Assignment 5

Q1 Shielding the TX

In week 2 you learnt about one of the three use cases of ZK, **Privacy**, and how zero-knowledge can be used to shield the identity of both parties in a transaction. This week we will be focusing on privacy within the context of bridges. This is especially useful as we move into a cross-chain era in blockchain and users require the same level of anonymity when bridging their tokens across different chains

1. You have been asked to present a mechanism that will allow a user to privately and securely bridge 100,000 UST tokens from Ethereum to Harmony. Draft a write-up explaining the protocol to be built to cater for this need, highlighting the challenges to be faced and potential resolution to them.

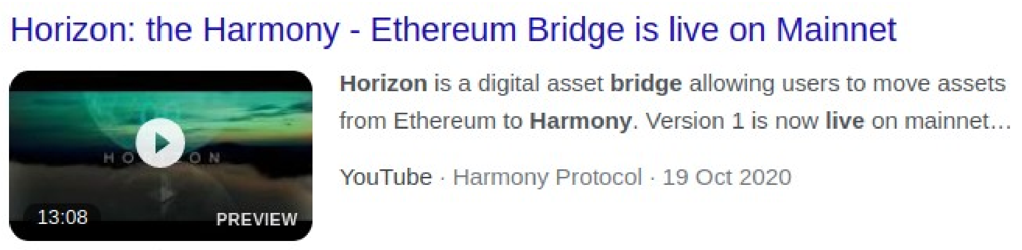
Answer 1.1: (For 100000 UST**) - (Converted to ERC20 Tokens)**

For the fact that webb protocol is not deployed on main net at the moment, I would use AZTECK's join-split protocol to annonymesly mix ethereum tokens and use Horizon bridge to bridge ETH tokens to Harmony.

I would use tornado cash nova to mix arbitrary amounts of ETH tokens and use Horizon bridge to transfer tokens to Harmony.

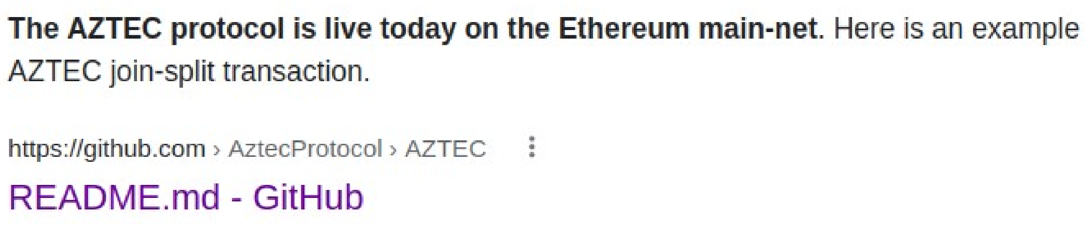
I prefer using AZTEC's notes for UTXOs commitments, However I am not sure about the security assumptions under elliptic curves choosing mechanism for the initial setup.

However, if I already mixed ETH tokens via AZTEC & Horizon bridge functioning on mainnet there is no need for WEBB protocol


Horizon: the Harmony - Ethereum Bridge is live on Mainnet

Horizon is a digital asset **bridge** allowing users to move assets from Ethereum to **Harmony**. Version 1 is now **live** on mainnet...

YouTube · Harmony Protocol · 19 Oct 2020


The AZTEC protocol is live today on the Ethereum main-net. Here is an example AZTEC join-split transaction.

https://github.com › AztecProtocol › AZTEC

README.md - GitHub

2. In February 2022, a hack on a popular crypto bridge led to the second biggest crypto heist where $320m was lost. Following the technical details behind the hack, it is very clear that bridge smart contracts need to be airtight to prevent scrupulous individuals from taking advantage of them. Briefly explain key mechanisms you will put in place in your interoperable private bridge (specifically the withdrawal methods) to prevent a similar attack (Double withdrawal and fake withdrawal).

Answer 1.2:

Solana attack overview: (Fake Withdrawal Attack**)** **Attack against Solana VAA verification and mint tokens**

@ https://twitter.com/kelvinfichter/status/1489041221947375616?lang=en --[Posted 20 mins after the attack]--[Attack mounted few hours after replacing usage of "load_instruction_at" with "load_instruction_at_checked"]

1. The contract might've incorrectly validated the signatures on the transfer, but the signatures completely checked out, As The Wormhole "guardians" had somehow signed off on this 80k ETH transfer as if it were 100% legit.

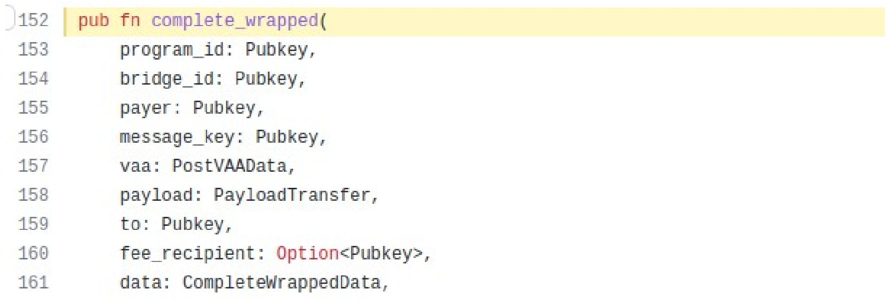
2. The attacker was able to mint Wormhole ETH on Solana, so they were able to correctly withdraw it back to Ethereum

3. The next interesting piece of information is this Solana transaction that came right before the 120k ETH one, where 0.1 Wormhole ETH was minted on Solana

4. the attacker *did* make a deposit of 0.1 ETH *into* Solana from Ethereum:

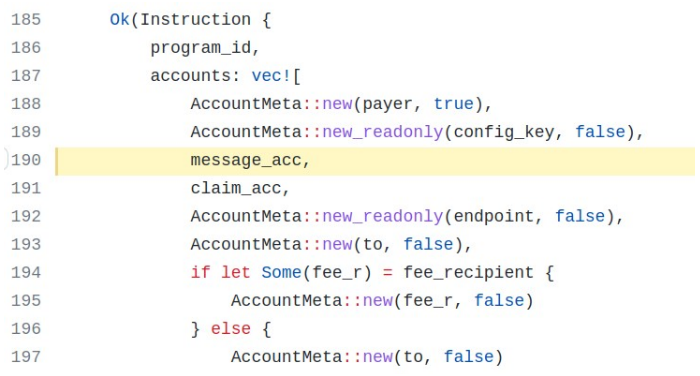
5. what happened wrong with solana contracts that ends up with allowing the attacker to exploit solana contracts with a deposit of 0.1 ETH from ethereum to solana?

