Verifiable Standardized Smart Contracts

Sebastián Chamena

chamenas@pm.me

Abstract

Since the smart contracts' first formal and complete approach by the Ethereum protocol, a universe of applications were under scrutiny to tackle real-world use cases. However, this potential is limited by important problems that the emerging industry must solve to improve its expansion.

This article shows problems that are fundamentally hard to solve for any smart contract party by itself, but can be sorted with standardization. The problems addressed here are: software bugs together with unfamiliarity towards limitations as business' concern and high decision making and mental transaction costs as customers', while both carry high bargaining costs.

What the Verifiable Standardized Smart Contracts aims to achieve is to improve more use case viability through standardized contracts that should be certified with as little trust as possible, providing a better solution for both commercial and technical aspects.

1 Introduction

From it's idea conception of smart contracts by Szabo up to the Ethereum's realisation, these promised a new rule set for facing trustless, transparent, censorship resistant and secure agreements. Also, in contrast to Bitcoin's protocol which uses a Turing-incomplete programming language [1], fully-featured smart contracts can be developed using a complete one [2] such as Solidity. Potentially, this allows a universe of real-world use cases, such as mortgage, insurance, securities, land recording and many others [9]. These use cases appear to be limited to categories related to the development ecosystem, even though research and development are growing consistently over time [23]. This is noticeable as smart contracts are widely adopted in exchanges (i.e: swapping, trading), token systems, ENS [3] and collectibles [4].

This article will address the problem's root cause and propose a solution: the Verifiable Standardized Smart Contracts (VSSC), smart contracts that easily verify it's following a certain open Standard Smart Contract (SSC) of its business. Those SSC could be conceived as an ISO standard for smart contracts, and should address a similar procedure to achieve more practical adoption.

2 The costs of universal adoption

The adoption of smart contracts for all-purpose uses addresses several problems that some businesses cannot handle appropriately.

2.1 High bargaining costs

Without any competition, the bargaining costs are higher due to coordination failures. To illustrate this, turn to the analysis of the Nash's bargaining-by-demands game [5]:

Suppose that two parties have one dollar to divide, being the dollar the maximal wealth attainable from exchange between two parties. For example, it can represent the value a potential buyer pays for an object to its owner. The rules of the game are the following: Each of the two parties, A and B, makes a demand, a and b. If the total demand is less or equal than the available resources (i.e: $a + b \leq 1$), then each party gets what it demands. Otherwise, both parties get zero payoff.

See that for any pair a, b of positive numbers such as $a + b \leq 1$, there is a Nash equilibrium with the respective payoffs a and b. Notice that this promotes an inefficient usage of the total available resources (because the total sum can be lower than one, without taking advantage of the surplus). Moreover, there is a Nash equilibrium at (a=b=*1)* which results in a zero pay-off for both parties, the most inefficient outcome from the possible scenarios.

Remarkably, introducing competition virtually eliminates the tendency to such coordination failures.

For instance, suppose a game with two sellers S_1, S_2 and one buyer B. Then, the demands are coordinated if either $b + s_1 \leq 1$or $b + s_2 \leq 1$, giving —as in any competitive market— the surplus to the buyer. In this new version, all the equilibria are efficient.

2.2. High customer mental transaction costs

The lack of accurate user interfaces of contractual information implies hidden actions (i.e: contractual actions that aren't revealed to the customer), and guarding customers against hidden actions with a wider contractual detail for corner-cases could be both insufficient and costly [6], and therefore raising the customer acquisition costs.

2.3 Costly Decision Making

As contracts aren't standardized, the customer might have multiple completely different code versions of a contract, with an *apparently* equal utility function on both. In

addition with the already mentioned incomplete observation of the product's attributes, this means a lack of stimulus for expressing preferences. This cuts off the potential that smart contracts have to reduce mental transaction costs [7].

For example, suppose having A_1, ..., A_n providers of insurance, each offering its own smart contract SC_1, ..., SC_n respectively. Then, a consumer should compare its utility maximization for all *n* contracts.

