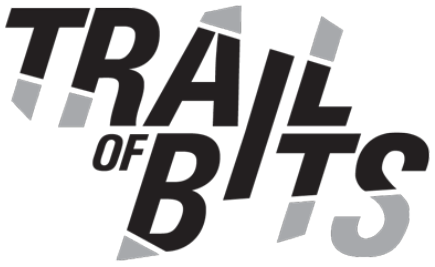

AZTEC

Security Assessment

September 27th, 2019

Prepared For:
Arnaud Schenk | *AZTEC*
arnaud@aztecprotocol.com

Prepared By:
Ben Perez | *Trail of Bits*
benjamin.perez@trailofbits.com

David Pokora | *Trail of Bits*
david.pokora@trailofbits.com

James Miller | *Trail of Bits*
james.miller@trailofbits.com

Will Song | *Trail of Bits*
will.song@trailofbits.com

Paul Kehrer | *Trail of Bits*
paul.kehrer@trailofbits.com

Alan Cao | *Trail of Bits*
alan.cao@trailofbits.com

Executive Summary

Project Dashboard

Engagement Goals

Coverage

Recommendations Summary
- Short Term
- Long Term

Findings Summary
- 1. IERC20 is incompatible with non-standard ERC20 tokens
- 2. AWS filesystem encryption is not configured
- 3. AWS instance is associated with a public IP address
- 4. AWS ALB load balancer allows plaintext traffic
- 5. AWS ALB load balancer has access logs disabled
- 6. AWS security group allowing all traffic exists
- 7. Failure to validate MPC output can lead to double spending attack
- 8. Random value generated can be zero or one
- 9. Missing check for address(0) in constructor of AdminUpgradeabilityProxy
- 10. Missing elliptic curve pairing check in the Swap Validator
- 11. Using nested hashes reduces security
- 12. Solidity compiler optimizations can be dangerous
- 13. Replay attack and revocation inversion on confidentialApprove
- 14. scalingFactor allows values leading to potentially undesirable behavior
- 15. MultiSigWallet can call invalid contracts
- 16. Time locking can overflow

A. Vulnerability Classifications

B. DeepState Testing Enhancements
- Build and Setup
- Test Cases
- Coverage Measurement
- Continuous Integration

C. Documentation Discrepancies

D. Solidity Testability

Executive Summary

From August 26th through September 27th 2019, AZTEC engaged with Trail of Bits to review the security of the AZTEC protocol. Trail of Bits conducted this assessment over the course of 10 person-weeks with 2 engineers working using commit hash c3f49df5 for the AztecProtocol/AZTEC repository and commit hash 230a1d8a for the AztecProtocol/Setup repository.

The first two weeks of the assessment focused on a review of the trusted setup protocol and codebase. Trail of Bits performed a cryptographic review of the trusted setup protocol described in the documentation provided. We also documented where the implementation differed from the protocol documentation (Appendix C) and examined whether these discrepancies caused any vulnerabilities. Additionally, we integrated DeepState to detect unsafe behavior and expand test coverage in `setup-tools`, the C++ portion of the codebase (Appendix B).

The remaining three weeks of the audit were dedicated to a review of the AZTEC protocol and codebase. Trail of Bits continued cryptographic review of the documentation and the AZTEC whitepaper, and documented and assessed any deviations from these specifications. In addition, we reviewed the AZTEC smart contracts to detect any unsafe, low-level Solidity behaviors. Finally, we integrated Echidna into the Solidity smart contracts and provided guidance for future improvements to Solidity testability (Appendix D) to improve coverage and detect unsafe behavior.

Over the course of the audit, Trail of Bits discovered four high-severity issues. Two of those issues (TOB-AZT-010 and TOB-AZT-013) were found to have low exploitation difficulty. TOB-AZT-010 can lead to an adversary arbitrarily increasing their balance, and TOB-AZT-013 can lead to an adversary controlling permissions for note spending. The other two high-severity issues (TOB-AZT-007 and TOB-AZT-008) had a high difficulty of exploitation as they would rely on low-probability events. Trail of Bits also discovered three medium-severity issues (TOB-AZT-001, TOB-AZT-009, and TOB-AZT-015) concerning potential non-standard token behavior, null administrator addresses, and invalid contracts, respectively. Seven low-severity issues were reported, five of which (TOB-AZT-002 through TOB-AZT-006) pertain to AWS configurations. The cryptographic low-severity issue, TOB-AZT-011 was found to be of high difficulty to exploit, relying on low-probability events. The final low-severity issue, TOB-AZT-016, resulted from not using `SafeMath` to prevent integer overflow. The remaining two issues, TOB-AZT-012 and TOB-AZT-014, were found to be of undetermined and informational severity, respectively. For these, we do not report any immediate threats, but believe they merit attention.