6. The transactions that minted Wormhole ETH on Solana were triggering this Wormhole function "complete_wrapped"



```
pub fn complete_wrapped(
    program_id: Pubkey,
    bridge_id: Pubkey,
    payer: Pubkey,
    message_key: Pubkey,
    vaa: PostVAAData,
    payload: PayloadTransfer,
    to: Pubkey,
    fee_recipient: Option<Pubkey>,
    data: CompleteWrappedData,
```

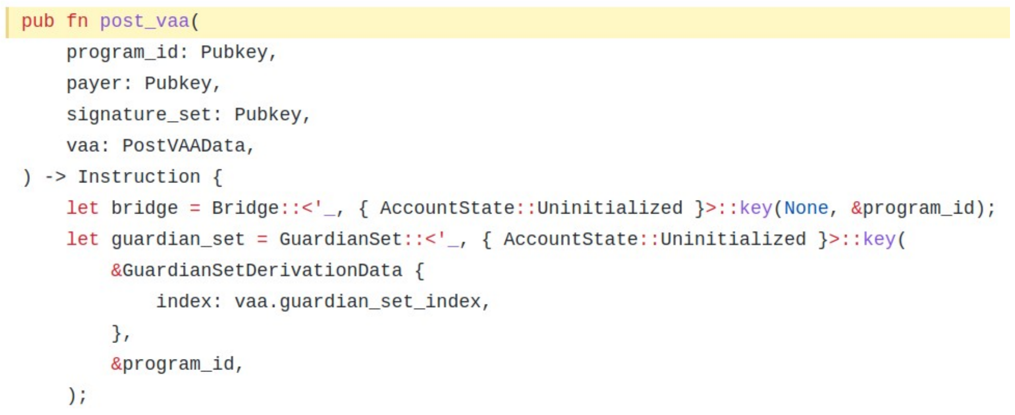
7. One of the parameters that this function takes is a "transfer message", basically a message signed by the guardians that says which token to mint and how much



```
    Ok(Instruction {
        program_id,
        accounts: vec![
            AccountMeta::new(payer, true),
            AccountMeta::new_readonly(config_key, false),
            message_acc,
            claim_acc,
            AccountMeta::new_readonly(endpoint, false),
            AccountMeta::new(to, false),
            if let Some(fee_r) = fee_recipient {
                AccountMeta::new(fee_r, false)
            } else {
                AccountMeta::new(to, false)
```

8. these parameters are actually smart contracts themselves. But the important thing is how these "transfer message" contracts get created.

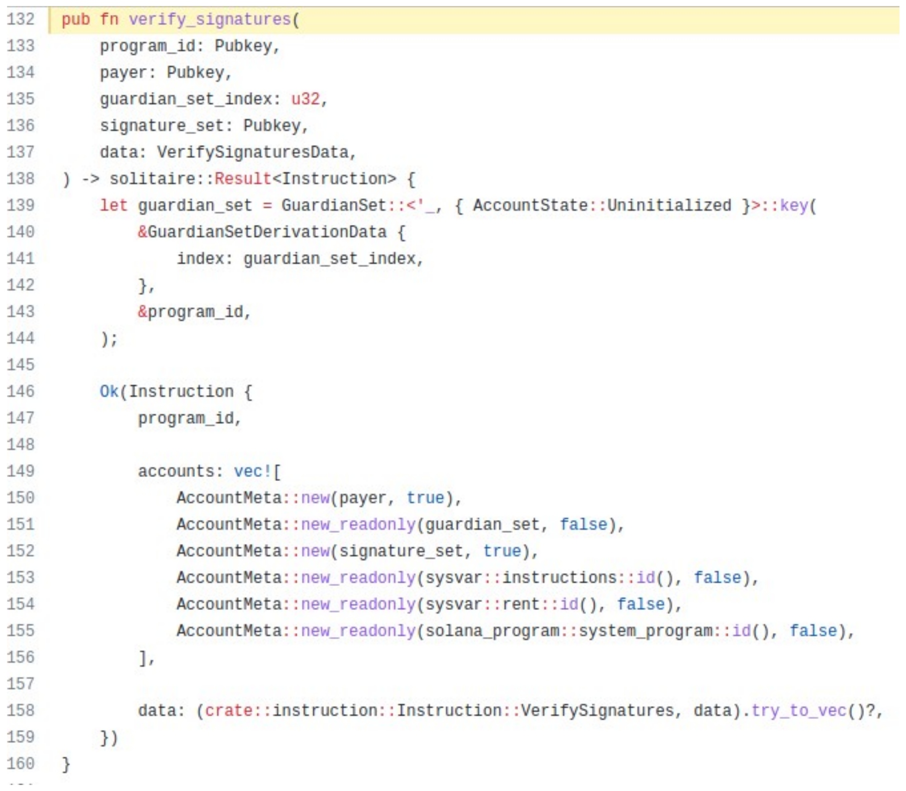
9. This "transfer message" contract is created by triggering a function called "post_vaa"



```
pub fn post_vaa(
    program_id: Pubkey,
    payer: Pubkey,
    signature_set: Pubkey,
    vaa: PostVAAData,
) -> Instruction {
    let bridge = Bridge::<'_, { AccountState::Uninitialized }>::key(None, &program_id);
    let guardian_set = GuardianSet::<'_, { AccountState::Uninitialized }>::key(
        &GuardianSetDerivationData {
            index: vaa.guardian_set_index,
        },
        &program_id,
    );
```

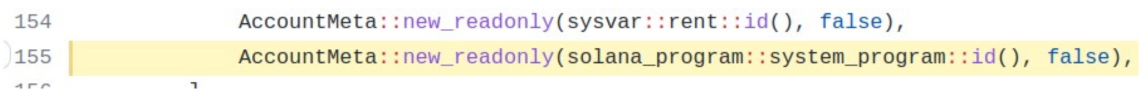
<u>The Breaking point:</u> post_vaa checks if the message is valid by checking the signatures from the guardians

- "post_vaa" doesn't actually check the signatures. Instead, in typical Solana fashion, there's another smart contract which gets created by calling the "verify_signatures" function.



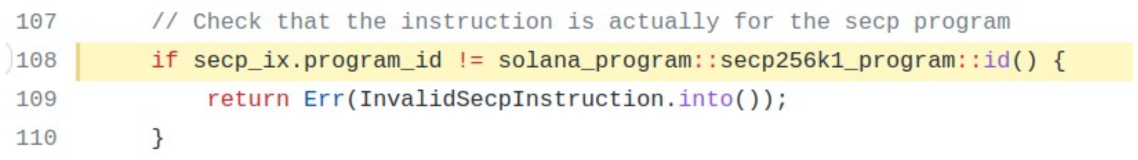
```rust
pub fn verify_signatures(
    program_id: Pubkey,
    payer: Pubkey,
    guardian_set_index: u32,
    signature_set: Pubkey,
    data: VerifySignaturesData,
) -> solitaire::Result<Instruction> {
    let guardian_set = GuardianSet::<'_, { AccountState::Uninitialized }>::key(
        &GuardianSetDerivationData {
            index: guardian_set_index,
        },
        &program_id,
    );

    Ok(Instruction {
        program_id,

        accounts: vec![
            AccountMeta::new(payer, true),
            AccountMeta::new_readonly(guardian_set, false),
            AccountMeta::new(signature_set, true),
            AccountMeta::new_readonly(sysvar::instructions::id(), false),
            AccountMeta::new_readonly(sysvar::rent::id(), false),
            AccountMeta::new_readonly(solana_program::system_program::id(), false),
        ],

        data: (crate::instruction::Instruction::VerifySignatures, data).try_to_vec()?,
    })
}
```

- One of the inputs to the "verify_signatures" function is a Solana built-in "system" program which contains various utilities the contract can use



```rust
            AccountMeta::new_readonly(sysvar::rent::id(), false),
            AccountMeta::new_readonly(solana_program::system_program::id(), false),
```

