that,

$$(\mathrm{T_n}\ (y_c,\ \ x_c)) \neq (\mathrm{A,B}(y_c,\ \ x_c))$$

We can treat $\mathcal{C}$ as *malicious*, if:

$$\exists \mathrm{T_n}\ (y_c,\ \ x_c) \neq (\mathrm{A,B}(y_c,\ \ x_c))\ \vdash \mathrm{C}{=}\,\theta_2$$

If the randomly selected contenders does not arrive at consensus, the contract can be rejected and *f, d* returned to $\mathcal{A},\mathcal{B},\mathcal{C}$. High value contracts may require full consensus with more challengers. Low value contracts may require only one or two blind peer reviewers.

Step 6a: Having derived the type of $\mathcal{C}$ from the verification game, we can now dispose of the deposited values: *d, f*. If,

$$\mathcal{C} \vdash \mathrm{C}{=}\,\theta_1$$

$\mathcal{A},\ \mathcal{B}$ can sign and release the deployed code *x*. This transfers *d, f* to $\mathcal{C}$ and the process is complete.

Step 6b: If the anonymous voting process showed that:

$$\mathcal{C} \vdash \mathrm{C}{=}\,\theta_2$$

f is returned to $\mathcal{A},\mathcal{B}$ and *d* divided amongst contesters $\mathrm{T_1}$. . . . $\mathrm{T_5}$ in consensus.

Step 7: The division of *d* to winning contenders can be distributed according to any bespoke mechanism, incentivizing the preferred agent behavior. Here, auction theory may provide useful inspiration. One might consider adopting a Dutch-auction model, allocating the potential *d* in a descending order, [50, 25, 12.5, 6.25%] incentivizing agents to compute and send their bid as fast as possible.

3 Discussion

We have shown a potential implementation of an incentive scheme for blind peer verification of simple computational results. The general framework produced in this paper, has been applied to the verification of compiled and deployed byte-code of smart-contracts. This scheme may be ported to other verticals, calling for the verification of mission critical computations. Adding incentive schemes to the verification of computations is applicable in situations where multiple agents with heterogeneous types are facing a) risk of manipulation, b) risk of hardware malfunction, or c) potentially compromised software. Depending on the computational class in question, adding an incentive layer to the validation of a proof may provide additional certainty, that the agent charged with completing the computation will be financially penalized for any misnomer.

3.1 Potential Concerns in the Suggested Implementation

We address challenges in the suggested implementation described above. First, we might question the source of true randomness in the peer selection process in step 5. Provable generation of true randomness is a non-trivial issue, addressed widely in the literature. Recent advancements in the field, alongside an increasing attention paid towards this issue, [8] leads us to the assumption that we will see positive developments in the generation of randomness on public blockchains. Another, similarly pertinent, issue is privacy. This model necessitates the sharing of source-code, which obviously is an undesirable property. The partial encryption and verification of source-code is a potential solution, but has not been pursued in this paper. A different frontier is the habits of strategic agents. If not restrained, agents might attempt to match hashes *before* submitting their bids, in the attempt at saving transaction fees. This can be mitigated either by encoding bids and evaluating after the tally is final, or by tweaking the incentive scheme as suggested in step 7. Along similar lines, we might be concerned with the frequency of malicious/naive behavior. [42] Indeed, how often does malicious behavior need to occur before the model becomes self sustainable? The TrueBit system proposes that system moderators simply inject falsified computations at randomized intervals. This measure incentivizes continuous search for malicious agents. Depending on the implementation, number of compiler pragma (z) and contestants, similar considerations might prove necessary in a practical implementation. A necessary concern is the networks capability of handling new releases. This implementation has shown a static scenario, in which only a single pragma or directive is applicable. An optimal implementation must account for multiple possible pragma, whilst supporting the propagation of software updates throughout the network.

3.1 Potential Optimizations to the Suggested Implementation

We discuss potential optimizations of the model presented in this paper. First, we might reduce on-chain logging by concatenating checksum hashes and compiler pragma into a Merkle tree, indexing only the Merkle root on the blockchain. This design might allow more contenders and faster process times, as checksum matching can be reduced to a simple operation. Additional efficiency gains can be achieved by hashing only the first, and last, *n* bits of the source and target code. This might enhance the models applicability in verification of slightly larger computational tasks.