In almost all of the cryptographic findings discovered by Trail of Bits, the implementation deviated from the protocol specified in the documentation. We do not report any

cryptographic findings with the documentation itself. The remaining findings were due to AWS configurations and Solidity behavior that can be dangerous.

We recommend that AZTEC adhere as closely as possible to their documentation in their implementations going forward. We also encourage AZTEC to familiarize themselves with AWS configuration best practices and to stay informed of issues that are found in other Solidity contracts. Lastly, we recommend that AZTEC integrate our tooling, DeepState and Echidna, to continue to expand test coverage. Specifically, we encourage AZTEC to address the software testing obstacles described in Appendix D.

Project Dashboard

Application Summary

Name	AZTEC
Version	AztecProtocol/AZTEC commit: c3f49df5 AztecProtocol/Setup commit: 230a1d8a
Type	C++, JavaScript, Solidity, TypeScript
Platforms	Ethereum

Engagement Summary

Dates	August 26th through September 27th 2019
Method	Whitebox
Consultants Engaged	2
Level of Effort	10 person-weeks

Vulnerability Summary

Total High-Severity Issues	4	■ ■ ■ ■
Total Medium-Severity Issues	3	■ ■ ■
Total Low-Severity Issues	7	■ ■ ■ ■ ■ ■ ■
Total Informational-Severity Issues	1	■
Total Undetermined-Severity Issues	1	■
Total	16	

Category Breakdown

Configuration	5	■ ■ ■ ■ ■
Cryptography	5	■ ■ ■ ■ ■
Data Validation	5	■ ■ ■ ■ ■
Undefined Behavior	1	■
Total	16	

Engagement Goals

The AZTEC protocol uses commitment functions and zero-knowledge proofs to define a confidential transaction protocol. Since the transactions are confidential, the exact balances of the digital assets are secret, but their validity is established with range proofs. The design of the protocol was aimed at reducing the complexity of constructing and verifying proofs. Specifically, the complexity of their range proofs does not scale with the size of the range. However, this achievement comes at the cost of requiring a trusted setup protocol to produce a set of common parameters. The security of the entire AZTEC protocol relies on producing these common parameters without revealing the secret value, `y`. To achieve this, AZTEC developed a multi-party computation setup protocol that attempts to compute these parameters without revealing the secret value.

Since this trusted setup protocol is vital to the security of the AZTEC protocol, AZTEC sought to assess the security of this protocol. This portion of the engagement was scoped to provide a cryptographic and security assessment of their `setup-tools` and `setup-mpc-server`. This took the form of both manual and dynamic analysis, using DeepState integration to perform dynamic test coverage of the C++ portions of the codebase.

AZTEC also sought to assess the security of the AZTEC protocol itself. This portion of the engagement was scoped to provide a cryptographic and security assessment of the `ACE` and `ERC1724` smart contracts. Assessment of these contracts coincided with assessment of the corresponding JavaScript packages, such as aztec.js and secp256k1. These assessments took the form of manual review and dynamic test coverage using Echidna.

Specifically, we sought to answer the following questions:

- Is the multi-party computation protocol specified in the documentation supplied by AZTEC cryptographically secure?
- Is the AZTEC protocol specified in the AZTEC whitepaper cryptographically secure?
- Does the implementation of the multi-party computation protocol in `setup-tools` and `setup-mpc-server` comply with the documentation?
- Does the implementation of the AZTEC protocol in the `ACE` and `ERC1724` contracts comply with the whitepaper?
- If the implementation does not comply with its corresponding documentation, does this introduce any vulnerabilities?
- Are there any unsafe, low-level Solidity behaviors? Does the use of the `delegatecall` pattern in Solidity introduce any vulnerabilities?
- Are there any flaws in how permissions and modifiers are used in Solidity?
- Can we use dynamic testing to detect unsafe behavior in both `C++` and Solidity?

