well, in another chat you had been helping me develop ideas for a paper I am working on - it employs ZKPs with one-time passwords for authentication in a V2I environment. OTP provides session-specific replay protection. Blockchain (BC) logs transactions to prevent tampering in auditability of authentication events. OTPs are generated dynamically for each authentication session and derived using a cryptographic hash function seeded with a shared secret & a timestamp. The steps: 1) vehicle generates an OTP; 2) OTP is embedded into a ZKP to prove the OTP is valid; 3) Proof is sent to a road side unit (RSU), which verifies the ZKP by using a smart contract. The blockchain logs vehicle identities (anonymized through the ZKPs), along with session details.

Excellent — I remember this topic, and it’s a very compelling integration of **ZKPs**, **OTPs**, and **blockchain** for secure, privacy-preserving V2I authentication. Now that you're revisiting this, would you like assistance with:

* **Refining the protocol steps or flowchart**?
* **Articulating the security guarantees** (e.g., replay protection, unlinkability)?
* **Comparing your approach to existing authentication frameworks** (e.g., EAP, TLS-based V2I systems)?
* **Explaining smart contract responsibilities** in the system?
* **Highlighting the novelty/contribution** in your paper?

Let me know what you’d like to focus on next — I can help draft sections, prepare diagrams, or assist with positioning your idea in the literature.

##### You said:

I think we've already done most of that, although I would appreciate help refining the protocol steps and flowchart, as well as articulating the security guarantees

Great — let’s start with a refined and clearly structured version of the **protocol steps**, followed by an outline for a **flowchart**, and then a set of **security guarantees** explained in concise, formal language.

## 🔐 Refined Protocol Steps: ZKP + OTP + Blockchain Logging

**Participants**:

* **Vehicle (V)**
* **Roadside Unit (RSU)**
* **Blockchain Network (with smart contract)**

### ****Shared Primitives****:

* h(): cryptographic hash function (e.g., SHA-256)
* K: shared secret between V and RSU
* T: current timestamp (synchronized or rounded to time window)
* OTP = h(K || T)
* ZKP = Prove(OTP is valid under K and T, without revealing K or OTP)
* SC: smart contract deployed on blockchain

### ****Protocol Steps****:

1. **OTP Generation by Vehicle**
   * V computes OTP = h(K || T) for the current session.
   * This OTP is **fresh**, session-specific, and unguessable without K.
2. **ZKP Construction**
   * V constructs a zero-knowledge proof π proving:

"I know an OTP such that OTP = h(K || T) and this OTP is correct for time T."

* + The proof reveals neither K nor OTP.

1. **ZKP Submission to RSU**
   * V sends (π, T, session metadata) to RSU over secure or short-range V2I link.
2. **Verification and Smart Contract Interaction**
   * RSU:
     + Validates π using public parameters and smart contract interface.
     + Sends (π, T, anonymized V-ID, session metadata) to smart contract SC.
3. **Smart Contract Audit Logging**
   * SC:
     + Verifies the ZKP off-chain or via on-chain verifier (e.g., ZoKrates or Solidity verifier).
     + Logs: [V-ID\*, T, session ID, success/failure, hash of proof] to blockchain.
     + V-ID\* is anonymized or hashed to preserve privacy.

## 🧭 Flowchart Structure (Text Description)

You can use this to build a diagram later:

[Vehicle]

|

|--1. Generate OTP: h(K || T)

|--2. Construct ZKP π proving OTP is valid

|

v

[RSU]

|

|--3. Receive (π, T)

|--4. Verify π

|--5. Send data to smart contract

|

v

[Smart Contract]

|

|--6. Verify proof (on- or off-chain)

|--7. Log anonymized session to blockchain

|

v

[Blockchain]

--> Immutable, auditable log of successful authentications

## 🔒 Security Guarantees

| Guarantee | How It’s Achieved |
| --- | --- |
| **Session-specific authentication** | OTP is derived using time T, so it changes every session |
| **Replay protection** | Old OTPs are invalid for new T; smart contract can reject duplicates or replays |
| **Anonymity** | ZKP reveals no secret values; vehicle identity can be hashed or blinded |
| **Unlinkability** | Each session uses a different OTP and proof — no link across sessions |
| **Tamper-proof audit log** | Authentication events are recorded on blockchain, ensuring integrity |
| **Verifier integrity** | Smart contract logic is deterministic and public |
| **Zero-knowledge soundness** | A false proof cannot be forged without knowledge of correct OTP |

Would you like:

* A LaTeX-friendly version of the protocol?
* A visual version of the flowchart?
* Help choosing or citing ZKP frameworks (e.g., Groth16, PLONK, Halo2, etc.) for your implementation?