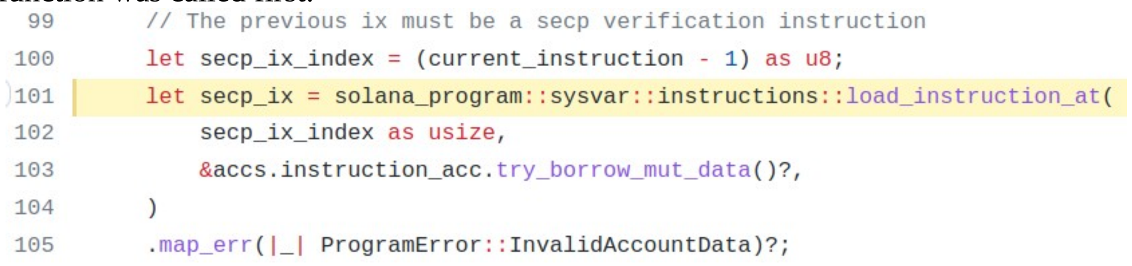
- Within "verify_signatures", the Wormhole program attempts to check that the thing that happened right before this function was triggered was that the Secp256k1 signature verification function was executed



```rust
    // Check that the instruction is actually for the secp program
    if secp_ix.program_id != solana_program::secp256k1_program::id() {
        return Err(InvalidSecpInstruction.into());
    }
```

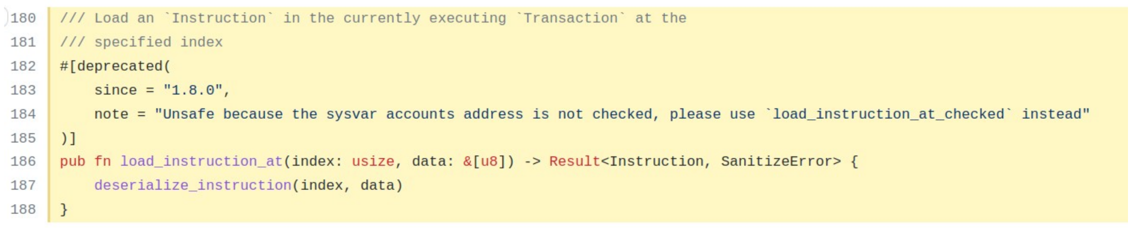
- This verification function is a built-in tool that's supposed to verify that the given signatures are correct. So the signature verification has been outsourced to this program. But here's where the bug comes in.

- The Wormhole contracts used the function load_instruction_at to check that the Secp256k1 function was called first:



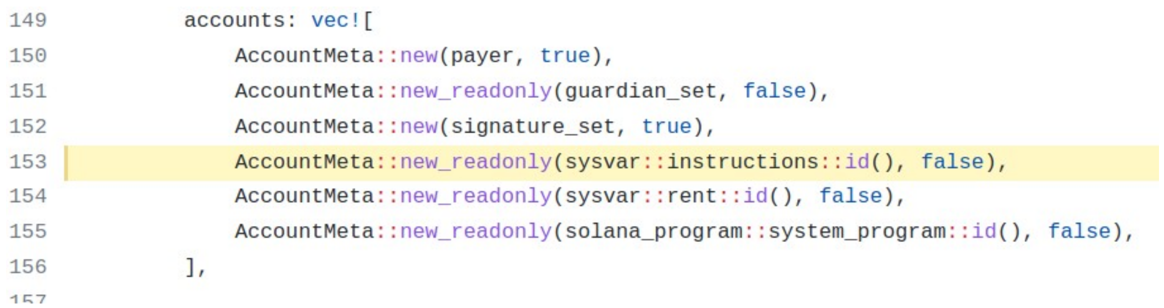
```
    // The previous ix must be a secp verification instruction
    let secp_ix_index = (current_instruction - 1) as u8;
    let secp_ix = solana_program::sysvar::instructions::load_instruction_at(
        secp_ix_index as usize,
        &accs.instruction_acc.try_borrow_mut_data()?,
    )
    .map_err(|_| ProgramError::InvalidAccountData)?;
```

- The load_instruction_at function was deprecated relatively recently because it *does not check that it's executing against the actual system address*!



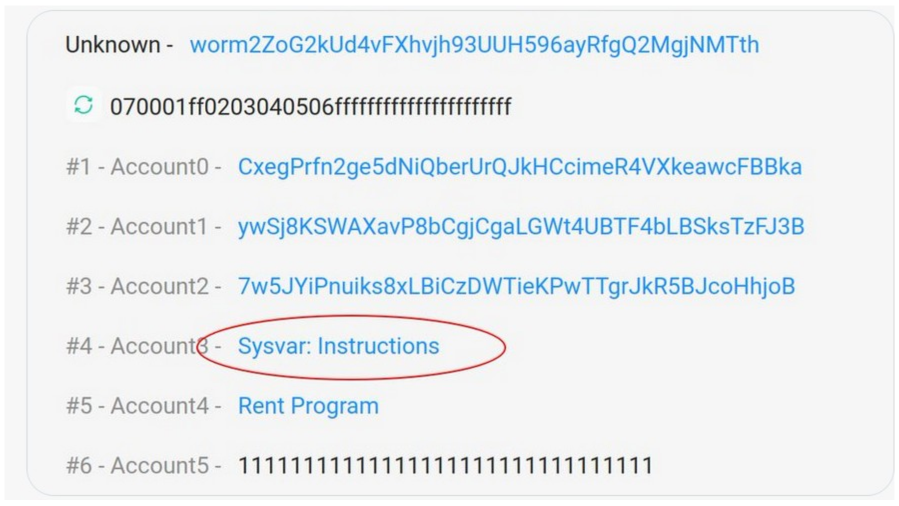
```
/// Load an `Instruction` in the currently executing `Transaction` at the
/// specified index
#[deprecated(
    since = "1.8.0",
    note = "Unsafe because the sysvar accounts address is not checked, please use `load_instruction_at_checked` instead"
)]
pub fn load_instruction_at(index: usize, data: &[u8]) -> Result<Instruction, SanitizeError> {
    deserialize_instruction(index, data)
}
```

- You're supposed to provide the system address as the program you're executing here (it's the third-to-last program input):



```
        accounts: vec![
            AccountMeta::new(payer, true),
            AccountMeta::new_readonly(guardian_set, false),
            AccountMeta::new(signature_set, true),
            AccountMeta::new_readonly(sysvar::instructions::id(), false),
            AccountMeta::new_readonly(sysvar::rent::id(), false),
            AccountMeta::new_readonly(solana_program::system_program::id(), false),
        ],
```

- Here's that system address being used as the input for the "verify_signatures" for the legit deposit of 0.1 ETH