The problem is that, as long as the observation of any (and therefore, of all) contracts is incomplete, the utility function has hidden benefits (or costs) associated.

2.4 Unfamiliarity of limitations

Companies need to familiarize themselves with the technology limitations in terms of scalability, security and profitability to take investment decisions.

In addition, the risks associated with inefficient implementations to establish industry standards are high [8].

2.5 Bugs

Smart contract development needs to be liable regarding software bugs to achieve real-world implementations [9]. To do a proper analysis for preventing bugs that could cause malfunction or vulnerabilities involves multiple challenges that are far from being resolved [10]. Nowadays, there are many tools for analyzing potential security breaches and malfunctions [11][12], but they aren't sufficient to provide reliability, and mostly can be considered experimental [13] [14] [22]. Finally, this implies a new 'quality control' step on top of the rest, the so-called security audits. But this creates two new problems: the customer's trust on those auditors (that are profitable through the contract owner) and the high costs of renewing auditions in a reasonable amount of time (e.g: suppose a new bug is discovered tomorrow for all contracts in a specific compiler version; does this affects your contract?).

Moreover, all of the bug-bounty programs that today are common for detecting these types of issues can't be as useful as for off-chain software, because any bug discovery implies risks by allowing anonymous exploiting for a zero-day vulnerability until the contract falls unused and replaced with a new one.

Finally, smart contracts will be more sophisticated in the long run, and are not only capable of being a solution for automating contracts, but for fully creating undiscovered ones under the new capabilities of the system [9].

3 Standardized smart contracts

To reduce the impact of these issues, the industry could follow open standards that already eliminate or diminish the potential harmness.

To achieve that, we define a Standard Smart Contract (SSC) as a smart contract that acts as a rule set for smart contract development. It should have:

- Well-defined terms: terms for both parties should be widely explained in a user-friendly while accurate interface, including not just covered but uncovered possible cases.
- Detailed and open security audits: security issues and audits must be open, and technical implications should be communicated in natural language.
- Targeted use cases: previous to its creation, any standard should have a set of business targets to improve adoption.
- Limitation measurement: the standard's limitations in terms of both scalability and case coverage should be measured.
- Deterministic computational costs: as these represent the network's fees for using a contract, the standard should give accurate computational cost estimations.
- Flexibility: standards should be malleable enough to admit parameterized or changed businesses' implementations.

3.1 Development

The development of SSCs should belong to an open, voluntary organization, composed mainly by industry experts, and follow a development process similar to the ISO standardization procedure [15].

Once released, the SSC would consist of two records stored on the blockchain: its source code and its bytecode.

Additionally, the organization should promote coordination for implementations by nature, standardizing frictionless updates. For instance, suppose that an SSC has a bug resolved in a new version. Then, the organization should give instructions for migrating (see for example the proxy-pattern implementation below) and detailed information about the probleem's impact to both parties of any implementation of the buggy SSC.

4 Verifiable Standardized Smart Contracts

Once having an SSC, we call a Verifiable Standardized Smart Contract (VSSC) a smart contract which anyone can verify that it follows a specific SSC.

That means that any potential customer must be able to check the standardization of the smart contract at any time. Therefore, he could consider all the standardized information proportionated by the SSC it follows.

Notice that this is crucial to remove the "reasonableness" interpretation of standards [9], guided by obscure or complex principles.

To define what "follow the standard" means, it's useful to clarify how SSC could be implemented. There are two ways to achieve it.

4.1 Parameterization

We define a parametrization implementation as one that delegates the call (e.g: using *delegateCall()* in Solidity) to a parameterized SSC.

For example, suppose having a SSC defines the tuple of parameters (PREMIUM_PRICE_GWEI, MAX_COMPENSATION_ETH, ORACLE_ADDRESS). Then, a valid parameterization of this SSC could be a new implementation which uses *delegateCall(2634300, 10, 0x514910771af9ca65…)*.