Actually, I’d like all three of these

### 📌 ****1. Flowchart (ZKP + OTP + Blockchain Logging)****

### 📄 ****2. LaTeX-Friendly Protocol Steps (Pseudocode Format)****

\begin{enumerate}

\item \textbf{OTP Generation (Vehicle)}: Compute $\mathsf{OTP} = h(K \parallel T)$

\item \textbf{ZKP Construction}: Create zero-knowledge proof $\pi$ such that:

\[

\pi : \text{Prove that } \exists K, T \text{ such that } \mathsf{OTP} = h(K \parallel T)

\]

\item \textbf{Proof Submission}: Vehicle sends $(\pi, T, \text{session metadata})$ to RSU

\item \textbf{Verification (RSU)}:

\begin{itemize}

\item Verify $\pi$ using public parameters

\item Submit $(\pi, T, \text{anonymized V-ID}, \text{session info})$ to smart contract

\end{itemize}

\item \textbf{Smart Contract Execution}:

\begin{itemize}

\item Validate $\pi$ (on- or off-chain)

\item Log session data to blockchain: $[\text{V-ID}^\*, T, \text{Session ID}, \text{Status}]$

\end{itemize}

\end{enumerate}

### 🧰 ****3. ZKP Frameworks to Consider****

| Framework | Description | Suitable For |
| --- | --- | --- |
| **ZoKrates** | High-level DSL for ZKPs over Ethereum, generates Solidity verifiers | Easy integration with EVM |
| **snarkjs** | JavaScript toolkit for zk-SNARKs (Groth16), works with Circom circuits | Web-focused deployments |
| **Halo2** | Recursive, efficient ZKPs by Electric Coin Company (Zcash) | Advanced, flexible, no trusted setup |
| **PLONK** | Universal setup, efficient prover/verifier, supported by many platforms | High performance, low overhead |
| **Risc0** | ZKVM for general-purpose code execution in ZKP form | For running full programs as ZKPs |

✅ For your system, **ZoKrates** is ideal if you're using Ethereum smart contracts. If you're targeting a broader or non-EVM platform, **PLONK** or **Halo2** are strong contenders.

Let me know if you'd like the LaTeX formatted into a standalone document, help generating sample ZoKrates circuits for your OTP logic, or how to benchmark ZKP verifier gas costs on-chain.

ok, let's go back to the topics in my paper. In the simulation, a vehicle generates an OTP and submits a ZKP to the nearest RSU. Can you assist in possible code for the OTP and the ZKP?

Absolutely — we can walk through sample code for both components:

## 🔐 ****1. OTP Generation (One-Time Password)****

We’ll use Python with the hashlib library to generate a time-based OTP seeded with a shared secret K.

### ✅ OTP Generation Code (Python)

import hashlib

import time

def generate\_otp(shared\_secret: str, time\_window: int = 30) -> str:

# Round current time to nearest time window (e.g., 30 seconds)

current\_time = int(time.time() // time\_window)

input\_str = f"{shared\_secret}|{current\_time}"

# Hash the input to create OTP

otp = hashlib.sha256(input\_str.encode()).hexdigest()

# Optionally truncate (e.g., to 6 digits or 16 hex chars)

return otp[:16]

# Example usage

K = "shared\_secret\_key"

otp = generate\_otp(K)

print("Generated OTP:", otp)

You could also use HMAC if you're following something like TOTP (RFC 6238), but for simulation and integration into ZKPs, this hash-based version suffices.

## 🧪 ****2. ZKP Construction****

Here’s an example using **ZoKrates**, which is great for embedding proofs in a smart contract later. The following circuit checks that an OTP was correctly derived using SHA256 and a known time window.