Unknown - worm2ZoG2kUd4vFXhvjh93UUH596ayRfgQ2MgjNMTth

070001ff0203040506ffffffffffffffffffff

#1 - Account0 - CxegPrfn2ge5dNiQberUrQJkHCcimeR4VXkeawcFBBka

#2 - Account1 - ywSj8KSWAXavP8bCgjCgaLGWt4UBTF4bLBSksTzFJ3B

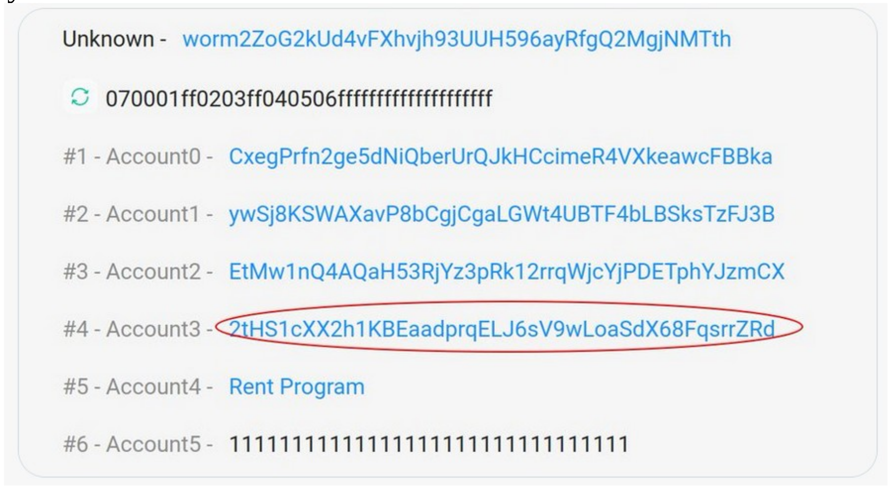
#3 - Account2 - 7w5JYiPnuiks8xLBiCzDWTieKPwTTgrJkR5BJcoHhjoB

#4 - Account3 - Sysvar: Instructions

#5 - Account4 - Rent Program

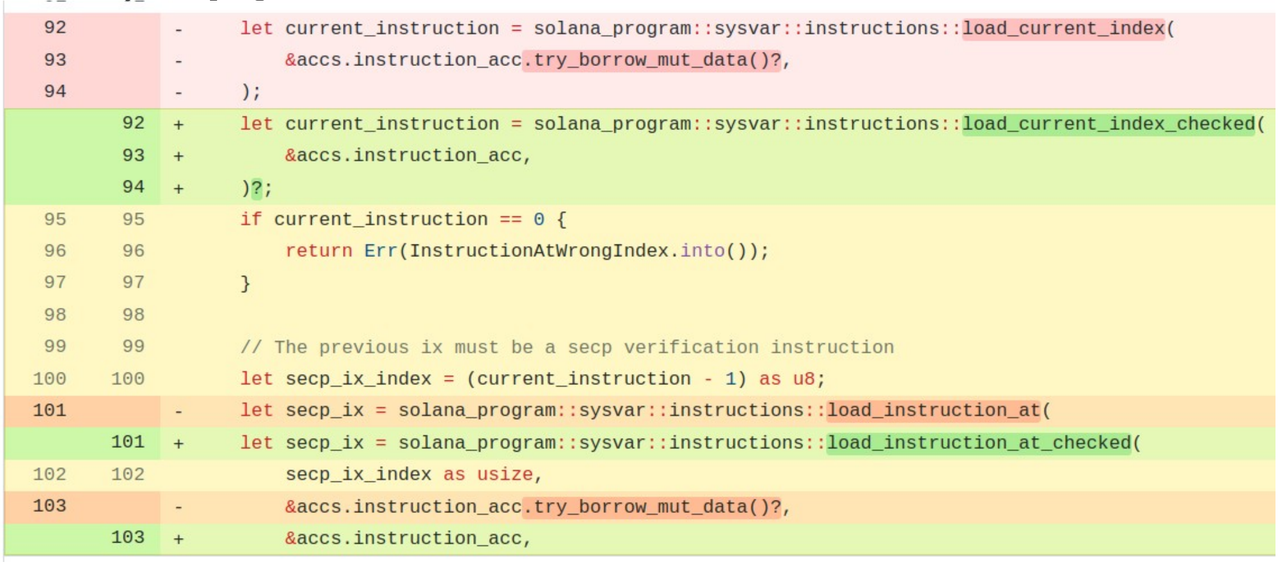
#6 - Account5 - 11111111111111111111111111111111

- But here's the "verify_signatures" transaction for the fake deposit of 120k ETH, That's not the system address!



```
Unknown - worm2ZoG2kUd4vFXhvjh93UUH596ayRfgQ2MgjNMTth
070001ff0203ff040506ffffffffffffffffff
#1 - Account0 - CxegPrfn2ge5dNiQberUrQJkHCcimeR4VXkeawcFBBka
#2 - Account1 - ywSj8KSWAXavP8bCgjCgaLGWt4UBTF4bLBSksTzFJ3B
#3 - Account2 - EtMw1nQ4AQaH53RjYz3pRk12rrqWjcYjPDETphYJzmCX
#4 - Account3 - 2tHS1cXX2h1KBEaadprqELJ6sV9wLoaSdX68FqsrrZRd
#5 - Account4 - Rent Program
#6 - Account5 - 11111111111111111111111111111111
```

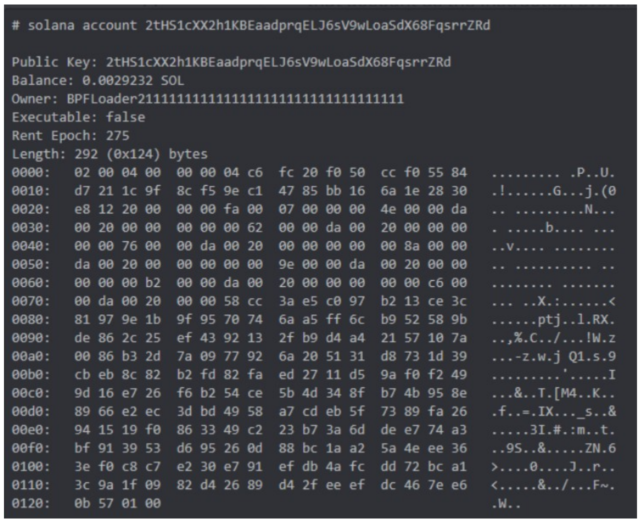
- Using this "fake" system program, the attacker could effectively lie about the fact that the signature check program was executed. The signatures weren't being checked at all!
- After that point, it was game over. The attacker made it look like the guardians had signed off on a 120k deposit into Wormhole on Solana, even though they hadn't. All the attacker needed to do now was to make their "play" money real by **withdrawing it back to Ethereum.**
- **And one withdrawal of 80k ETH + 10k ETH later (everything in the bridge on Ethereum), everything was gone.**
- **After they halted the bridge network**
- The fact that a commit was made hours before the attack replacing usage of load_instruction_at with load_instruction_at_checked, which actually confirms that the program being executed is the system program:



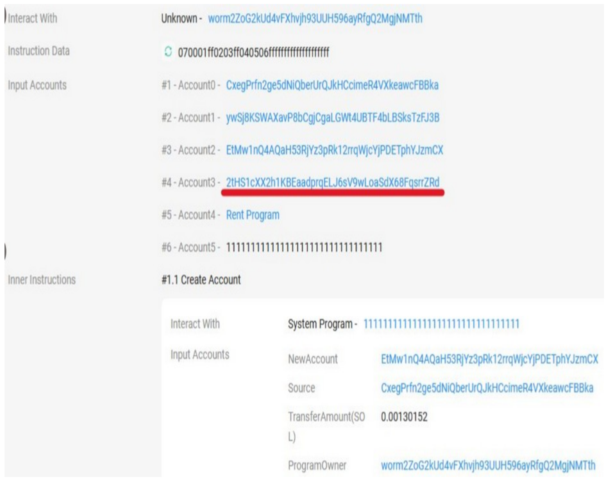
```
-     let current_instruction = solana_program::sysvar::instructions::load_current_index(
-         &accs.instruction_acc.try_borrow_mut_data()?,
-     );
+     let current_instruction = solana_program::sysvar::instructions::load_current_index_checked(
+         &accs.instruction_acc,
+     )?;
      if current_instruction == 0 {
          return Err(InstructionAtWrongIndex.into());
      }

      // The previous ix must be a secp verification instruction
      let secp_ix_index = (current_instruction - 1) as u8;
-     let secp_ix = solana_program::sysvar::instructions::load_instruction_at(
+     let secp_ix = solana_program::sysvar::instructions::load_instruction_at_checked(
          secp_ix_index as usize,
-         &accs.instruction_acc.try_borrow_mut_data()?,
+         &accs.instruction_acc,
```

- Possible that an attacker was keeping an eye on the repository and looking out for suspicious commits.
- This meant that attacker could create his own account which stored the same data that the Instructions sysvar would have stored, and substituted that account for the Instruction sysvar in the call to `verify_signatures`. This would essentially bypass signature validation entirely.

- Sure enough, that's exactly what the attacker did. Hours earlier, he created this account which contained a single serialized instruction corresponding to a call to the Secp256k1 contract. Then they passed in that account as the Instruction sysvar.



```
# solana account 2tHS1cXX2h1KBEaadprqELJ6sV9wLoaSdX68FqsrrZRd

Public Key: 2tHS1cXX2h1KBEaadprqELJ6sV9wLoaSdX68FqsrrZRd
Balance: 0.0029232 SOL
Owner: BPFLoader2111111111111111111111111111111111
Executable: false
Rent Epoch: 275
Length: 292 (0x124) bytes
0000:   02 00 04 00  00 00 04 c6  fc 20 f0 50  cc f0 55 84   ......... .P..U.
0010:   d7 21 1c 9f  8c f5 9e c1  47 85 bb 16  6a 1e 28 30   .!......G...j.(0
0020:   e8 12 20 00  00 00 fa 00  07 00 00 00  4e 00 00 da   .. .........N...
0030:   00 20 00 00  00 00 00 62  00 00 da 00  20 00 00 00   . .....b.... ...
0040:   00 00 76 00  00 da 00 20  00 00 00 00  00 8a 00 00   ..v.... ........
0050:   da 00 20 00  00 00 00 00  9e 00 00 da  00 20 00 00   .. .......... ..
0060:   00 00 00 b2  00 00 da 00  20 00 00 00  00 00 c6 00   ........ .......
0070:   00 da 00 20  00 00 58 cc  3a e5 c0 97  b2 13 ce 3c   ... ..X.:......<
0080:   81 97 9e 1b  9f 95 70 74  6a a5 ff 6c  b9 52 58 9b   ......ptj..l.RX.
0090:   de 86 2c 25  ef 43 92 13  2f b9 d4 a4  21 57 10 7a   ..,%.C../...!W.z
00a0:   00 86 b3 2d  7a 09 77 92  6a 20 51 31  d8 73 1d 39   ...-z.w.j Q1.s.9
00b0:   cb eb 8c 82  b2 fd 82 fa  ed 27 11 d5  9a f0 f2 49   .........'.....I
00c0:   9d 16 e7 26  f6 b2 54 ce  5b 4d 34 8f  b7 4b 95 8e   ...&..T.[M4..K..
00d0:   89 66 e2 ec  3d bd 49 58  a7 cd eb 5f  73 89 fa 26   .f..=.IX..._s..&
00e0:   94 15 19 f0  86 33 49 c2  23 b7 3a 6d  de e7 74 a3   .....3I.#.:m..t.
00f0:   bf 91 39 53  d6 95 26 0d  88 bc 1a a2  5a 4e ee 36   ..9S..&.....ZN.6
0100:   3e f0 c8 c7  e2 30 e7 91  ef db 4a fc  dd 72 bc a1   >....0....J..r..
0110:   3c 9a 1f 09  82 d4 26 89  d4 2f ee ef  dc 46 7e e6   <.....&../...F~.
0120:   0b 57 01 00                                          .W..
```



Interact With	Unknown - worm2ZoG2kUd4vFXhvjh93UUH596ayRfgQ2MgjNMTth
Instruction Data	070001ff0203ff040506ffffffffffffffffffff
Input Accounts	#1 - Account0 - CxegPrfn2ge5dNiQberUrQJkHCcimeR4VXkeawcFBBka
	#2 - Account1 - ywSj8KSWAXavP8bCgjCgaLGWt4UBTF4bLBSksTzFJ3B
	#3 - Account2 - EtMw1nQ4AQaH53RjYz3pRk12rrqWjcYjPDETphYJzmCX
	#4 - Account3 - 2tHS1cXX2h1KBEaadprqELJ6sV9wLoaSdX68FqsrrZRd
	#5 - Account4 - Rent Program
	#6 - Account5 - 11111111111111111111111111111111
Inner Instructions	#1.1 Create Account

Interact With	System Program - 11111111111111111111111111111111	
Input Accounts	NewAccount	EtMw1nQ4AQaH53RjYz3pRk12rrqWjcYjPDETphYJzmCX
	Source	CxegPrfn2ge5dNiQberUrQJkHCcimeR4VXkeawcFBBka
	TransferAmount(SOL)	0.00130152
	ProgramOwner	worm2ZoG2kUd4vFXhvjh93UUH596ayRfgQ2MgjNMTth

Briefly explain key mechanisms you will put in place in your interoperable private bridge (specifically the withdrawal methods) to prevent a similar attack?

➔ I would implement MMRroots to transactions called autonomously by contracts (not all), a simple field to be included in leaves compared against sender MMRroot to make sure that only the legitmet withdrawal function called and not interfering with other verifaction functions, and called by the sender's address.

➔ I can think of (On smart contract level: Rust), Implementing an asyn function verify_signatures() to compile to a block of op-codes waiting for the post_vaa() function to check that the thing that happened right before signature verification function was executed.

3. **[Bonus]** Design a smart contract to be deployed on both blockchain to take care of the user's request and circuits to take care of the privacy

Answer 1.3:

Link: **Bonus-Passed**

Q.2 Aztec

AZTEC protocol utilizes a set of zero-knowledge proofs to define a confidential transaction protocol, to shield both native assets and assets that conform with certain standards (e.g. ERC20) on a Turing-complete general-purpose computation.

1. Briefly explain the concept of AZTEC Note and how the notes are used to privately represent various assets on a blockchain. There are various proofs utilized by the AZTEC protocol including range proofs, swap proofs, dividend proofs and join-split proofs. Highlight the features of these proofs and the roles they play in creating confidential transactions on the blockchain

Answer 2.1:

The concept of AZTEC Note:

An AZTEC note is an encrypted representation of abstract value. How this abstract representation maps to real quantities The 'note' is an output of the AZTEC commitment function

The AZTEC commitment function $\mathsf{com} : [1; k_{\mathsf{max}}] \times \mathbb{Z}_p^* \to \mathbb{G} \times \mathbb{G}$ is defined from the setup as

$$\mathsf{com}(k; a) := (\mu_k^a, \mu_k^{ka} h^a).$$

When a sender desires to commit to a value k ∈ [1; kmax] she picks at random a ← Zp and computes a commitment (γ, σ) = com(k; a).