The quality of an agent's reputation may be quantified utilizing Token Curated Registry schemes (TCR). [27] Agents serving as verifiers and challengers in a certain subset of compiler pragma can be qualified by the number of reputation tokens associated with a designated address. TCR schemes comprise an interesting development in these novel fields, as we may discriminate further amongst the perceived qualifications of agents, without requiring association with identity or location. The scheme proposed here may be instructed to issue a specific number of reputation tokens upon the completion of a successful verification round. Adjacent to the scheme defined above, reputation tokens may be staked with the deposit value (*d*). Such a model would facilitate the quantification of reputation, as trusted agents would be able to stake large amounts for high-risk computations. Consequently, agents in possession of large quantities of reputation tokens, may require higher fee (*f*) for their services.

4 Conclusion

Smart contracts are generally simple computations, often involving financial risks. The emerging ecosystem for financial applications on public blockchain infrastructure, can benefit from a broader variety of domain specific languages. However, there is little incentive for talent to engage in the commercial development of domain specific languages or verified compilers. We attribute this state of affairs, partially to the problem of *honest deployment*. How can a non-technical user verify the integrity of a compilation, without re-executing the computation locally? Departing from the extant literature, predominantly consisting of technically dense and computationaly complex solutions, we develop a basic verification scheme for simple high-risk computations. The scheme proposes a solution to the problem of honest deployment, through the introduction of financial incentives. Agents tasked with the compilation of source code is asked to deposit values corresponding to the value at risk in a given contract. The computation is re-executed by a randomly selected committee of peers. If the computation is shown to be flawed or manipulated, the deposit is shared amongst contenders and the fee reimbursed. If the computation is shown to be correct, the deposit is returned and the round is concluded. The current implementation does not account for reputation systems and software updates. We identify these issues as potential future work on this or related models.

