Implementation of Strain: A Secure Auction for Blockchains

LIWEN DAI, L7DAI@EDU.UWATERLOO.CA, 20552153

I present my implementation of Strain, a new auction protocol running on a simulated blockchain that guarantees bid confidentiality. I implemented Goldwasser-Micali Encryption and its AND variance, which are the used in Fischlin's protocol to compare bids without leakding their actual values. I also implemented the verifiable key distribution process before the auction begins. Then I tested and benchmarked my implementation. Finally, I discuss issues of Strain, including the future work of its migration to the Ethereum blockchain.

1 INTRODUCTION

A Blockchain is a distributed ledger that offers data integrity, immutability and transparency. A blockchain driven by smart contracts, like Ethereum [13], is ideal for conducting auctions. A bid that has been submitted to the contract can no longer be retracted, or modified, if this immutability is the desired behaviour. Today's smart contracts have been very attractive, and are undergoing rapid development, despite still being in the early experimental stage.

Because all transactions on the blockchain are transparent and traceable, data confidentiality on the blockchain can be a big problem under certain scenarios. That is to say, in order for everyone to be able to verify every transaction, it is inevitable that all the data inputs and outputs are visible to everyone. Moreover, the pseudonymity of public addresses can also be broken, linked to their real-world owners, simply by analyzing the blockchain [2]. These problems exist in not only Bitcoin [11] and Ethereum, but also in a lot of other cryptocurrencies and blockchain applications.

Some blockchain applications for privacy keeping exist, like Zerocoin [10], which however haven't been extended to smart contracts yet.

1.1 Problem Definition

Consider a procurement auction, or a reverse auction, where multiple suppliers (sellers), compete to obtain business from the one auctioner (the buyer). If we assume every supplier provides equivlent goods or services, the supplier submitting the lowest bid will win the auction. This can be easily implemented as a decentralized smart contract on Ethereum, using its programming language Solidity [1]. While such a smart contract can be programmed to avoid possible corruption, everyone's bidding price has to be revealed to all parties in order to determine the winner. Intuitively we can see that these bidding prices can be very sensitive and are usually prefered not being leaked to potential competitors or other buyers, who might at least use them to bargain for even lower prices in the future.

In order to keep the bidding privacy, one can implement the auction as Secure Multi-Parity computation [9]. However most MPCs require high interactivity, therefore are unsuitable for a smart contract on a blockchain.

1.2 This Report

I present my implementation of Strain("Secure aucTions foR blockchAINs")[5] targeting low latency on blockchains, in order to solve the problem presented above. The key parts of my implementation are:

1. Goldwasser-Micali (GM) encryption [7] and its AND variance [12];
2. Fishlin's semi-honest two-party comparsion protocol [6];
3. Verifiable private key distribution;
4. Auction protocol Simulation.

This reports describes my implementation of Strain, which is inevitably very similar to the original design of Strain by Florian K.[5]. If my desription is the same as Florian's design, that means I implemented exactly what is stated in the orignal paper. When there's a difference, that means I have made some changes in my implementation, either because I thought there was an error, or I was trying a better way to express the solution. Not everything in the orignal paper is mentioned in my report, which means I didn't implement them. After describing my implementation I give my testing results and benchmarks. Finally I discuss how to migrate Strain to Ethereum, as well as other possible improvements.

2 DESIGN AND IMPLEMENTATION

2.1 Programming Tools

I implemented Strain in Python 2.7. Libraries used include Gmpy2 for speeding up computation on big numbers, PyCrypto for generating primes and random numbers. My source code is available on Github at https://github.com/lwdai/Strain.

2.2 Comparing Two Bidding Prices

In order to determine the winner of the auction, we compare every pair of prices. While the order of the prices will be revealed, our goal its to do the comparison without leaking any actual value of the prices.