It is comprised of a tuple of elliptic curve commitments and three scalars: a **viewing key, a spending key and a message.** Knowledge of the viewing key allows the note to be decrypted, revealing the message. Knowledge of the viewing key can be used to create valid join-split zero-knowledge proofs. These proofs are then signed by the spending key.

Who owns the note:

When an AZTEC note is issued, a public key, Q, is defined by the transaction issuer. The witness to Q is referred to as the 'spending key'. If this note is used as an input to a future join-split transaction, the prover must provide a signature signed by the spending key. The message of this signature is a hash of the input string of the join-split transaction involving the note. Ownership of a note is defined as possessing knowledge of the witness to Q

How the notes are used to privately represent various assets on a blockchain?

The 'note' is an output of the AZTEC commitment function, It is comprised of a tuple of elliptic curve commitments and three scalars: a viewing key, a spending key and a message.

Knowledge of the viewing key allows the note to be decrypted, revealing the message. Knowledge of the viewing key can be used to create valid join-split zero-knowledge proofs. These proofs are then signed by the spending key.

a ‘traditional’ digital asset refers to a program em-bedded into a blockchain protocol that defines the ownership record and transfer logic for a digital representation of value, . The AZTEC protocol can be used to interact with traditional digital assets that utilize the same underlying blockchain as the AZTEC protocol.

The protocol can be used to define two different types of digital assets: fully anonymous assets and public/private assets.

Fully anonymous assets are represented exclusively via optimized AZTEC notes and do not have a traditional digital asset representation. The only methods of trading a fully anonymous asset is via join-split transactions or via the AZTEC decentralized exchange protocol.

The reference implementation of the AZTEC protocol is tailored to minimize the gas costs required to validate an AZTEC join-split transaction through a smart contract defined by the Ethereum protocol’s Ethereum Virtual Machine.

1. the **note’s spending key Q** is defined over the secp256k1 [16] curve, as opposed to the bn128 curve that is used for the AZTEC zero-knowledge protocols. The gas cost to validate an ecdsa signature on the secp256k1 curve is 3,000 gas.
2. Using the **secp256k1 curve enables the protocol to define ‘ownership’** using the same elliptic curve that the base Ethereum protocol uses, allowing for the potential re-use of Ethereum addresses.
3. **Q is a stealth address** that is generated via a modified form of the dual-key stealth address protocol.
4. A stealth address ‘wallet’ contains two public/private key pairs, **a scan key (V, v), V = g′v** , and an **issue key (S, s), s = g′s**
5. When constructing an **AZTEC note**, the transaction sender generates an ephemeral key pair (B, b), B = g′b. The sender then constructs a Diffie-Hellman **shared secret between B and V , x = H(g′vb) = H(V b).**
6. The note’s spending key, Q, is defined as **Q = S · g′x.** In addition, for every note the ephemeral key B is also published.
7. The **viewing key a**, required to decrypt an AZTEC note, is generated by **taking x mod p, where p is the order of the bn128 group g.**

The various proofs utilized by the AZTEC protocol?

Note: EVM Blockchains protocols are blocks of opcodes runs on distributed machines organized with syntax & semantics after the consensus bottleneck, so dividend proofs, swap proofs and join-split proofs relies on the underlying zero-knowledge primitives and inherits it' security assumption.

The Anonymous Zero-knowledge Transactions with Efficient Communication (AZTEC) protocol describes a set of zero-knowledge proofs that define a confidential transaction protocol, designed for use within blockchain protocols that support Turing-complete general-purpose computation, AZTEC zero-knowledge proofs cost approximately 840,000 'gas' to verify on the Ethereum main-net.

The protocol is utilizes a commitment scheme that enables the efficient verifica-
tion of range proofs. This is combined with a set of zero-knowledge Sigma protocols to enable efficiently verifiable confidential transactions.

Range proofs:

The AZTEC protocol validates the legitimacy of a join-split transaction using a combination of homomorphic arithmetic and range proofs.

The range proof requires 3 elliptic curve point scalar multiplications and 1 bilinear pairing comparison to verify.

Multiple range proofs can be combined, in which case verifying the set of proofs requires 4 elliptic curve point scalar multiplications per proof and 1 bilinear pairing comparison for the complete set of proofs.

EVM verifier effeciency:

@https://raw.githubusercontent.com/AztecProtocol/AZTEC/master/AZTEC.pdf#section.8

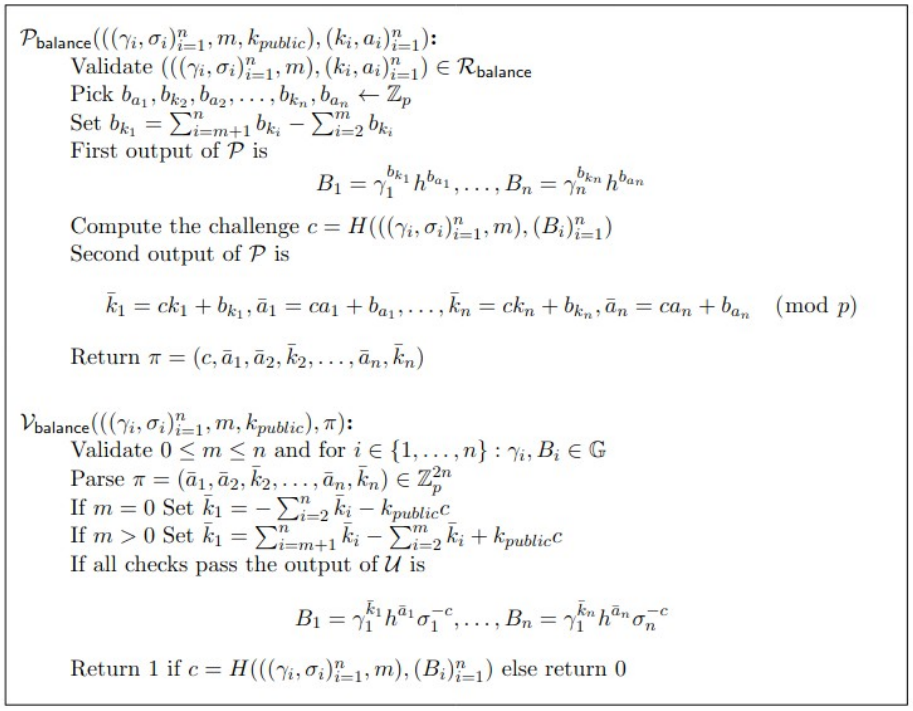
Note: This efficiency comes at the cost of a trusted setup phase that generates a common reference string that contains a set of elliptic curve points.