Coverage

`setup-tools`. Extensive manual review was dedicated to the `setup-tools`. This review coincided with an extensive review of the trusted setup documentation, as these tools are direct implementations of this documentation. Manual review took the form of verifying the cryptographic security of the protocol, discovering and documenting any deviations from the protocol in the implementation, and identifying potential vulnerabilities associated with those deviations. DeepState was also integrated into this codebase to increase testing coverage and attempt to discover any dangerous code behavior.

`setup-mpc-server`. Manual review was also dedicated to the `setup-mpc-server`. Since the `setup-mpc-server` uses the `setup-tools` to verify the setup protocol, this review focused on ensuring the `setup-tools` were being used in the correct manner.

`ACE` contracts. Manual review was first performed on the AZTEC whitepaper. Once that review was complete, Trail of Bits assessed how well the `ACE` contracts adhered to the whitepaper specifications. We also reviewed the Solidity assembly and other areas of the codebase that can lead to dangerous behavior, like the `delegatecall` pattern and the use of `SafeMath`. Review of the `ACE` contracts coincided with a review of the `aztec.js` package, as this package is used to interact with the `ACE` contracts.

`ERC1724` contracts. Trail of Bits also dedicated manual review to the `ERC1724` contracts. These contracts use a signature scheme that is mentioned in the documentation but not in great detail; for example, features like `confidentialApprove` were not mentioned in the supplied documentation. As a result, we performed a cryptographic review of the digital signatures used throughout these contracts. In addition, the Solidity code was again reviewed for potential dangerous behavior. Review of these contracts coincided with a review of the `secp256k1` signature package.

Recommendations Summary

Short Term

❑ **Add support for popular ERC20 tokens with incorrect/missing return values.** Doing so prevents non-standard ERC20 tokens from failing unexpectedly with AZTEC. TOB-AZT-001

❑ **Enable `encrypted` configuration properties for sensitive items in AWS.** This is a Defense in Depth strategy to minimize the risk of exposing potentially sensitive data. TOB-AZT-002

❑ **Disable any public IP address association to any AWS instances.** Minimizing your infrastructure's exposure to the public internet allows for better monitoring and access control, and limits attack surface. TOB-AZT-003

❑ **Only allow connections to the load balancer via HTTPS.** Disabling HTTP redirect prevents silent downgrade attacks and removes the risk of plaintext communication. TOB-AZT-004

❑ **Enable access logs for the setup `aws_alb.setup`.** Enabling logging will significantly improve infrastructure visibility and faster detection of potential attacks. TOB-AZT-005

❑ **Restrict the HTTPS security group such that only the load balancer placed in front of the instance can access it.** This will prevent lateral movement from other AWS components that may have been compromised. TOB-AZT-006

❑ **In AZTEC protocol setup, verify that `y` is not less than `k_max`.** If `y` is less than `k_max` then an attacker can recover `y` and will be able to conduct a double spend attack. TOB-AZT-007

❑ **When generating random values used for verification or proving, ensure that those values are not equal to zero or one.** If these values occur, invalid data may pass verification checks in both the trusted setup and JoinSplit proofs. TOB-AZT-008

❑ **Always perform zero-address checks when setting up permissions.** If this value is set to zero, the contract cannot be administered. TOB-AZT-009

❑ **Validate output notes in the `Swap` protocol by using the same elliptic curve pairing verification check as the other Join-Split protocols.** Without this verification check, adversaries can raise their balance arbitrarily. TOB-AZT-010

❑ **Add an additional value as input into the hash function at each iteration in the JoinSplit verifier to avoid the security reductions associated with nested hashes.**

Nested hashes reduce the image space and lower the security margin of the hash function. TOB-AZT-011

❑ **Measure the gas savings from Solidity compiler optimizations, and carefully weigh them against the possibility of an optimization-related bug.** Solidity compiler optimizations have had multiple security-related issues in the past. TOB-AZT-012

