[Obfuscation

](https://www.iacr.org/archive/crypto2001/21390001.pdf) is in many ways the ultimate cryptographic primitive. Obfuscation allows you to turn a program P

into an “obfuscated program” P'

such that (i) P'

is equivalent to P

, ie. P'(x) = P(x)

for all x

, and (ii) P'

reveals nothing about the “inner workings” of P

. For example, if P

does some computation that involves some secret key, P'

should not reveal that key.

Obfuscation is not yet available; candidate constructions exist, but they all depend on cryptographic assumptions that cryptographers are not happy with and some candidates have already been broken. However, recent research suggests that we are very close to secure obfuscation being possible, even if inefficient

.

The usual way to formalize the privacy property is that if there are two programs P

and Q

that implement the same functionality (ie. P(x) = Q(x)

for all x

) but maybe with different algorithms, then given obfuscate(P)

and obfuscate(Q)

you should not be able to tell which came from which (to see how this leads to secrecy of internal keys, consider a function that uses a key k

to sign a message out of the set [1....n]

; this could be implemented either by actually including k

and signing a message that passes a range check, or by simply precomputing and listing all n

signatures; the formal property implies that you can’t extract k

from the first program, because the second program does not even contain k

).

Obfuscation is considered to be so powerful because it immediately implies almost any other cryptographic primitive. For example:

- Public key encryption: let enc

/dec

be a symmetric encryption scheme (which can be implemented easily using just hashes): the secret key is k

, the public key is an obfuscated program of enc(k, x)

- Signatures: the signing key is k

, the verification key

is an obfuscated program that accepts M

and sig

and verifies that sig = hash(M, k)

- Fully homomorphic encryption: let enc

/dec

be a symmetric encryption scheme. The secret key is k

, the evaluation key is enc(k, dec(k, x1) + dec(k, x2))

and enc(k, dec(k, x1) * dec(k, x2))

- Zero knowledge proofs: an obfuscated program is published that accepts x

as input and publishes sign(k, x)

only if P(x) = 1

for some P

More generally, obfuscation is viewed as a potential technology for creating general-purpose privacy-preserving smart contracts. This post will both go into this potential and other applications of obfuscation to blockchains.

Smart contracts

Currently, the best available techniques for adding privacy to smart contracts use zero knowledge proofs, egAZTEC and Zexe. However, these techniques have an important limitation: they require the data in the contract to be broken up into "domains" where each domain is visible to a user and requires that user's active involvement to modify. For a currency system, this is acceptable: your balance is yours, and you need your permission to spend money anyway. You can send someone else money by creating encrypted receipts that they can claim. But for many applications this does not work; for example, something like Uniswap contains a very important core state object which is not owned by anyone. An auction could not be conducted fully privately; there needs to be someone to run the calculation to determine who wins, and they need to see the bid amounts to compute the winning bid.

Obfuscation allows us to get around this limitation, getting much closer to "perfect privacy". However, there is still a limitation remaining

. One can naively assume obfuscation lets you create contracts of the form "only if event X happens, then release data Y". However, outside observers can create a private fork of the blockchain, include and censor arbitrary transactions in this private fork (including copying over some but not all transactions from the main chain), and see the outputs of the contract in this private fork.

To give a particular example, key revocation for data vaults cannot work: if at some time in the past, a key k_1

could have released data D

, but now that key was switched in the smart contract to k_2

, then an attacker with k_1

could still locally rewind the chain to before the time of the switch, and send the transaction on this local chain where k_1

still suffices to release D

and see the result.

Obfuscating an auction is a particular example of this: even if the auction is obfuscated, you can determine others' bids by locally pretending to bid against them with every possible value, and seeing under what circumstances you win.

One can partially get around this, by requiring the obfuscated program to verify that an instruction was confirmed by the consensus, but this is not robust against failures of the blockchain (51% attacks or more than 1/3 going offline). Hence, it's a lower security level than the full blockchain. Another way to get around this is by having the obfuscated program check a PoW instance based on the inputs; this limits the amount of information an attacker can extract by making executions of the program more expensive.

With this restriction, however, more privacy with obfuscation is certainly possible. Auctions,voting schemes (including in DAOs), and much more are potential targets

.

Other benefits

- ZKPs with extremely cheap verification

: this is basically the scheme mentioned above. Generate an obfuscated program which performs some pre-specified computation f

on (x, y)

(x is the public input, y is the private input), and signs (x, f(x))

with an internal key k

. Verification is done by verifying the signature with the public key corresponding to k

. This is incredibly cheap because verifying a proof is just verifying a signature. Additionally, if the signature scheme used is BLS, verification becomes very easy to aggregate.

- One trusted setup to rule them all

: generating obfuscated programs will likely require a trusted setup. However, we can make a single obfuscated program that can generate all future trusted setups for all future protocols, without needing any further trust. This is done as follows. Create a program which contains a secret key k

, and takes as input a program P

. The program executes P(h(P, k))

(ie. it generates a subkey h(P, k)

specific to that program), and publishes the output and signs it with k

. Any future trusted setup can be done trustlessly by taking the program P

that computes the trusted setup and putting it into this trusted setup executor as an input.

- Better accumulators

: for example, given some data D

, one can generate in O(|D|)

time a set of elliptic curve point pairs (P, k*P)

where P = hash(i, D[i])

and k

is an internal secret key (K = k*G

is a public verification key). This allows verifying any of these point pairs (P1, P2)

by doing a pairing check e(P1, K) = e(P2, G)

(this is the same technique as inolder ZK-SNARK protocols). Particularly, notice that a single pairing check also suffices to verify any subset of the points. Even better constructions are likely possible.