Consider that we evaluate two unsigned integers with $\eta = 32$ bits each: $v_i = v_{i,1}...v_{i,\eta}$ and $v_j = v_{j,1}...v_{j,\eta}$. We start the comparison from the most significant bit, moving to the least significant bit (form left to right). If $v_i > v_j$, we can find their first different bit: $\exists 1 \leq l \leq \eta$, s.t. $v_{i,l} = 1 \wedge v_{j,l} = 0$. Also $\forall 1 \leq u < l$, $v_{i,u} = v_{j,u}$. To rewrite, for $l = 1, ..., \eta$, we define variables

$$c_l = (v_{i,l} \wedge \neg v_{j,l} \wedge \bigwedge_{u=1}^{\eta} (v_{i,u} = v_{j,u}))$$

We compute $c_{1...\eta}$ for every position l. If $v_i > v_j$, exactly one of $c_l \in c_{1...\eta}$ will be 1; and if $v_i \leq v_j$, all $c_l \in c_{1...\eta}$ will be 0.

This formula to compute c_l can be exploited using homomorphic encryptions. Next I present GM encryption, which can be used to compute the ciphertext of c_l homomorphically without revealing v_i or v_j.

2.3 GM Encryption

GM encryption allows to do plaintext XOR and bit flipping operations directly on the ciphertexts. Its variance, GM-AND allows AND operations on plaintexts.

2.3.1 Key Generation. Two large, random, strong prime numbers, p, q are generated using *getStrongPrime*, with default (provable) false positive probability = 10^{-6}, such that p has 768 bits and p has 896 bits; and $p \equiv q \equiv 3 \mod 4$.

The public key is $pk^{GM} = n = pq$, and the private key $sk^{GM} = \frac{(p-1)(q-1)}{4}$. Also let $z = n - 1$. If not otherwise specified, all computations for encryption and decryption are done in group $Z_n = \{0, 1, ..., n-1\}$, that is, taking $\mod n$.

The reason to set p, q with lengths of 768, 896 bits are: 1) the interface requires the number of bits be a multiple of 128 and > 512; 2) avoids the condition where p, q are too close leading to factorization of n [4]. Although one may still argue that even if p, q have the same length, the probablity of factoring n is extremely low, I decided to set the length difference as an easy way to work around this "Small Prime Difference" problem.

2.3.2 Encrypt-GM. To encrypt a bit b, take a random number $r \in Z_n$, then

$$Enc^{GM}_{pk^{GM}}(b) = r^2 \cdot z^b \mod n$$

Every bit is encoded into a large integer. To encrypt an integer, every bit is encrypted independently.

2.3.3 Decrypt-GM. A large integer is decoded into a single bit:

$$Dec^{GM}_{sk^{GM}}(c) = 1 - (c^{sk^{GM}} \equiv 1 \mod n)$$

The *powmod* function in Gmpy2 is used to compute $c^{sk^{GM}} \mod n$, since $c^{sk^{GM}}$ is too large to be computed directly. If encryption is done correctly, $c \in Z_n$ and $jacobi(c, n) = 1$. Decryption is also done bit by bit independently.

2.3.4 GM-Homomorphic Properties. Now we can do plaintext XOR, bit flipping and re-encryption (ignoring input keys):

$$Dec^{GM}(Enc^{GM}(b_1) \cdot Enc^{GM}(b_2)) = b_1 \oplus b_2$$
$$Dec^{GM}(Enc^{GM}(b) \cdot z) = 1 - b$$
$$ReEnc^{GM}(c) = c \cdot Enc^{GM}(0)$$

Important: $Enc^{GM}(b_1) \cdot Enc^{GM}(b_2) \neq Enc^{GM}(b_1 \oplus b_2)$ in general, because the encryption results are random.