❑ **Update `confidentialApprove` to tie the `_status` value into the signature used to give and revoke permission to spend.** An attacker can modify permissions without authorization because the signature is not tied to `_status`. TOB-AZT-013

❑ **Maintain state to prevent replay of previous signatures with `confidentialApprove`.** An attacker can replay previous signatures to reinstate revoked permissions. TOB-AZT-013

❑ **Add checks in the `NoteRegistryManager` to prevent unsafe `scalingFactor` values.** Accidental input of unsafe values could result in a Denial of Service. TOB-AZT-014

❑ **Prevent calling invalid contracts in `MultiSigWallet` by adding checks in `addTransaction` and `executeTransaction` or `external_call`.** When a transaction calls a self-destructed contract, the transaction will still execute without the contract behaving as expected. TOB-AZT-015

❑ **Use `SafeMath` to avoid overflow in time locking contracts.** Overflow can cause checks to improperly pass. TOB-AZT-016

Long Term

❑ **Carefully review common misuses of the ERC20 standard.** Slight deviations from this standard have led to subtle bugs with interoperability in the past. TOB-AZT-001

❑ **Review all Terraform configurations to ensure best practices are in place.** Configuration management systems such as Terraform represent a large surface area of attack with deep access across the entire platform. TOB-AZT-002, TOB_AZT_003, TOB_AZT_004, TOB_AZT_005, TOB_AZT_006

❑ **Validate all cryptographically sensitive parameters, even if they are unlikely to be malicious.** Parameter validation serves as an additional defense against system compromise. TOB-AZT-007

❑ **Add an additional check for trivial points in `verifier.cpp`, as this will prevent similar exploits.** Defending against this issue more generically hardens the codebase against similar attacks in the future. TOB-AZT-008

❑ **Review invariants within all components of the system and ensure these properties hold.** Consider testing these properties using a property-testing tool such as Echidna to increase the probability of detecting dangerous edge cases. TOB-AZT-009

❑ **Whenever developing contracts for verifying Join-Split protocols, ensure that there is always an elliptic curve pairing check.** Consider property-based testing with tools such as Echidna in these types of validators and other cryptographic protocols. TOB-AZT-010

❑ **Check that none of the x_i values during verification repeat or are equal to `0`.** Without these checks, invalid data may pass verification. TOB-AZT-011

❑ **Monitor the development and adoption of Solidity compiler optimizations to assess their maturity.** As the Solidity compiler matures, these optimizations should stabilize. TOB-AZT-012

❑ **Adjust the signature verified in `confidentialApprove` so that it is not replayable.** Mixing a nonce and the status type with the signature allows for minimal state while preventing both replay and mutability. TOB-AZT-013

❑ **Ensure all call sites where `SafeMath` is used are carefully verified.** Improper use of `SafeMath` can trap certain state transitions. TOB-AZT-014

❑ **Monitor issues discovered in code that is imported from external sources.** Contracts like `MultiSigWallet` are commonly used and can have known vulnerabilities that developers are not aware of. TOB-AZT-015

❑ **Use DeepState to increase coverage and catch more subtle bugs.** Ensemble fuzzing and property testing allow more complex bugs to be caught before deployment. Appendix B

❑ **Ensure the documentation accurately reflects the protocol as implemented.** Deviations between specification and implementation can lead to correctness bugs when other developers attempt to create their own interoperable systems. Appendix C

❑ **Consider expanding your Solidity code to allow for better property-based fuzzing.** The current mix of JavaScript and Solidity makes use of fuzzing and property testing tools more difficult. Appendix D

Findings Summary

#	Title	Type	Severity
1	IERC20 is incompatible with non-standard ERC20 tokens	Data Validation	Medium
2	AWS file system encryption is not configured	Configuration	Low
3	AWS instance is associated with a public IP address	Configuration	Low
4	AWS ALB load balancer allows plaintext traffic	Configuration	Low
5	AWS ALB load balancer has access logs disabled	Configuration	Low
6	AWS security group allowing all traffic exists	Configuration	Low
7	Failure to validate MPC output can lead to double spending attack	Cryptography	High
8	Random value generated can be zero	Cryptography	High
9	Missing check for address(0) in constructor of AdminUpgradeabilityProxy	Data Validation	Medium
10	Missing elliptic curve pairing check in the Swap Validator	Cryptography	High
11	Using nested hashes reduces security	Cryptography	Low
12	Solidity compiler optimizations can be dangerous	Undefined Behavior	Undetermined
13	Replay attack and revocation inversion on confidentialApprove	Cryptography	High

