ECDSA Verification in Private Kernel Circuit

tags: aztec3-speccing-book

The private kernel circuit is tasked with validating two types of transactions:

1. Contract deployment,
2. Private function execution.

The private kernel circuit iterates over the items in the call stack to validate each of the calls and aggregated the verification information of the respective function proof. For the very first iteration of the private kernel circuit, we need to validate if the transaction is coming from a legitimate user. For this, the user has to sign over the transaction and the private kernel circuit can then validate the signature to confirm that the transaction was submitted by the rightful owner.

ECDSA Algorithms

Since our users will be Ethereum users, they will be signing the transactions with their Ethereum keys (i.e. secp256k1 public keys). Suppose the private key of Alice is p\in\mathbb{F}_r

where \mathbb{F}_r

is the subgroup field of the secp256k1 curve.

1. Private key: p\in\mathbb{F}_r

,

1. Public key:

P := p * G \tag{1}

where G\in\mathbb{G}_{\textsf{secp256k1}}

is the generator of the secp256k1 curve,

1. Ethereum address:

E := \textsf{uint160}(\textsf{uint256}(\textsf{keccak}(P))) \tag{2}

1. ECDSA signature construction:

\begin{aligned} & \textsf{ecdsa_sign}(m, p):\ & \qquad k \leftarrow \textsf{hmac}(m, p) & & \in \mathbb{F}_r \ & \qquad R := k * G & & \in \mathbb{G}_\textsf{secp256k1} \ & \qquad r := R_x \text{ mod } r & & \in \mathbb{F}_r \ & \qquad z := \textsf{hash}(m) & & \in \mathbb{F}_r \ & \qquad s := k^{-1} \cdot (z + r \cdot p) & & \in \mathbb{F}_r \ & \qquad \text{return } (r, s) & & \in \mathbb{F}_r^2 \end{aligned} \tag{3}

1. ECDSA signature verification:

\begin{aligned} & \textsf{ecdsa_verify}((r, s), m, P):\ & \qquad z := \textsf{hash}(m) \ & \qquad u_1 := z \cdot s^{-1} & & \small{//} \ \ u_1 = z \cdot k \cdot (z + r \cdot p)^{-1} \ & \qquad u_2 := r \cdot s^{-1} & & \small{//} \ \ u_2 = r \cdot k \cdot (z + r \cdot p)^{-1} \ & \qquad R' := (u_1 * G + u_2 * P) & & \small{//} \ \ R' = (u_1 + u_2 \cdot p) * G \ & & & \small{//} \hspace{0.65cm} = [z \cdot k \cdot (z + r \cdot p)^{-1} \ + \ & & & \small{//} \hspace{1.4cm} r \cdot k \cdot (z + r \cdot p)^{-1} \cdot p] * G \ & & & \small{//} \hspace{0.65cm} = k * G \ & \qquad \text{return } R_x \stackrel{?}{=} r \end{aligned} \tag{4}

1. ECDSA key recovery:

\begin{aligned} & \textsf{ecdsa_pubkey_recover}((r, s), m, P):\ & \qquad z := \textsf{hash}(m) \ & \qquad (y_+, y_-) \leftarrow \left(\pm \sqrt{x^3 + Ax + B}\right)_{x = r} \ & \qquad R_+ \equiv (r, y_+) & & \small{// \ \textsf{positive y-coordinate}} \ & \qquad R_- \equiv (r, y_-) & & \small{// \ \textsf{negative y-coordinate}} \ & \qquad L_\pm := s * R_{\pm} & & \small{//} \ \ L = (s \cdot k) * G \ & & & \small{//} \hspace{0.55cm} = (z + r \cdot p) * G \ & & & \small{//} \hspace{0.55cm} = z * G + r * P \ & \qquad P_\pm = r^{-1} * (L_\pm - z * G) & & \ & \qquad \quad = (r^{-1} \cdot s) * R_\pm \ + \ (–r^{-1}\cdot z) * G & & \ & \qquad \text{return } (P_+, P_-) \end{aligned} \tag{5}

Key recovery in ECDSA needs a variable base size-2 MSM just like ECDSA verification. Additionally, it requires us to extract the y-coordinate of a point given its x-cordinate. Therefore, the cost of key recovery in a circuit would be more than the cost of verifying an ECDSA signature in a circuit.

Problem at Hand

The TxContext

has the from

and to

fields which are currently set to address

type. Behind the scenes, the address

is just a field_t

and refers to an Ethereum address. An ethereum address is derived from the secp256k1 public key as shown in equation (2)

.

The private kernel circuit gets E

(from: address

) as one of the inputs and it also gets an ECDSA signature (r, s)

signed by the user. The private kernel circuit requires the public key P

to verify the signature (r, s)

, as shown in equation (5)

. It is possible to recover the public key P

from the signature itself (see equation (6)

) with an additional cost. Let us look a few way of getting around this.