<u>Sigma-protocols:</u>

1) Non-interactive zero-knowledge proof

- P

s



```
$\mathcal{P}_{\mathsf{balance}}(((\gamma_i,\sigma_i)_{i=1}^n, m, k_{public}), (k_i,a_i)_{i=1}^n)$:
    Validate $(((\gamma_i,\sigma_i)_{i=1}^n, m), (k_i,a_i)_{i=1}^n) \in \mathcal{R}_{\mathsf{balance}}$
    Pick $b_{a_1}, b_{k_2}, b_{a_2}, \dots, b_{k_n}, b_{a_n} \leftarrow \mathbb{Z}_p$
    Set $b_{k_1} = \sum_{i=m+1}^n b_{k_i} - \sum_{i=2}^m b_{k_i}$
    First output of $\mathcal{P}$ is
        $B_1 = \gamma_1^{b_{k_1}} h^{b_{a_1}}, \dots, B_n = \gamma_n^{b_{k_n}} h^{b_{a_n}}$
    Compute the challenge $c = H(((\gamma_i,\sigma_i)_{i=1}^n, m), (B_i)_{i=1}^n)$
    Second output of $\mathcal{P}$ is
        $\bar{k}_1 = ck_1 + b_{k_1}, \bar{a}_1 = ca_1 + b_{a_1}, \dots, \bar{k}_n = ck_n + b_{k_n}, \bar{a}_n = ca_n + b_{a_n} \pmod p$
    Return $\pi = (c, \bar{a}_1, \bar{a}_2, \bar{k}_2, \dots, \bar{a}_n, \bar{k}_n)$

$\mathcal{V}_{\mathsf{balance}}(((\gamma_i,\sigma_i)_{i=1}^n, m, k_{public}), \pi)$:
    Validate $0 \le m \le n$ and for $i \in \{1,\dots,n\} : \gamma_i, B_i \in \mathbb{G}$
    Parse $\pi = (\bar{a}_1, \bar{a}_2, \bar{k}_2, \dots, \bar{a}_n, \bar{k}_n) \in \mathbb{Z}_p^{2n}$
    If $m = 0$ Set $\bar{k}_1 = -\sum_{i=2}^n \bar{k}_i - k_{public}c$
    If $m > 0$ Set $\bar{k}_1 = \sum_{i=m+1}^n \bar{k}_i - \sum_{i=2}^m \bar{k}_i + k_{public}c$
    If all checks pass the output of $\mathcal{U}$ is
        $B_1 = \gamma_1^{\bar{k}_1} h^{\bar{a}_1} \sigma_1^{-c}, \dots, B_n = \gamma_1^{\bar{k}_n} h^{\bar{a}_n} \sigma_n^{-c}$
    Return 1 if $c = H(((\gamma_i,\sigma_i)_{i=1}^n, m), (B_i)_{i=1}^n)$ else return 0
```

Figure 2: Algorithms $\mathcal{P}_{\mathsf{balance}}$, $\mathcal{V}_{\mathsf{balance}}$

- Proof of perfect completeness

$$B_i = \gamma_i^{\bar{k}_i} h^{\bar{a}_i} \sigma_i^{-c} = \gamma_i^{ck_i + b_{k_i}} h^{ca_i + b_{a_i}} (\gamma_i^{k_i} h^{a_i})^{-c} = \gamma_i^{b_{k_i}} h^{b_{a_i}}.$$

- Proof of special soundness

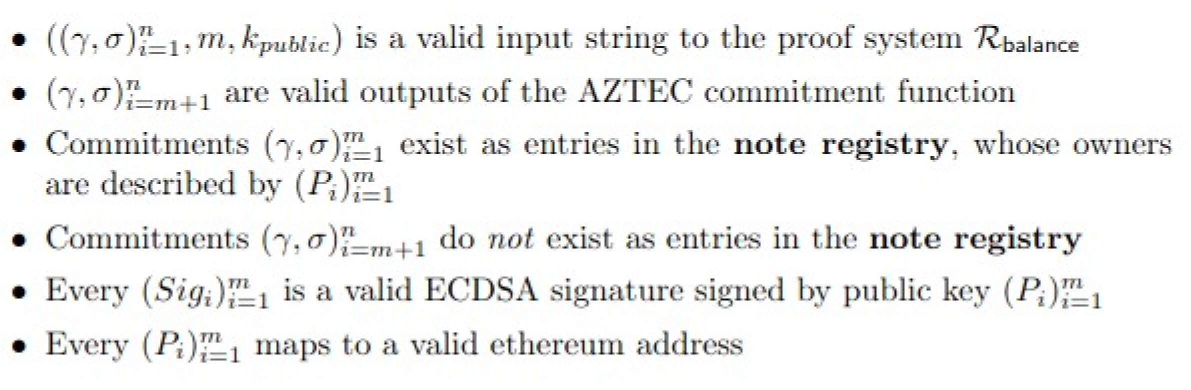
$$\frac{\bar{k}_1' - \bar{k}_1}{c' - c} = \sum_{i=m+1}^{n} \frac{\bar{k}_i' - \bar{k}_i}{c' - c} - \sum_{i=2}^{m} \frac{\bar{k}_i' - \bar{k}_i}{c' - c} + k_{public} = \sum_{i=m+1}^{n} k_i - \sum_{i=2}^{m} k_i + k_{public} = k_1$$

- Proof of special honest verifier zero-knowledge

$$k_i = \frac{\bar{k}_i - b_{k_i}}{c} \quad a_i = \frac{\bar{a}_i - b_{a_i}}{c}.$$

- Join – Split protocol:

  A valid ‘join-split’ transaction proves the following properties about these commitments:



- $((\gamma,\sigma)_{i=1}^{n}, m, k_{public})$ is a valid input string to the proof system $\mathcal{R}_{\text{balance}}$
- $(\gamma,\sigma)_{i=m+1}^{n}$ are valid outputs of the AZTEC commitment function
- Commitments $(\gamma,\sigma)_{i=1}^{m}$ exist as entries in the **note registry**, whose owners are described by $(P_i)_{i=1}^{m}$
- Commitments $(\gamma,\sigma)_{i=m+1}^{n}$ do *not* exist as entries in the **note registry**
- Every $(Sig_i)_{i=1}^{m}$ is a valid ECDSA signature signed by public key $(P_i)_{i=1}^{m}$
- Every $(P_i)_{i=1}^{m}$ maps to a valid ethereum address

2. ***[Infrastructure Track Only]*** Using the loan application as a reference point, briefly explain how AZTEC can be used to create a private loan application on the blockchain highlighting the benefits and challenges. In the Loan Application, explain the Loan.sol and LoanDapp.sol file (comment inline)

Answer 2.2:

Link: https://github.com/amrosaeed/zku/blob/main/Assignment5_Q2/Loan.sol
Link: https://github.com/amrosaeed/zku/blob/main/Assignment5_Q2/LoanDapp.sol

Q3. Webb

Webb protocol is tornado cash with a bridge built on top of it. EVM code**,** relayer code, substrate code (in development). Webb is not live yet, it's only on testnet. Code is not yet complete so be aware of that while reading it.

1. What is the difference between commitments made to the Anchor and VAnchor contracts? Can you think of a new commitment structure that could allow for a new feature? (eg: depositing one token and withdrawing another?) if so, what would the commitment look like?

What is the difference between commitments made to the Anchor and VAnchor contracts?