14	scalingFactor allows values leading to potentially undesirable behavior	Data Validation	Informational
15	MultiSigWallet can call invalid contracts	Data Validation	Medium
16	Time locking can overflow	Data Validation	Low

1. IERC20 is incompatible with non-standard ERC20 tokens

Severity: Low
Type: Data Validation
Target: `IERC20.sol`

Difficulty: Medium
Finding ID: TOB-AZT-001

Description
`IERC20` is meant to work with any ERC20 token. Several high-profile ERC20 tokens do not correctly implement the ERC20 standard. Therefore, `IERC20` will not work with these tokens.

The ERC20 standard defines three functions, among others:

- `approve(address _spender, uint256 _value) public returns (bool success)`
- `transfer(address _to, uint256 _value) public returns (bool success)`
- `transferFrom(address _from, address _to, uint256 _value) public returns (bool success)`

Several high-profile ERC20 tokens do not return a boolean on these three functions. Starting from Solidity 0.4.22, the return data size of external calls is checked. As a result, any call to `approve`, `transfer`, or `transferFrom` will fail for ERC20 tokens that implement the standard incorrectly.

Examples of popular ERC20 tokens that are incompatible include BinanceCoin, OmiseGo, and Oyster Pearl.

Exploit Scenario
Bob creates a note registry, providing a `linkedTokenAddress` pointing to a non-standard ERC20 token that does not implement the correct interface for `transfer`/`transferFrom`. Upon updating the state of the note registry, a call to the transfer methods will cause a revert.

Recommendation
Short term, consider adding support for popular ERC20 tokens with incorrect/missing return values. This could be achieved with a whitelist for such tokens that should use an alternative interface. Alternatively, a carefully crafted low-level call would prevent a revert and allow for manual validation of return data.

Long term, carefully review the usage and issues of the ERC20 standard. This standard has a history of misuses and issues.

References

- Missing return value bug—At least 130 tokens affected
- Explaining unexpected reverts starting with Solidity 0.4.22

2. AWS filesystem encryption is not configured

Severity: Low
Type: Configuration
Target: `setup-mpc-server\terraform\main.tf`

Difficulty: Low
Finding ID: TOB-AZT-002

Description
The Terraform configuration provided in `setup-mpc-server` does not have filesystem encryption enabled.

By default, users should opt-in to AWS filesystem encryption to add an extra layer of security to any servers housing sensitive data. If the contents of the AWS are accessed by an intruder, retrieving a plain-text copy of the underlying data will be non-trivial.

Exploit Scenario
Bob manages the AWS instance described by `setup-mpc-server`'s Terraform configuration file. As a result of an AWS vulnerability, an attacker, Eve, gains access to Bob's AWS instance. When the instance is offline, the lack of encryption will make data easier to exfiltrate. Additionally, setting up an encryption scheme with rotating keys would be non-trivial.

Recommendation
Short term, enable the `encrypted` configuration property for `aws_efs_file_system.setup_data_store`, setting a `kms_key_id`, and using an `ebs_block_device`. If `ebs_block_device` is not sensible to use, encryption and key rotation should be manually configured for the instance.

Long term, review all Terraform configurations to ensure best practices are in place. Use static code analyzers such as Terrascan to ensure all recommended security practices are met.

3. AWS instance is associated with a public IP address

Severity: Low
Type: Configuration
Target: `setup-mpc-server\terraform\main.tf`

Difficulty: Low
Finding ID: TOB-AZT-003

Description
The Terraform configuration provided in `setup-mpc-server` associates a public IP address with the AWS instance.

It is recommended that users do not associate a public IP address with their AWS instances because it allows direct access to the server. In the event of an attack, the victim AWS instance would be solely responsible for any security mitigations. Instead, users should opt to use an Application Load Balancer (ALB) or an Elastic Load Balancer (ELB) to expose any underlying servers. By default, AWS' load balancers only allow traffic to the underlying AWS instances if they are explicitly approved. In this way, load balancers behave similar to a firewall, adding an extra layer of protection for the AWS instance.