[A] Recover public key from the signature in the circuit:

- \textsf{Solution}

: * The TxContext

remains unchanged,

- We recover the signing public key from the signature (r,s)

using equation (5)

,

- Then proceed to verify the signature using the recovered public key.
- The TxContext

remains unchanged,

- We recover the signing public key from the signature (r,s)

using equation (5)

,

- Then proceed to verify the signature using the recovered public key.
- \color{green}{\textsf{Pros}}

: * Nothing needs to be changed in the interfaces (circuit inputs and related TS code),

- Safely use TxContext

with only ethereum addresses without any public keys.

- Nothing needs to be changed in the interfaces (circuit inputs and related TS code),
- Safely use TxContext

with only ethereum addresses without any public keys.

- \color{red}{\textsf{Cons}}

: * Expensive because recovering a public key costs more than verifying the signature.

- For secp256k1, there are two possible y-coordinates for a given x-coordinate. This would mean that we need to verify the signature against both possible public keys.
- Signature verification with two public keys might not mean twice verification costs because we need to verify signature against two public keys P_1, P_2

such that P_1 + P_2 = \mathcal{O}

. So optimising verification of signature against such special-case public keys is possible.

- Overall, additional circuit cost would be in the range of 40,000 gates primarily due to public key recovery.
- Expensive because recovering a public key costs more than verifying the signature.
- For secp256k1, there are two possible y-coordinates for a given x-coordinate. This would mean that we need to verify the signature against both possible public keys.
- Signature verification with two public keys might not mean twice verification costs because we need to verify signature against two public keys P_1, P_2

such that P_1 + P_2 = \mathcal{O}

. So optimising verification of signature against such special-case public keys is possible.

- Overall, additional circuit cost would be in the range of 40,000 gates primarily due to public key recovery.

[B] Modify the TxContext

to include a new type from_public_key

- \textsf{Solution}

: * The TxContext

will include a new type from_public_key: secp256k1_point

along with the corresponding ethereum address from

.

- This new from_public_key

will need to be recovered from the signature (r,s)

either in native C++ or in TS (not recovered in the circuit).

- Then we can proceed to verify the signature using the new input from_public_key

.

- The TxContext

will include a new type from_public_key: secp256k1_point

along with the corresponding ethereum address from

.

- This new from_public_key

will need to be recovered from the signature (r,s)

either in native C++ or in TS (not recovered in the circuit).

- Then we can proceed to verify the signature using the new input from_public_key

.

- \color{green}{\textsf{Pros}}

: * No circuit-costs for recovering the public key,

- We can natively recover the public key and pass it as a circuit input (i.e. from_public_key

)^{\ast}

. In this case, TS does not need to do any additional computation.

- No circuit-costs for recovering the public key,
- We can natively recover the public key and pass it as a circuit input (i.e. from_public_key

)^{\ast}

. In this case, TS does not need to do any additional computation.

- \color{red}{\textsf{Cons}}

: * Circuit will need to verify the relation between the address from

and the public key from_public_key

using equation (2)

. A single-block keccak hash costs 17,000.^{\dagger}

- Signature structure should be changed to (r,s,v)

where v

encodes which of the two possible public keys was used to sign the transaction.

- Circuit will need to verify the relation between the address from

and the public key from_public_key

using equation (2)

. A single-block keccak hash costs 17,000.^{\dagger}

- Signature structure should be changed to (r,s,v)

where v

encodes which of the two possible public keys was used to sign the transaction.

[C] Replace all address

types with curve points

- \textsf{Solution}

: * Change the types of from, to

in TxContext

from address

to secp256k1_point

and do away with addresses altogether.

- It will be TS’s responsibility to supply only the public keys (even if it only has access to user’s ethereum address, it will have to recover public key(s) from the signature).
- Change the types of from, to

in TxContext

from address

to secp256k1_point

and do away with addresses altogether.

- It will be TS's responsibility to supply only the public keys (even if it only has access to user's ethereum address, it will have to recover public key(s) from the signature).
- \color{green}{\textsf{Pros}}

: * No address

type in stdlib

so no need to deal with any ethereum addresses

- No conversions between address

and public_key

anywhere in C++ circuits or natively.

- No address

type in stdlib

so no need to deal with any ethereum addresses

- No conversions between address

and public_key

anywhere in C++ circuits or natively.

- \color{red}{\textsf{Cons}}

: * TS will have to figure out how to fetch or compute public keys.

- The TxContext

structure will no longer be analogous to TxRequest

in ethereum transaction.

- TS will have to figure out how to fetch or compute public keys.
- The TxContext

structure will no longer be analogous to TxRequest

in ethereum transaction.

Conclusion

Clearly, second solution seems to be the best in terms of additional gate costs and time required to implement the change.

^{\dagger}{\scriptsize \textsf{Using a field-friendly hash function is not possible here because of the definition of how an ethereum address is computed.}}

^{\ast}{\scriptsize \textsf{Thanks to Mike for this suggestion.}}