This can be achieved by using a proxy contract [16], and it also encourages a better adoption of updated standards.

Additionally, this methodology implies a user-friendly information about the implementation details for free by giving all the predetermined parameters values over the standard.

4.2 Code injection and subtraction

Code injection or subtractions are defined as implementations which extend or reduce the standard functionality by cutting off code or adding new one. With an extended version of the compiler, such as solc-verify [17], injections and subtractions from the SSC could be achieved while following a predetermined set of rules set in annotations.

After doing an implementation, both source code and bytecode must be stored on-chain for verification purposes. The process of validating the implementations' correctness is detailed below.

4.3 Source verification process

To validate that any SSC implementation is correct, we need to verify its source code relation with the bytecode stored on-chain. There are already platforms that do so, but involving trust since they rely on centralized companies [18]. Also, there are many tools used mostly by technical people to analyze transparency of a contract [19] [20]. The latter ones could be useful to achieve this goal if using more user-friendly interfaces.

The following is the definition of a process for doing a client-side (i.e: trustless) computation to perform a source code verification.

Suppose we want to verify that a source of a bytecode contract BC is a source code contract SRC.

1. Download the same compiler which produced the BC (written on the SRC).
2. Verify the compiler's integrity using the respective public checksum.
3. Check if the SRC compiles to BC.

Notice that the process could be benefited by using an on-browser compiler, by using a cross-compiled compiler such as solc-js [21]. With this, the source verification could be fully-client side and directly within the browser.

4.4 Standardization verification process

Once a smart contract's source code is known, we must ensure that its source is a correct implementation of the standard.

Suppose we want to verify that a contract with the verified source SRC is a valid implementation of an SSC, which has its bytecode SBC and source code SSRC stored on-chain.

1. Check if SRC is a proxy that delegates the call to the SBC_ADDRESS, being capable of pre-defining parameters. In which case, it corresponds to a valid parametrization implementation. Furthermore, here the process takes all the parameters it's defining for the call to later give them to the user (e.g: if defining on proxy the parameter 'PRICE' as '20', then the user would receive {"price": 20} after the successful verification).
2. Check if its a valid code injection, for which:
    a. Check if there are out-of-scope call delegations (i.e: *delegateCall* to addresses not admitted by the standard). If it does, the verification isn't successful.

b. Check if SRC public or external functions are a subset of the SSRC public or external functions. If it's a strict subset, then those missing interface implementations should be informed to the user. If it isn't, the verification isn't successful.
c. For each public or external function f in SRC, verify that the annotations made on SSRC's equivalent function f hold for its definition.

See that the verification software performing those steps could also receive an input which type of implementation it's checking to avoid computing any of the steps. Also, an on-browser experience could be possible as well for the source verification.

5 Open problems and known limitations

5.1 Proof of compiling

Create a proof that the source code SC corresponds to the bytecode BC with a more efficient algorithm than compilation.

If the algorithm is efficient enough to be stored in a contract, then the verification process can be done on-chain, without the need to repeat the verification on the client's side with a trustless alternative.

Zero-knowledge: If the proof of compiling existence holds, then its zero-knowledge form could address verification of private source code, and therefore allowing for private VSSCs.

Notice that private VSSCs wouldn't be much private if the respective SSC is public. This limitation is addressed in the following section.

5.2 Code obfuscation

This is an old problem in cryptography. Simplifying its formulation, the problem is to create a bytecode which cannot be reverse-engineered to get information about it, but can work as expected.

Its solution would carry over private calls to smart contracts, encrypting the parameters with a shared key stored in the obfuscated contract. Therefore, it can allow permissioned contracts in a public blockchain by validating the parameter encryption's correctness.

The resolution of this problem allows the creation of permissioned SSCs.

6 Conclusion

We defined Standard Smart Contracts as standards that would be achieving a business-oriented goal to promote efficient, more secure and widely documented smart contracts. This implies a higher adoption of this technology from both parties to tackle more real-world applications, giving remarkably lower transaction costs and democratizing high quality software with a minimum overhead on the implementation.