Exploit Scenario
Bob operates an AWS instance that processes sensitive information. Eve wishes to hack into Bob's server and notices that Bob's AWS instance has been assigned a public IP address. Eve can now carry out attacks directly against the machine.

Recommendation
Short term, disable any public IP address association with any AWS instances. Opt to use a load balancer in front of the instance instead, thus adding an additional layer of protection for the instance.

Long term, review all Terraform configurations to ensure best practices are in place. Use static code analyzers such as Terrascan to ensure all recommended security practices are met.

4. AWS ALB load balancer allows plaintext traffic

Severity: Low
Type: Configuration
Target: `setup-iac\main.tf`

Difficulty: Low
Finding ID: TOB-AZT-004

Description
The Terraform configuration provided in `setup-iac` is configured to use the HTTP protocol where it should use HTTPS.

Currently, the `alb_listener` is an HTTP listener meant to redirect to HTTPS. Allowing such a redirection encourages usage of insecure protocols, and data POSTed to the server will be sent unencrypted until such a redirect occurs. Although it may not pose an immediate threat, later code revisions may transmit sensitive information over the network that could be sniffed until such a redirect occurs. More generally, it enables HTTP downgrade attacks.

Exploit Scenario
Alice operates an AWS instance that processes sensitive information. Bob wishes to use the services provided by Alice's server, and POSTs his own sensitive information to it over HTTP. An attacker, Eve, who is connected to the same local network as Bob, can then perform a man-in-the-middle attack to sniff Bob's local network traffic and extract his underlying communications. Thus Eve can extract Bob's secrets.

Recommendation
Short term, disallow connection to the load balancer via HTTP. Ensure all tooling making use of HTTP connections is updated to access the HTTPS endpoint, and all relevant documentation encourages users to use HTTPS.

Long term, review all Terraform configurations to ensure best practices are in place. Use static code analyzers such as Terrascan to ensure all recommended security practices are met.

5. AWS ALB load balancer has access logs disabled

Severity: Low
Type: Configuration
Target: `setup-iac\main.tf`

Difficulty: Low
Finding ID: TOB-AZT-005

Description
The Terraform configuration provided for `setup-iac` is not configured to maintain access logs.

Due to the lack of access logs on the ALB configured by `setup-iac`, the owner of the ALB load balancer will be limited in their ability to investigate attacks against their infrastructure.

Exploit Scenario
Bob operates an AWS instance that is exposed through the above-mentioned ALB. Eve, an attacker, sends a large number of requests intended to attack Bob's server. Without access logs configured for the ALB, Bob struggles to determine if an attack recently occurred, or where it originated.

Recommendation
Short term, enable access logs for the setup `aws_alb.setup` in order to provide additional data that will help diagnose/detect attacks.

Long term, review all Terraform configurations to ensure best practices are in place. Use static code analyzers such as Terrascan to ensure all recommended security practices are met.

6. AWS security group allowing all traffic exists

Severity: Low
Type: Configuration
Target: `setup-iac\main.tf`

Difficulty: Low
Finding ID: TOB-AZT-006

Description
The Terraform configuration provided for `setup-iac` configures a security group rule for HTTPS that allows any incoming address.

Security groups are used to constrain incoming and outgoing traffic. However, `setup_public_allow_https` allows all incoming addresses to access HTTPS (443). Instead, the AWS instance should only allow ingress from a load balancer, and the load balancer should impose restrictions on connections to the instance.

Exploit Scenario
Bob operates an AWS instance described by the above Terraform configuration. He allows traffic to the AWS instance from all addresses. Despite not being associated with a public IP address, any additional components added to the AWS group can then access Bob's loosely configured AWS instance. Access to any machine in the access group allows access to communicate with the AWS instance, when only the explicitly allowed devices (e.g., a load balancer) should be able to maintain such communications with the instance.

Recommendation
Short term, restrict the HTTPS security group so that only the load balancer placed in front of the instance can access it. This will prevent lateral movement from other AWS components that may have been compromised.

Long term, review all Terraform configurations to ensure best practices are in place. Use static code analyzers such as Terrascan to ensure all recommended security practices are met.

