We recently open-sourced our zk-sgx-attester, a set of utilities designed to verify SGX Remote Attestation quotes onchain by moving DCAP verification offchain to a zkVM. In this post, I will provide background information and explain why and how we integrate ZK into TEE verification.

Background

The TEE convinces users that their application is indeed running within a genuine TEE by a protocol called Remote Attestation. It ensures:

- The hardware is a genuine TEE
- The software deployed to the TEE is unmodified
- It's possible to establish a secure communication channel between the user and the software

Making Remote Attestation verification trustless is fundamental to the trustworthiness of the entire protocol we build.

Intel SGX provides two methods for Remote Attestation:

- EPID-based attestation

: This method requires interaction with the Intel IAS service, which can introduce app-specific censorship issues. Fortunately, this method is being deprecated, and we have a better alternative: DCAP.

- DCAP-based attestation

: This method eliminates the need for interaction with Intel IAS and only requires fetching collateral (the information needed to verify the report) without any app-specific data from Intel. By using DCAP, we can achieve fully trustless remote attestation verification by proving the DCAP verification with ZK circuit.

[

DCAP-Verification.png

2000×1122 137 KB

](https://collective.flashbots.net/uploads/default/original/2X/3/3b6de4f92a81b712d76a2dd51b08053cd8477be1.jpeg)

Verify Remote Attestation offchain with ZK

One way to make the verification trustless is to move it onchain. For example, Automata implemented this in their Solidity DCAP verifier, and we did something similar in the Phala Blockchain, where it is part of the blockchain runtime model. In this way, trustworthiness is guaranteed by the blockchain consensus protocol. However, verifying Remote Attestation onchain is challenging, particularly on Ethereum.

- Lack of Cryptographic Primitives

: EVM lacks support for some native cryptographic primitives needed for quote verification, especially P256 signature verification.

- High Gas Costs

: Verifying attestation quotes on Ethereum is prohibitively expensive because it requires complex computations, such as parsing X.509 certificates and iterating through collateral data.

Instead of verifying the quote directly on Ethereum, we can leverage RiscZero's zkVM to handle the quote verification offchain and only submit the proof onchain.

Here's how it works:

1. Off-Chain Quote Verification

: * RiscZero's zkVM can execute the quote verification process using full Rust libraries. This includes accessing JSON, WebPKI certificate libraries, and cryptographic libraries easily.

- By running the quote verification with DCAP in RiscZero zkVM, we ensure that the process is secure and efficient. The zkVM acts as part of the guest code, handling the complexities of verification.
- RiscZero's zkVM can execute the quote verification process using full Rust libraries. This includes accessing JSON, WebPKI certificate libraries, and cryptographic libraries easily.

- By running the quote verification with DCAP in RiscZero zkVM, we ensure that the process is secure and efficient. The zkVM acts as part of the guest code, handling the complexities of verification.

- Generating Proof with STARK and SNARK

: * Under the hood, RiscZero zkVM uses a STARK prover to validate the verification program. This is then converted into a SNARK proof, specifically a Groth16 zk proof, which is very cheap to verify onchain.

- This approach significantly reduces the computational burden and gas costs associated with onchain verification, consuming only around 250k

gas for proof verification.

1. Under the hood, RiscZero zkVM uses a STARK prover to validate the verification program. This is then converted into a SNARK proof, specifically a Groth16 zk proof, which is very cheap to verify onchain.

2. This approach significantly reduces the computational burden and gas costs associated with onchain verification, consuming only around 250k

gas for proof verification.

1. Onchain Verification by Smart Contracts

: * Once the SNARK proof is generated, it can be submitted to a groth16 verifier deployed by RiscZero on Ethereum for verification.

1. Once the SNARK proof is generated, it can be submitted to a groth16 verifier deployed by RiscZero on Ethereum for verification.

[

emote-Attestation-With-RiscZero-zkVM

2000×1122 136 KB

](https://collective.flashbots.net/uploads/default/original/2X/e/e866e2cb1d87fd32ffc679ba7a4b2a54292c03ed.jpeg)

Code walkthrough

Let's jump into the code to see how we make it. The code is fully open sourced, you can find it onGithub.