### ✅ ZoKrates Circuit (High-Level)

import "hashes/sha256/sha256" as sha256

def main(private field secret\_key, field time\_slot, field[2] expected\_hash) -> bool {

field[2] otp\_hash = sha256([secret\_key, time\_slot])

return otp\_hash == expected\_hash

}

### 🛠️ ZoKrates Workflow (Command Line)

# Compile the circuit

zokrates compile -i otp\_proof.zok

# Setup trusted setup (only needed for Groth16)

zokrates setup

# Compute witness (example values)

zokrates compute-witness -a 1234567890 1688000 <expected\_hash>

# Generate proof

zokrates generate-proof

# Export verifier smart contract (for Solidity)

zokrates export-verifier

You'd write a small wrapper to compute expected\_hash = sha256(K || T) in Python and then pass it into the ZoKrates circuit.

## 🧩 Integration Strategy

1. Vehicle:
   * Computes OTP via h(K || T)
   * Generates ZKP that proves knowledge of secret K matching the OTP
2. RSU:
   * Verifies proof using on-chain smart contract or precompiled verifier
   * Accepts authentication if proof is valid and T is recent

Would you like help creating:

* A **full ZoKrates input/output test case**?
* The **Solidity verifier** for use with the smart contract?
* A Python-to-ZoKrates converter for OTP generation?

Let me know how deep you want to go!

yes - the RSU is supposed to use a smart contract to verify the proof, so I'd like help creating that. Additionally, I'd like the python-to-zokrates converter for OTP generation

## ✅ Part 1: ****Smart Contract Verifier (ZoKrates → Solidity)****

Once you’ve written and compiled your ZoKrates circuit (e.g., proving that OTP = sha256(K || T)), ZoKrates can generate a Solidity verifier contract.

### 🧰 Steps:

1. **Export Verifier from ZoKrates**:

bash:

zokrates export-verifier

This produces a file like verifier.sol.

1. **Smart Contract Usage Example**:

Here’s how you can integrate it:

solidity:

pragma solidity ^0.8.0;

import "./verifier.sol"; // This is the ZoKrates-generated contract

contract OTPAuth {

Verifier verifier;

constructor(address \_verifier) {

verifier = Verifier(\_verifier);

}

event AuthenticationLogged(bytes32 anonymizedID, uint256 timestamp, bool success);

function authenticate(

bytes32 anonymizedID,

uint256 timestamp,

Verifier.Proof memory proof

) public {

bool valid = verifier.verifyTx(proof);

emit AuthenticationLogged(anonymizedID, timestamp, valid);

require(valid, "Invalid proof.");

// Additional on-chain logic here (e.g., logging, session handling)

}

}

You’ll pass Verifier.Proof as an object with a, b, c, and input arrays (ZoKrates formats this for you in the output).

## ✅ Part 2: ****Python-to-ZoKrates Converter for OTP Hash Input****

Here’s Python code to generate a SHA-256 OTP and convert it into ZoKrates-friendly field elements:

### 🐍 Python Code

import hashlib

import time

def hash\_to\_field\_elements(input\_str):

# SHA256 hash

digest = hashlib.sha256(input\_str.encode()).hexdigest()

# Split into two 128-bit parts for ZoKrates field[2]

first = int(digest[:32], 16)

second = int(digest[32:], 16)

return [first, second]

def generate\_zokrates\_inputs(secret\_key, time\_window=30):

current\_time = int(time.time() // time\_window)

concat\_input = f"{secret\_key}|{current\_time}"

field\_elements = hash\_to\_field\_elements(concat\_input)

return current\_time, field\_elements

# Example

K = "shared\_secret\_key"

T, expected\_hash = generate\_zokrates\_inputs(K)

print("Time window:", T)

print("ZoKrates input hash:", expected\_hash) # field[2] format

These expected\_hash values can now be passed into ZoKrates as field[2].

## ✅ Optional: Input File for ZoKrates

Create a text file input.txt:

<secret\_key\_as\_integer> <time\_slot\_integer> <expected\_hash[0]> <expected\_hash[1]>

This makes it easy to test in ZoKrates:

bash:

zokrates compute-witness -a $(cat input.txt)