7. Failure to validate MPC output can lead to double spending attack

Severity: High
Type: Cryptography
Target: `verifier.cpp`

Difficulty: High
Finding ID: TOB-AZT-007

Description
Currently the MPC server does not check if the jointly computed secret value is between 0 and `k_max`. If this occurs, any user of the AZTEC system will be able to compute the secret value and create transactions with arbitrary amounts of money. While this event is extremely unlikely, failure to detect it would lead to a complete failure of the AZTEC system.

Exploit Scenario
During the MPC setup, the parties jointly compute a secret value that is smaller than `k_max`. A user of the AZTEC system is able to detect this by checking if any of the Boneh-Boyen signatures are 1. Using this information, they compute the secret value and can double spend arbitrary amounts of money.

Recommendation
Short term, verify that the secret value is not less than k_max. This can be achieved by checking if any of the final Boneh-Boyen signatures are equal to 1.

Long term, validate all cryptographically sensitive parameters, even if they are extremely unlikely to be malicious.

8. Random value generated can be zero or one

Severity: High | Difficulty: High
Type: Cryptography | Finding ID: TOB-AZT-008
Target: `setup/setup.cpp, verify/verifier.cpp, ACE/validator`

Description
Using the libff random number generator can lead to accidentally generating a zero or one. The libff random number generator generates random bits, setting all bits higher than the highest non-zero bit of the modulus to zero, and succeeds when the resulting number is between 0 (inclusive) and the modulus (exclusive). If the random value generated in `same_ratio_preprocess` is zero or one, this can lead to a false verification of invalid `G1` points.

```
void run_setup(std::string const &dir, size_t num_g1_points, size_t num_g2_points)
{
    // ...
    Secret<Fr> multiplicand(Fr::random_element());
    // ...
    if (cmd == "create")
    {
        size_t num_g1_points, num_g2_points, points_per_transcript;
        iss >> num_g1_points >> num_g2_points >> points_per_transcript;
        compute_initial_transcripts(dir, num_g1_points, num_g2_points,
                                    points_per_transcript, multiplicand, progress);
    }
    else if (cmd == "process")
    {
        size_t num;
        iss >> num;
        compute_existing_transcript(dir, num, multiplicand, progress);
    }
    // ...
}
```

Figure 8.1: Multiplicand of zero being used in `Setup`.

`Keccak256` is used to generate random values, x_i, that are used in the ACE/validator contracts. The `Keccak256` function can output `0` and `1`, which can lead to the false validation of invalid proofs.

Exploit Scenario
This vulnerability can be exploited both in `setup.cpp` and `verifier.cpp`.

Verification
A malicious adversary can generate a valid `G2` point and a valid `G1[0]` point, with the rest of the `G1` points invalid. If the random value generated is zero, this will verify.

Similarly, an adversary can generate a valid `G2` point and a valid `G1[0]`, set the `G1[N-1]` point equal to a valid `G1[1]` point, and zero out the rest of the `G1` points. If the random value generated is one, this will also lead to a false verification.

```
VerificationKey<GroupT> same_ratio_preprocess(std::vector<GroupT> const &g_x)
{
    Fq challenge = Fq::random_element();

    std::vector<Fq> scalars(g_x.size());
    scalars[0] = challenge;
    for (size_t i = 1; i < scalars.size(); ++i)
    {
        scalars[i] = scalars[i - 1] * challenge;
    }
    // ...
}
```

Figure 8.2: Scalar array being set to zero in `Verify`.

Setup
If an honest client running the Setup tool has a random "toxic waste" value generated to be zero, their transcript will be classified as invalid. This is not an exploit, but it is undesirable behavior for parties acting honestly.

ACE/validator
A malicious party can generate an invalid commitment (γ_1, σ_1) and another invalid commitment $(\gamma_i, \sigma_i) = (\gamma_1^{-1}, \sigma_1^{-1})$, with the rest of the commitments valid. Then if the resulting x_i value is either `0` or `1`, the validator will incorrectly validate invalid commitments.

Recommendation
Short term, keep generating random values until a value not equal to zero or one is found.