- We built a local prover that allows anyone to generate ZKPs on their local machine. The main entry point ishere, and the prove

function accepts input passed from the command line, which includes the Attestation Quote generated inside the TEE and the Quote Collateral fetched from Intel PCS. We do not provide an implementation for quote generation, as it is highly framework-specific. Instead, we use a pre-generated quote for this purpose.

pub struct LocalProver {} impl LocalProver { /// Generates a snark proof as a triplet (Vec<u8>, FixedBytes<32>, /// `Vec) for the given elf and input. pub fn prove(elf: &[u8], input: &[u8]) -> Result<(Vec, FixedBytes<32>, Vec)> { ... } }

- As we mentioned, our goal is to make the process of DCAP verification trustless. The code running inside the zkVM (which will be proved) can be found here. dcap::verify performs the actual verification operation, here are some snippet codes:

// ...

// Check TCB info cert chain and signature verify_certificate_chain(&leaf_cert, intermediate_certs, now_in_milli)?; let asn1_signature = encode_as_der(&quote_collateral.tcb_info_signature)?; if leaf_cert.verify_signature(webpki::ring::ECDSA_P256_SHA256, quote_collateral.tcb_info.as_bytes(), &asn1_signature,)?;

// ...

// Check QE signature if leaf_cert .verify_signature(webpki::ring::ECDSA_P256_SHA256, &auth_data.qe_report, &asn1_signature)?;

// Check QE hash if qe_hash.as_ref() != &qe_report.report_data[0..32] { return Err(Error::QEReportHashMismatch); }

// Check signature from auth data peer_public_key.verify(&raw_quote[..(HEADER_BYTE_LEN + ENCLAVE_REPORT_BYTE_LEN)], &auth_data.ecdsa_signature,)?; // ...

The definition of the verify function can be found athere. Basically it does the following checks:

- Check TCB info cert chain and signature
- Check quote version and attestation key algo
- Extract certification from quote and verify the certification chain
- Exact QE signature and hash from quote for integrity checking
- Exact and check TCB info

The DCAP verification implementation is not the topic of this post, you can find more details on Intebnline documentation.

- Check TCB info cert chain and signature
- Check quote version and attestation key algo
- Extract certification from quote and verify the certification chain
- Exact QE signature and hash from quote for integrity checking
- Exact and check TCB info

- After the proving process is completed, we obtain the STARK proof generated by the zkVM. However, verifying a STARK proof on-chain is expensive. Instead, we translate it to a Groth16 proof, which is a type of SNARK proof that offers a cheap and constant verification cost onchain.

let seal = stark_to_snark(&seal_bytes).unwrap().to_vec();

- The final step is to verify the proof on-chain (specifically, on the EVM using a Solidity smart contract). The verification contract is straightforward because the contract for verifying SNARK proofs has been implemented and deployed by RiscZero. All we need to do is pass the proof to the Groth16 verifier.

/// @notice Check the proof of attestation verification and return the attestation output. function verifyAttestation(bytes calldata x, bytes32 postStateDigest, bytes calldata seal) external returns(bytes memory) { // Construct the expected journal data. Verify will fail if journal does not match. bytes memory journal = x; require(verifier.verify(seal, imageId, postStateDigest, sha256(journal)));

return journal;

}

Porting Ring to RiscZero

The DCAP verification relies on some cryptographic primitives provided by the Rust library ring

. However, ring

does not officially support compilation to the RiscZero zkVM target. If you want to use it in RiscZero zkVM, please ensure you use the patched version of ring

, where we have added support for compiling to RiscZero's RISC-V instruction set.

[patch.crates-io] ring = { git = "https://github.com/tolak/ring.git", package = "ring", branch = "patch-for-risc0" }

Conclusion

We have just completed the most important step of introducing ZK to TEE verification. However, there are still several improvements we need to make. The proving process on a local machine is still too time-consuming without GPU acceleration (cost 8 hours to prove on AMD EPYC 7742 64-Core Processor with 128G Mem). Outsourcing the proving computation to RiscZero Bonsai is a promising solution. Additionally, TDX Remote Attestation is more complex, and we are still exploring the proper way to verify it onchain.