2.4 GM-AND Encryption

A single bit is encrypted into $\lambda' = 128$ random numbers:

$$Enc^{AND}(1) = (Enc^{GM}(0), ..., Enc^{GM}(0))$$
$$Enc^{AND}(0) = (Enc^{GM}(a_1), ..., Enc^{GM}(a_{\lambda'}))$$

where $a_{1...\lambda'}$ are λ' number of random bits.

Decryption is correct with probability $1 - 2^{-\lambda'}$:

$$Dec^{AND}(c_1, ..., c_{\lambda'}) = \begin{cases} 1, & \text{if } \forall c_i, Dec^{GM}(c_i) = 0 \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

Enc^{AND} is homomorphic w.r.t. AND operation: let $Enc^{AND}(b) = (c_1, ..., c_{\lambda'})$, $Enc^{AND}(b') = (c'_1, ..., c'_{\lambda'})$, then $Dec^{AND}(c_1 \cdot c'_1, ..., c_{\lambda'} \cdot c'_{\lambda'}) = b \wedge b'$.

An existing GM ciphertext $\gamma = Enc^{GM}(b)$ of bit b can be embeded into $Enc^{AND}(b) = (c_1, ..., c_{\lambda'})$ without decryption. Independently for each c_i, with probability 0.5 set $c_i = Enc^{GM}(0)$; and with probability 0.5 set $c_i = Enc^{GM}(0) \cdot \gamma \cdot z \mod n$.

2.5 Fischlin's Protocol

Recall in section 2.2 how c_l is computed. Note that $v_{i,u} = v_{j,u}$ is just $\neg(v_{i,u} \oplus v_{j,u})$. Using GM Encryption and GM-AND, c_l can be computed homomorphically on ciphertexts without decryption.

Every supplier S_i publishes its public key n_i, as well as its encrypted bid price $C_i = Enc^{GM}_{pk_i}(v_i)$. Another supplier S_j executes the following $Eval(C_i, n_i, v_j)$ function, to compute whether $v_i > v_j$, without revealing its own value v_j:

def Eval(C_i, n_i, v_j):
    $C_{i,j} = Enc^{GM}_{n_i}(v_j)$
    $C_{i,j}, C_i, C_j, n_i \Rightarrow$ homomorphically
        $c_{1,\ldots,\eta}$ encrypted by pk_i, as in 2.2 - 2.4
    $res_{i,j} = shuffle(c_1, \ldots, c_\eta)$.
    return $res_{i,j}$ to S_i.

S_i decrypts $res_{i,j}$. If all $c_{1\ldots\eta}$ are decrypted to 0, S_i knows $v_i \leq v_j$. If exactly one of $c_l \in c_{1\ldots\eta}$ is decrypted to 1, S_i knows $v_i > v_j$.

2.6 Comparison between Two Malicious Suppliers

A semi-honest judge A is introduced as a trusted third party (TTP), who may be curious but cannot collude with any supplier. The key observation is that, in $Eval$ function above, computing $res_{i,j}$ from $C_{i,j}, C_i, C_j, n_i$ can be done by the judge A, as no other information is needed.

The judge may not leak $C_{i,j}$ to S_i, as that reveals v_j to S_i. However, S_j may still cheat by providing wrong $C_{i,j}, C_j$, which encrypt different values. Therefore S_j needs to prove to the judge that $C_{i,j}$ and C_j encrypt the same input v_j.

S_j executes the following $ProofEval$ and sends the result to the judge.