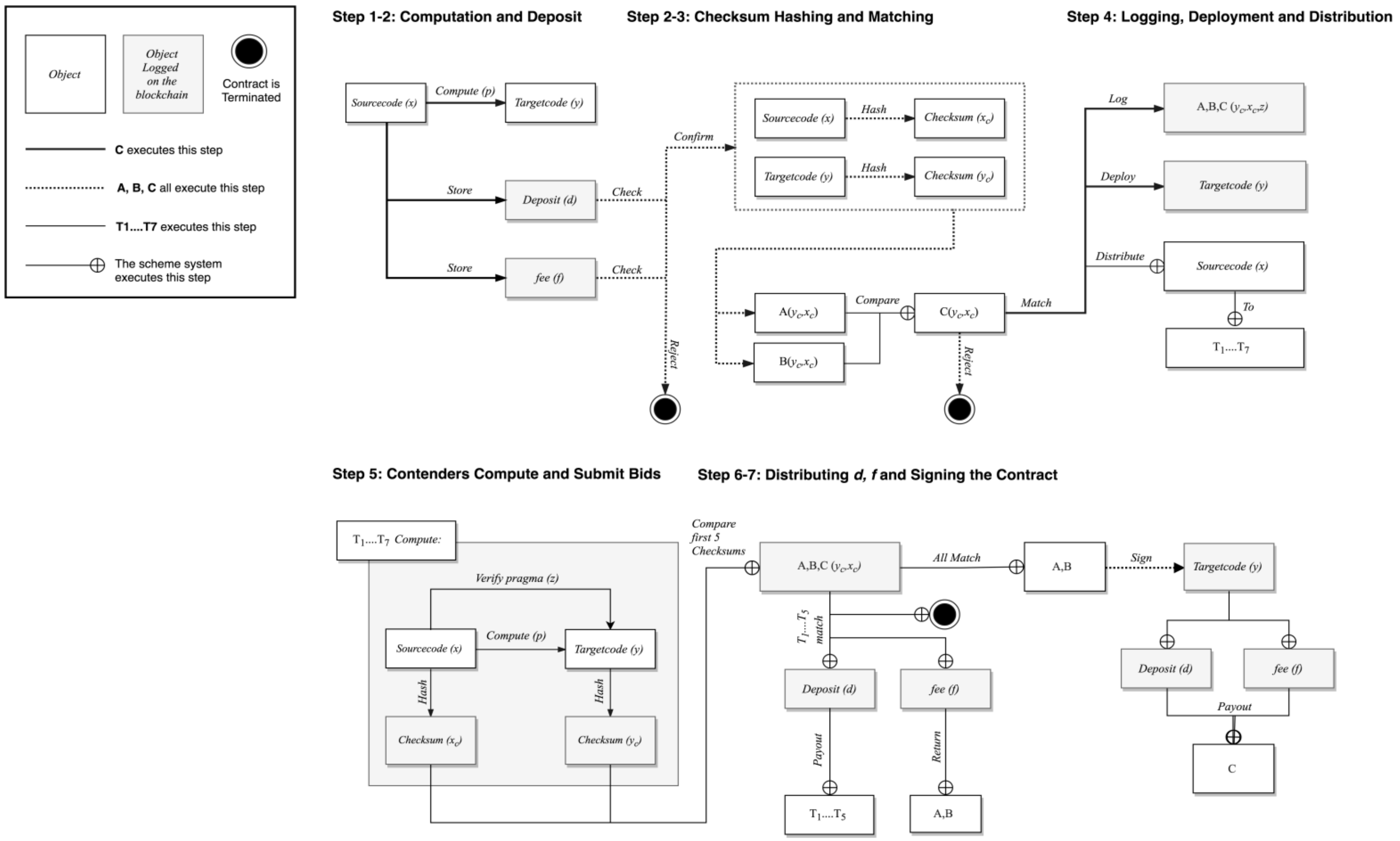
5 References

1. Atzei, N., Bartoletti, M. & Cimoli, T., 2017. A survey of attacks on Ethereum smart contracts (SoK). In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*.
2. Avital, M. et al., 2016. Jumping on the blockchain bandwagon: lessons of the past and outlook to the future. *Proceedings of the 37th International conference on information systems, Dublin*.
3. Barz, S. et al., 2013. Experimental verification of quantum computations. pp.1–13. Available at: http://arxiv.org/abs/1309.0005%0Ahttp://dx.doi.org/10.1038/nphys2763.
4. Bayardo, R.J. & Sorensen, J., 2005. Merkle tree authentication of HTTP responses. In *Special interest tracks and posters of the 14th international conference on World Wide Web - WWW '05*. p. 1182. Available at: http://portal.acm.org/citation.cfm?doid=1062745.1062929.
5. Beck, R., Avital, M. & Rossi, M., 2017. Blockchain Technology in Business and Information Systems. *Business & Information Systems Engineering*.
6. Bhargavan, K. et al., 2016. Formal verification of smart contracts: Short paper. *PLAS 2016 - Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, co-located with CCS 2016*.
7. Braun, B. et al., 2013. Verifying computations with state. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM.
8. Cascudo, I. & David, B., 2017. SCRAPE: Scalable randomness attested by public entities. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*.
9. Chen, X., Park, D. & Rosu, G., 2018. A Language-Independent Approach to Smart Contract Verification. *8th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'18)*.
10. David, B., Ga, P. & Russell, A., 2017. Proof-of-stake Protocol.
11. Delmolino, K. et al., 2015. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. *IACR Cryptology ePrint Archive*, 2015, p.460. Available at: https://eprint.iacr.org/2015/460.pdf.
12. Elsman, M., 2018. Certified Compilation of Financial Contracts contracts which can be turned into a concrete contract.
13. Ethereum Foundation, Solidity, Read the Docs. Available at: https://solidity.readthedocs.io/en/v0.4.24/ [Accessed August 12, 2018].
14. Fitzsimons, J.F. & Hajdusek, M. Post hoc verification of quantum computation.
15. Fowler, M., 2010. *Domain-specific languages*, Pearson Education.
16. Glaser, F., 2017. Pervasive Decentralisation of Digital Infrastructures : A Framework for Blockchain enabled System and Use Case Analysis. In *Proceedings of the 50th Hawaii International Conference on System Sciences | 2017*.
17. Goldreich, O., 2007. Probabilistic Proof Systems: A Primer. *Foundations and Trends® in Theoretical Computer Science*, 3(1), pp.1–91. Available at: http://www.nowpublishers.com/article/Details/TCS-023.
18. Goldwasser, S., Kalai, Y.T. & Rothblum, G.N., 2008. Delegating Computation: Interactive Proofs for Muggles. *40th Annual ACM symposium on Symposium on theory of computing - STOC*, 62(4), pp.113–122. Available at: https://www.microsoft.com/en-us/research/wp-