Answer 3.1:

Anchor contracts: @ https://github.com/webb-tools/protocol-solidity/blob/main/contracts/anchors FixedDepositAnchor.sol

The FixedDepositAnchor system is an inter-operable shielded pool supporting fixed denomination deposits of ERC20 tokens.

> The system is built on top the AnchorBase/LinkableTree system which allows it to be linked to other FixedDepositAnchors through a simple graph-like interface where anchors maintain edges of their neighboring anchors.
> The system requires users to both deposit a fixed denomination of ERC20 assets into the smart contract and insert a commitment into the underlying merkle tree of the form:
>
> **commitment = Poseidon(destinationChainId, nullifier, secret).**

Information regarding the commitments:

- Poseidon is a zkSNARK friendly hash function.
- destinationChainId is the chainId of the destination chain, where the withdrawal is intended to be made.
- nullifier is a random field element and identifier for the deposit that will be used to withdraw the deposit and ensure that the deposit is not double withdrawn.
- secret is a random field element that will remain secret throughout the lifetime of the deposit and withdrawal.

Using the **preimage of the commitment**, users can generate a zkSNARK proof that the deposit is located in one-of-many anchor merkle trees and that the commitment's destination chain id matches the underlying chain id of the anchor where the withdrawal is taking place. The chain id opcode is leveraged to prevent any tampering of this data.

Vanchor contracts: @ https://github.com/webb-tools/protocol-solidity/blob/main/contracts/vanchors/VAnchor.sol

The Variable Anchor is a variable-denominated shielded pool system derived from Tornado Nova (tornado-pool). This system extends the shielded pool system into a bridged system and **allows for join/split transactions.**

> The system requires users to create and deposit UTXOs for the supported ERC20 asset into the smart contract and insert a commitment into the underlying merkle tree of the form:
>
> **commitment = Poseidon(chainID, amount, pubKey, blinding).**
>
> The hash input is the UTXO data. All deposits/withdrawals are unified under a common `transact` function which requires a zkSNARK proof that the UTXO commitments are well-formed (i.e. that the deposit amount matches the sum of new UTXOs' amounts).

Information regarding the commitments:

- Poseidon is a zkSNARK friendly hash function.
- destinationChainID is the chainId of the destination chain, where the withdrawal is intended to be made.
- Details of the UTXO and hashes are below.

UTXO = { destinationChainID, amount, pubkey, blinding }
commitment = Poseidon(destinationChainID, amount, pubKey, blinding)
nullifier = Poseidon(commitment, merklePath, sign(privKey, commitment, merklePath))

Using the **preimage / UTXO of the commitment**, users can generate a zkSNARK proof that the UTXO is located in one-of-many VAnchor merkle trees and that the commitment's destination chain id matches the underlying chain id of the VAnchor where the transaction is taking place. The chain id opcode is leveraged to prevent any tampering of this data.

2. [Optional] Compare commitments from Anchor and Tornado Cash too

Answer 3.2: Optional-Passed

3. Describe how the UTXO scheme works on the VAnchor contract.

Answer 3.3:

Details of the UTXO scheme on the VAnchor contract:

nullifier = Poseidon(commitment, merklePath, sign(privKey, commitment, merklePath))

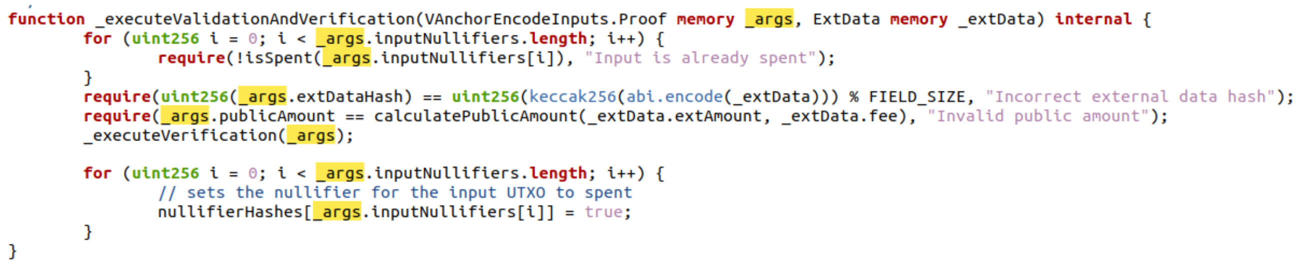
nullifier algo value to the output of Posiden hash function that takes commitmet, merklePath and a signature function on sender's privKey, commitment and merklePath AS an INPUT.

commitment = Poseidon(destinationChainID, amount, pubKey, blinding)

commitment algo value to the output of Posiden hash function that takes destinationChainID, amount sender's pubKey and blinding AS an INPUT

Using the **preimage / UTXO of the commitment**, users can generate a zkSNARK proof that the UTXO is located in one-of-many VAnchor merkle trees and that the commitment's destination chain id matches the underlying chain id of the VAnchor where the transaction is taking place.

By Using "for loops" for looping through mapping "length attribute" of ZkSNARK arguments (_args) **TO "nullifierHashes".**



```
function _executeValidationAndVerification(VAnchorEncodeInputs.Proof memory _args, ExtData memory _extData) internal {
        for (uint256 i = 0; i < _args.inputNullifiers.length; i++) {
                require(!isSpent(_args.inputNullifiers[i]), "Input is already spent");
        }
        require(uint256(_args.extDataHash) == uint256(keccak256(abi.encode(_extData))) % FIELD_SIZE, "Incorrect external data hash");
        require(_args.publicAmount == calculatePublicAmount(_extData.extAmount, _extData.fee), "Invalid public amount");
        _executeVerification(_args);

        for (uint256 i = 0; i < _args.inputNullifiers.length; i++) {
                // sets the nullifier for the input UTXO to spent
                nullifierHashes[_args.inputNullifiers[i]] = true;
        }
}
```

@ Function _executeValidationAndVerification(), checking that _args.inputNullifiers.length attribute through mapping by using a for loop until the value at index (I) length passed as parameter for nullifiersHashes targets **TRUE**

The Function uses at the top, the for loop mechanism for checking whether the index (I) of the nullifierHashes is already spent or not.

And checks _args.extDataHash & _args.publicAmount attributes to check the external data and amount.

4. **[Infrastructure Track Only] Explain how the relayer works for the deposit part of the** tornado contract

Answer 3.4:

Link: https://github.com/amrosaeed/zku/blob/02aba533713457e70544189e6992eb9d5f3316c3/Assignment5_Q3/tornado_leaves_watcher.rs#L83

5. **[Bonus]** Write a program using ethers-rs to interact with the dark-forest smart contract you created for assignment 3.

Answer 3.5:

Link: Bonus-Passed

Q4. Thinking In ZK

1. If you have a chance to meet with the people who built the above protocols what questions would you ask them?

Answer 4.1:

For AZTEC protocol:

1- I would like to whether the security assumption under Aztec's SNARKs initial setups is hardened enough under attacks, or still tested by academia, as I am pro using SNARKs as a scaling approach for privacy instead of STARKs ?

For WEBB protocol:

1- I may ask about their UTXO scheme; a hypothetical question, have they ever considered imitating AZTEC's UTXO/commitment scheme 'Notes' ? Or it wont work?, also I would ask about weather they ever considered STARKs?

(Just interested in arguments made by zk tech pioneers about market ready-ability for STARKs vs SNARKs)