def ProofEval(C_i, C_j, $C_{i,j}$, v_j, n_i, n_j, $\lambda'' = 16$):
    Toss $\eta \cdot \lambda''$ random bits $\delta_{l,m}, 1 \leq l \leq \eta, 1 \leq m \leq \lambda''$
    Compute $4 \cdot \eta \cdot \lambda''$ encryptions:
        $\gamma_{l,m} = Enc^{GM}_{pk_j}(\delta_{l,m})$
        $\gamma'_{l,m} = Enc^{GM}_{pk_i}(\delta_{l,m})$
        $\Gamma_{l,m} = Enc^{GM}_{pk_j}(\delta_{l,m} \oplus v_{j,l})$
        $\Gamma'_{l,m} = Enc^{GM}_{pk_i}(\delta_{l,m} \oplus v_{j,l})$
    Send all the ciphertexts above to the judge A as proof
    Set $h = Sha256(\gamma_{1..\eta,1..\lambda''}, \gamma'_{1..\eta,1..\lambda''}, \Gamma_{1..\eta,1..\lambda''}, \Gamma'_{1..\eta,1..\lambda''}, C_i, C_j, C_{i,j})$
        as the random seed
    Generate $\eta \cdot \lambda''$ bits $b_{l,m}$
    if $b_{l,m} = 0$, append $\delta_{l,m}$, and the random coins for $\gamma_{l,m}, \gamma'_{l,m}$ to the proof.
    if $b_{l,m} = 1$, append $\delta_{l,m} \oplus v_{l,m}$, and the random coins for $\Gamma_{l,m}, \Gamma'_{l,m}$ to the proof.

Note that the random coins for GM Encryption are just the random number r as in 2.3.2.

The judge A verifies the following; let $(C_i)_j$ denotes the j-th number in C_i:

1. Verifies the homomorphic relations of the ciphertexts:
$\gamma_{l,m} \cdot (C_j)_l = \Gamma_{l,m}$, and $\gamma'_{l,m} \cdot (C_i)_l = \Gamma'_{l,m}$

However it turns out that this verification doesn't work: as in 2.3.4, the homomorphic relations only hold when the ciphertexts are decrypted. There's no equality relation on ciphertexts directly, because the encryption results are random.

2. Checks that the ciphertexts above are computed correctly, by redoing the computations from the given random coins.

3. If checks pass, computes $res_{i,j}$ and sends to S_i.

Because verification 1 doesn't work, I couldn't detect whether S_j is cheating. To compromise, the judge just computes $res_{i,j}$ directly.

2.7 Key Distribution

Before the auction begins, every supplier $S_i \in S_{1..s}$ splits secret shares of $sk_i = \frac{(p_i-1)(q_i-1)}{4}$ into $s-1$ random secret shares $r_{1,\ldots,s-1} \in \{0, \ldots, 4sk_i\}$, such that $\sum_{j=1}^{s-1} r_j = sk_i \mod 4sk_i$. The original design is that each r_j is further split to $s-1$ secret shares using Shamir's secret shares, with some reconstruction threshold τ [8]. I didn't implement Shamir's shares therefore reconstructing a secret key requires all the $s-1$ suppliers.

To check the shares of a given sk_i, every other supplier needs to participate. First, every supplier needs to check the proof provided by S_i that n_i is indeed a Blum integer (not implemented). Then every supplier S_j submits a random integer $\rho_j \in [0, n_i - 1]$, and computes the following:

$$x_i = \sum_j \rho_j \mod n_i$$

$$y_i = x_i^2 \mod n_i$$

$$\gamma_{i,j} = y_i^{r_j} \mod n_i$$

$$\zeta_{i,j} = z_i^{r_j} \mod n_i$$

together with a proof P_j^{DLOG} for the statement $\log_{y_i} \gamma_{i,j} = \log_{z_i} \zeta_{i,j}$, with details coming next.

Each supplier S_j publishes $(\gamma_{i,j}, \zeta_{i,j}, P_j^{DLOG})$.

Then every supplier S_j checks whether S_i has computed $r_{1,\ldots,s-1}$ honestly. For correctness it must satisfy both $\prod_{j=1}^{s} \gamma_{i,j} = y_i^{\sum_{j=1}^{s-1} r_j} = y_i^{sk_i} \equiv 1 \mod n_i$, and $\prod_{j=1}^{s} \zeta_{i,j} = z_i^{\sum_{j=1}^{s-1} r_j} = z_i^{sk_i} \equiv -1 \mod n_i$. If not satisfied, S_j reports that S_i is cheating.