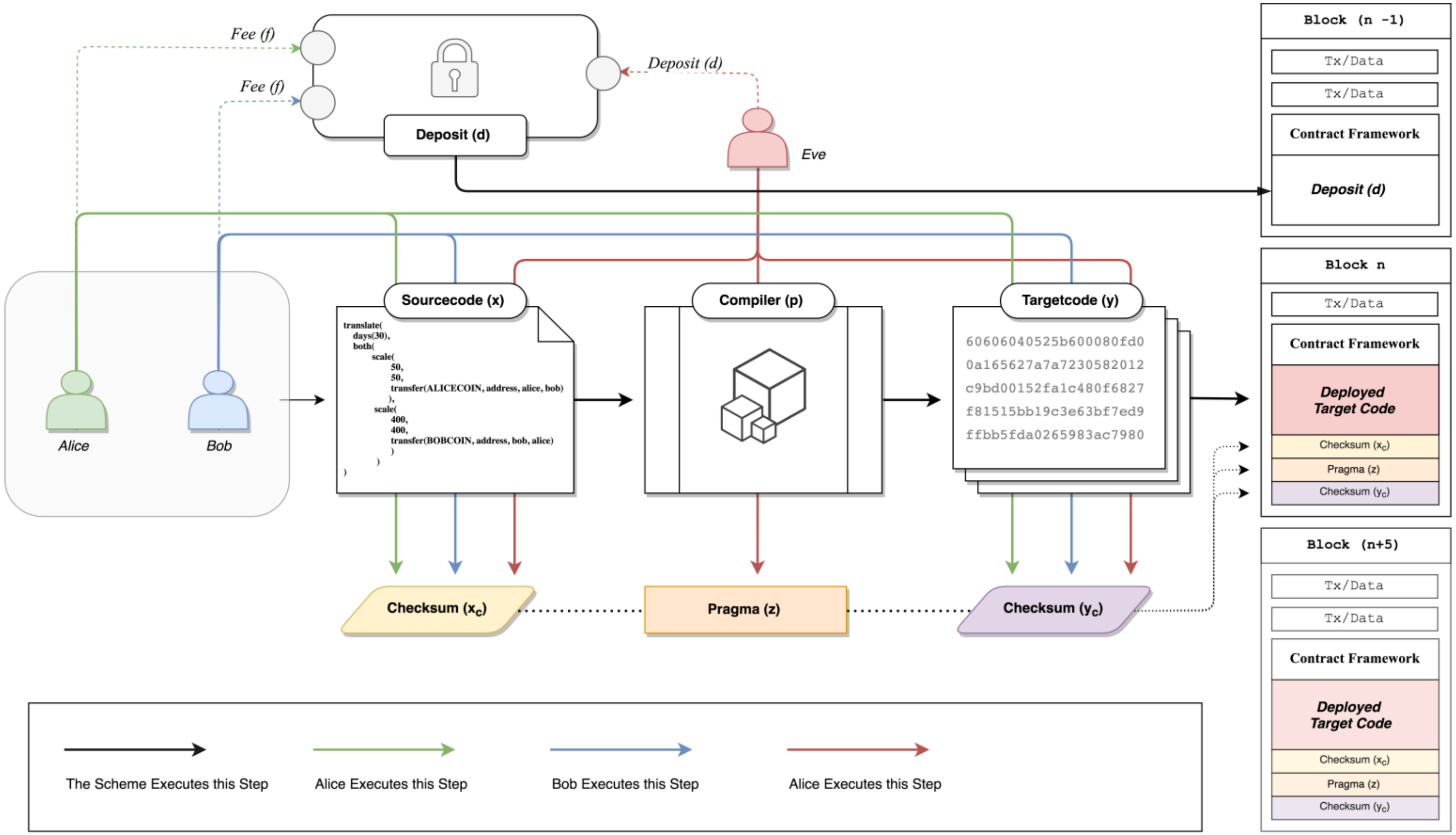
content/uploads/2016/12/2008-DelegatingComputation.pdf%0Ahttp://portal.acm.org/citation.cfm?id=1374396.