Also, we sketched a client-side process which certifies the correctness of an implementation, without having to trust any certification body and having a proof on-demand. We called these verified standardized contracts as Verifiable Standardized Smart Contract (VSSC).

7 References

[1] Bitcoin: A Peer-to-Peer Electronic Cash System. Satoshi Nakamoto. https://bitcoin.org/bitcoin.pdf

[2] A Next-Generation Smart Contract and Decentralized Application Platform. Ethereum Foundation, Vitalik Buterin. https://ethereum.org/en/whitepaper/

[3] https://ens.domains

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

[5] Bargaining costs, influence costs, and the organization of economic activity. Milgrom, Roberts. https://web.stanford.edu/~milgrom/publishedarticles/BargainingCosts.pdf

[6] Micropayments and Mental Transaction Costs. Nick Szabo. https://nakamotoinstitute.org/static/docs/micropayments-and-mental-transaction-costs.pdf

[7] Unenumerated: Smart contracts reduce mental transaction costs. Nick Szabo. https://unenumerated.blogspot.com/2006/04/smart-contracts-reduce-mental.html

[8] Blockchain in insurance – opportunity or threat? McKinsey & Company. https://www.mckinsey.com/~/media/McKinsey/Industries/Financial%20Services/Our%20Insights/Blockchain%20in%20insurance%20opportunity%20or%20threat/Blockchain-in-insurance-opportunity-or-threat.ashx

[9] Smart Contracts - 12 Use Cases for Business & Beyond. Smart Contracts Alliance. https://digitalchamber.org/assets/smart-contracts-12-use-cases-for-business-and-beyond.pdf

[10] An Overview on Smart Contracts: Challenges, Advances and Platforms. Zibin Zheng, Shaoan Xie, Hong-Ning Dai, Weili Chen, Xiangping Chen, Jian Weng, Muhammad Imran. https://arxiv.org/pdf/1912.10370.pdf

[11] Blockchain-based smart contracts: A systematic mapping study. Maher Alharby, Aad van Moorsel. https://arxiv.org/ftp/arxiv/papers/1710/1710.06372.pdf

[12] Defects and Vulnerabilities in Smart Contracts, a Classification using the NIST Bugs Framework. Wesley Dingman, Aviel Cohen, Adam Lynch, Nick Ferrara. https://www.researchgate.net/publication/334908571_Defects_and_Vulnerabilities_in_Smart_Contracts_a_Classification_using_the_NIST_Bugs_Framework

[13] How Effective are Smart Contract Analysis Tools? Evaluating Smart Contract Static Analysis Tools Using Bug Injection. Asem Ghaleb, Karthik Pattabiraman, 2020. https://arxiv.org/pdf/2005.11613.pdf

[14] Detecting Nondeterministic Payment Bugs in Ethereum Smart Contracts. Shuai Wang, Chengyu Zhang, Zhendong Su. https://chengyuzhang.com/papers/oopsla19.pdf

[15] https://www.iso.org/developing-standards.html

[16] Proxy Patterns – OpenZeppelin. https://blog.openzeppelin.com/proxy-patterns

[17] solc-verify: A Modular Verifier for Solidity Smart Contracts. Ákos Hajdu, Dejan Jovanović. https://arxiv.org/abs/1907.04262

[18] https://etherscan.io/verifyContract

[19] https://github.com/ethereum/sourcify

[20] https://github.com/ConsenSys/bytecode-verifier

[21] https://github.com/ethereum/solc-js

[22] Ethereum smart contracts verification: a survey and a prototype tool. Vera Bogdanich Espina. https://lafhis.dc.uba.ar/users/~diegog/licTesis/2019-12-17-Vera-Bogdanich.pdf

[23] Smart Contracts on the Blockchain - A Bibliometric Analysis and Review. Lennart Ante. https://www.researchgate.net/publication/340646200_Smart_Contracts_on_the_Blockchain_-_A_Bibliometric_Analysis_and_Review