2.7.1 P_DLOG. For public values (y, z, γ, ζ), needs to prove $\log_y \gamma = log_z \zeta (= \sigma)$, without revealing σ [3].

```
def proofDlog(σ, y, n, K = 10):
        compute z, γ, ζ
        P_DLOG = []
        for i = 1...K:
            // r is not affected by setRandSeed
            r = rand1.randint(0, n-1) s.t. jacobi(r,n) = 1
            t1 = y^r mod n
            t2 = z^r mod n
            // generating c with a different random generator from r
            rand2.setRandSeed(Sha256([y, z, γ, ζ, t1, t2, i]))
            c = rand2.randint(0, n-1) s.t. jacobi(r,n) = 1
            s = r + c · σ
            P_DLOG.append((t1, t2, s))
        return P_DLOG
```

To verify *P_DLOG*, at each iteration, the verifier computes c according to the random oracle specified above. Then it verifies that $y^s \equiv t_1 \gamma^c \mod n$, and $z^s \equiv t_2 \zeta^c \mod n$.

Some details: *Crypto.random* (secure random) is used for *rand*1; and default python *random* is used for *rand*2, which is insecure but can be seeded.

2.7.2 Auction Simulation. The auction process is simulated on a mock blockchain in memory. The whole program runs on a single thread without block latency. As stated before, other suppliers will know if a given supplier has cheated when spliting its private key in the key distribution process. There are other possible cheating opportunities, which unfortunately remain undetected for now.

The auction process can be summarized as following:

1) Every supplier publishes its public key to the blockchain.
2) Every supplier performs key distribution and verification, as in section 2.7.
3) Every supplier commits a bid, GM-encrypted with its own public key, to the blockchain.
4) Every supplier generates *ProofEval* with every other supplier, then sends *ProofEval* to the judge privately.
5) The judge verifies all *ProofEval* (skipped), then publishes the encrypted comparison results to the blockchain.
6) Every supplier decrypts their comparison results, and publishes them to the blockchain.
7) The winner is determined from the comparison results on the blockchain.

Currently cheaters can be discovered at the key distribution stage, but it's unclear how to deal with cheaters cleanly.

3 BENCHMARK

I tested the run time for the functions I implemented. All benchmarks were tested on my laptop with i7-4700mq, in a single-thread setting. All timings below 10 seconds were measured from the total seconds of running 10 iterations, divided by 10. These benchmarks are for intuition purpose of how much extra computing is required for Strain, rather than the actual run time on the blockchain.

For GM key-pair generation, with $p = 768$ bits and $q = 896$ bits, generating a key pair takes about 1.5 seconds on average.

Calling Enc^{GM} then Dec^{GM} on a 32-bit unsigned integer takes about 0.07 seconds.

Calling Enc^{AND} then Dec^{AND} on a single bit, with $\lambda' = 128$ takes about 0.17 seconds.

Calling *Eval* then decrypting *res* using Fischlin's protocol takes about 7.5 seconds to compare one pair of 32-bit unsigned integers.

The total time (sum of all parties) used for key distribution is quadratic to the number of suppliers s. It takes about 9 seconds with $s = 2$; 23 seconds with $s = 3$; 49 seconds with $s = 4$; and 81 seconds with $s = 5$. Note that if Shamir's share is implemented, the total time should be $O(s^3)$ instead.

The total time of the whole auction process is also quadratic to the number of suppliers s. With $s = 2$, the auction takes about 26 seconds; 77 seconds with $s = 3$; 145 seconds with $s = 4$,and 244 seconds with $s = 5$. Table 1 summarizes my benchmark of total runtime vs number of suppliers.