19. Goldwasser, S., Micali, S. & Rackoff, C., 1989. The knowledge complexity of interactive proof systems. *SIAM Journal on computing*, 18(1)
20. Henriksen, T. et al., 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*, (i), pp.556–571. Available at: http://dl.acm.org/citation.cfm?doid=3062341.3062354.
21. Homer, S. & Selman, A.L., 2011. Interactive Proof Systems. In *Computability and Complexity Theory*. Springer.
22. Jain, S. et al., 2016. How to verify computation with a rational network. Available at: http://arxiv.org/abs/1606.05917.
23. Kakhbod, A., 2013. *Resource Allocation in Decentralized Systems with Strategic Agents*. The University of Michigan. Available at: http://link.springer.com/10.1007/978-1-4614-6319-1.
24. Kasampalis, T. et al., 2018. IELE: An intermediate-level Blockchain language designed and implemented using formal semantics.
25. Lamela, P., Marlowe: Financial contracts on blockchain., pp.1–20. Available at: https://github.com/input-output-hk/scdsl.
26. Leroy, X., 2013. The CompCert C verified compiler: Documentation and user’s manual. *INRIA Paris-Rocquencourt*, pp.1–10. Available at: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:The+CompCert+C+verified+compiler+Documentation+and+user+?+s+manual#0.
27. Lokyer, M., Token Curated Registry (TCR) Design Patterns. Available at: https://hackernoon.com/token-curated-registry-tcr-design-patterns-4de6d18efa15 [Accessed September 8, 2018].
28. Luu, L. et al., 2016. Making smart contracts smarter. In *Proceedings of the ACM Conference on Computer and Communications Security*. pp. 254–269.
29. Luu, L. et al., 2015. Demystifying Incentives in the Consensus Computer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS ’15*. pp. 706–719. Available at: http://dl.acm.org/citation.cfm?doid=2810103.2813659.
30. Mast, K., Chen, L. & Sirer, E.G., 2018. Enabling Strong Database Integrity using Trusted Execution Environments. *arXiv:1801.01618v2*. Available at: http://arxiv.org/abs/1801.01618.
31. Micali, S., 1994. {CS} Proofs (Extended Abstracts). In pp. 436–453.
32. Open Source Contributors, Oyente: An analysis tool for smart contracts. Available at: https://github.com/melonproject/oyente [Accessed September 8, 2018].
33. Palladino, S., 2017. The Parity Wallet Hack Explained. *Zeppelin.Solutions*, p.19. Available at: https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7. [Accessed September 2, 2018].
34. Patrov, S. Another Parity Wallet Hack Explained. Available at: https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c [Accessed September 2, 2018].
35. Pang, H., Zhang, J. & Mouratidis, K., 2009. Scalable Verification for Outsourced Dynamic Databases. *Proceedings of the VLDB Endowment*, 2(Privacy I). Available at: http://www.vldb.org/pvldb/2/vldb09-625.pdf.
36. Parno, B. et al., 2013. Pinocchio: Nearly practical verifiable computation. *Proceedings - IEEE Symposium on Security and Privacy*.
37. Popejoy, S., 2017. *The Pact Smart-Contract Language*, Available at: http://kadena.io.
38. Setty, S. et al., 2013. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, pp. 71–84.

39. Setty, S. & McPherson, R., 2012. Making argument systems for outsourced computation practical (sometimes). *NDSS*, 1(9), p.17. Available at: http://www.cs.utexas.edu/pepper/pepper-ndss12.pdf.
40. Siegel, D., 2016. Understanding the DAO attack. *Web. http://www. coindesk. com/understanding-dao-hack-journalists*. [Accessed September 2, 2018].
41. Sirer, E.G., 2016. Thoughts on The DAO hack, 2016. Available at: http://hackingdistributed.com/2016/06/17/thoughts-on-the-dao-hack/ [Accessed September 8, 2018].
42. Teutsch, J. & Reitwießner, C., 2017. A scalable verification solution for blockchains. p.50. Available at: https://people.cs.uchicago.edu/~teutsch/papers/truebit.pdf.
43. Von Krogh, G., Spaeth, S. & Lakhani, K.R., 2003. Community, joining, and specialization in open source software innovation: A case study. *Research policy*, 32(7), pp.1217–1241.
44. Wahby, R.S. et al., 2017. Full Accounting for Verifiable Outsourcing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS '17*. pp. 2071–2086. Available at: http://dl.acm.org/citation.cfm?doid=3133956.3133984.
45. Walfish, M. & Blumberg, A.J., 2015. Verifying computations without reexecuting them. *Communications of the ACM*, 58(2), pp.74–84. Available at: http://dl.acm.org/citation.cfm?doid=2728770.2641562.
46. Wood, G., 2014. Ethereum: a secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*.
47. Xiaoqi, Lyu & Liu, 2004. A Trust Model Based Routing Protocol for Secure Ad Hoc Networks. *Aerospace Conference, 2004. Proceedings. 2004 IEEE*, (April 2004), pp.1286–1295.



Appendix A: Verification Scheme Process Chart



```
translate(
   days(30),
   both(
         scale(
               50,
               50,
               transfer(ALICECOIN, address, alice, bob)
              ),
         scale(
               400,
               400,
               transfer(BOBCOIN, address, bob, alice)
               )
          )
)
```

```
60606040525b600080fd0
0a165627a7a7230582012
c9bd00152fa1c480f6827
f81515bb19c3e63bf7ed9
ffbb5fda0265983ac7980
```

Appendix B: Verification Scheme Step 1-4