Table 1. Total Runtime (seconds) vs Number of Suppliers

Suppliers $s =$	**Key Distribution** (*seconds*)	**Auction** (*seconds*)
2	9	26
3	23	77
4	49	45
5	81	244

4 DISCUSSION

4.1 Performance

Strain is designed to be efficient w.r.t. to latency. As the benchmark showed, the latency seems manageable when the total number of suppliers is small. Consider that an auction may last for hours or even days, Strain has the potential to be applied to a larger number of participants.

4.2 Reporting Malicious Parities

Reporting cheaters still seem to be a messy process. While we can use the judge as the "gold standard", Strain requires that every supplier also to report cheaters, as in the key distribution process. As a cheater may maliciously report that an honest party is cheating, we need to decide how to deal with situations like "S_1 reports S_2 cheats; S_2 reports S_3 cheats; S_3 reports S_1 cheats", unless we can prove such situations do not exist. Of course we may still rely on the judge to make a decision; or we can reassemble the private keys to discover the true cheaters – which will inevitably leak all bids.

4.3 Migrating to Ethereum Blockchain

Migrating Strain to the actual Ethereum blockchain turned out to be a lot of work, as I have only simulated the auction on a mock blockchain in memory, with no block latency. As a distributed application, smart contracts programmed in Solidity are very hard to debug, since it gives literally

no error message if the contract crashes on the blockchain. TestRPC can be used to simulate a complete blockchain on a local computer, but its debugging feature is limited to stepping the code, yet unable to print the value of any variable.

Node.js might be the most supported client framework to call smart contracts remotely. However javascript is not very good at heavy-lifting cryptographic computations. As a result, one may need to implement the cryptographic primitive of Strain in Python or even C. This leads to another layer of inter-process communication, which may require a lot of engineering.

5 CONCLUSION

I implemented Strain, a secure auction protocol for blockchains. I successfully implemented GM encryption , its AND variance, Fishlin's semi-honest two-party comparison protocol, and most of the verifiable private key distribution. Unfortunately there's a design error and the secure comparisons between malicious parties didn't work. I simulated the auction protocol in in memory. My benchmark results showed that Strain's latency is enough for an auction with not too many suppliers.

ACKNOWLEDGEMENT

Thanks Florian for giving me such an interesting project!

REFERENCES

[1] Solidity Documentation. https://solidity.readthedocs.io/en/develop/. (????).
[2] Alex Biryukov, Dmitry Khovratovich, and Ivan Pustogarov. 2014. Deanonymisation of clients in Bitcoin P2P network. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 15–29.
[3] David Chaum and Torben P Pedersen. 1992. Wallet Databases with Observers.. In *Crypto*, Vol. 92. Springer, 89–105.
[4] Benne De Weger. 2002. Cryptanalysis of RSA with small prime difference. *Applicable Algebra in Engineering, Communication and Computing* 13, 1 (2002), 17–28.
[5] Florian K. etc. 2017. Strain: A Secure Auction for Blockchains. (2017).
[6] Marc Fischlin. 2001. A cost-effective pay-per-multiplication comparison method for millionaires. *Topics in Cryptology CT-RSA 2001* (2001), 457–471.
[7] Shafi Goldwasser and Silvio Micali. 1984. Probabilistic encryption. *Journal of computer and system sciences* 28, 2 (1984), 270–299.
[8] Jonathan Katz and Moti Yung. 2002. Threshold cryptosystems based on factoring. *Advances in CryptologyâĂŤASIACRYPT 2002* (2002), 139–147.
[9] Ahmed Kosba, Andrew Miller, Elaine Shi, Zikai Wen, and Charalampos Papamanthou. 2016. Hawk: The blockchain model of cryptography and privacy-preserving smart contracts. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 839–858.
[10] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. 2013. Zerocoin: Anonymous distributed e-cash from bitcoin. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 397–411.
[11] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).
[12] Tomas Sander, Adam Young, and Moti Yung. 1999. Non-interactive cryptocomputing for NC/sup 1. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 554–566.
[13] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014).