Long term, we also recommend adding an additional check for trivial points in `verifier.cpp`, as this will prevent similar exploits. The specification should also be updated to note that `r=0` causes the first validation in V_{mpc} to always succeed. Lastly, when dealing with random numbers, include test coverage for when that random value is `0`.

9. Missing check for address(0) in constructor of AdminUpgradeabilityProxy

Severity: Medium
Type: Data Validation
Target: `AdminUpgradeabilityProxy.sol`

Difficulty: High
Finding ID: TOB-AZT-009

Description
In `BaseAdminUpgradeabilityProxy`, the `changeAdmin` function performs a zero-address check before calling `_setAdmin()`.

```
function changeAdmin(address newAdmin) external ifAdmin {
        require(newAdmin != address(0), "Cannot change the admin of a proxy to the zero address");
        emit AdminChanged(_admin(), newAdmin);
        _setAdmin(newAdmin);
    }
```

Figure 9.1: `changeAdmin` function in `BaseAdminUpgradeabilityProxy`.

However, the constructor of `AdminUpgradeabilityProxy` calls `_setAdmin()` without performing a zero-address check.

```
constructor(address _logic, address _admin, bytes memory _data) UpgradeabilityProxy(_logic, _data) public payable {
        assert(ADMIN_SLOT == bytes32(uint256(keccak256('eip1967.proxy.admin')) - 1));
        _setAdmin(_admin);
    }
```

Figure 9.2: The constructor for `AdminUpgradeabilityProxy`.

If the constructor for `AdminUpgradeabilityProxy` is called with `_admin` set to zero, then the contract will be un-administrable.

Exploit Scenario
The AZTEC deployment system has an implementation error and mistakenly sets the admin address of `AdminUpgradeabilityProxy` to zero.

A malicious internal user at AZTEC sabotages the setup procedure and uses their administrative privileges to set the admin address in `AdminUpgradeabilityProxy` to zero.

Recommendation
Short term, always perform zero-address checks when setting up permissions.

Long term, review invariants within all components of the system and ensure these properties hold. Consider testing these properties using a property-testing tool such as Echidna.

10. Missing elliptic curve pairing check in the Swap Validator

Severity: High
Type: Cryptography
Target: `Swap.sol`

Difficulty: Low
Finding ID: TOB-AZT-010

Description
The Swap protocol does not perform an elliptic curve pairing check to validate its output notes. The protocol performs all of the other checks of the Join-Split protocol, but without the elliptic curve pairing, the validator can be tricked into validating output notes with invalid commitments.

Exploit Scenario
Any adversary can easily exploit this by submitting an invalid commitment that is not computationally binding. For example, an adversary can submit the following invalid commitment: $(\gamma, \sigma) = (1, h^a)$. This commitment can be used to commit to `(k,a)` for any `k` value.

An adversary can submit this commitment as an output note to the Swap protocol, where they choose the `k` value that will make the validator verify this proof (this `k` value will just be the `k` value of the input note). Since there is no elliptic curve pairing check, the validator will accept this as valid.

Once accepted by the validator, this malicious note will now be on the Note Registry, where it can be input to any Join-Split protocol. All of the Join-Split protocols will assume that the input notes have already been verified, so they will not detect it as malicious. Since this commitment can commit to any `k` value, an adversary can make this `k` value much higher than the value in the Swap protocol, successfully raising their balance for free.

Recommendation
Short term, validate output notes by using the same elliptic curve pairing verification check as the other Join-Split protocols.

Long term, whenever developing contracts for verifying Join-Split protocols, ensure that there is always an elliptic curve pairing check. Consider property-based testing with tools such as Echidna in these types of validators and other cryptographic protocols.

11. Using nested hashes reduces security

Severity: Low
Type: Cryptography
Target: `ACE/validators`

Difficulty: High
Finding ID: TOB-AZT-011

Description
The validator contracts implement an optimized Join-Split verification that uses random x_i values to verify a proof with only one elliptic curve pairing operation. To compute the `i`-th `x` value, the validator computes `i` hashes on the same input. Using nested hashes in this manner can be shown to be less secure than using the regular hash function. For instance, for a hash function `H`, it can be shown that it is `n` times easier to invert $H^n(x)$ than `H(x)` for any `x` (here, $H^n(x)$ refers to computing the hash of `x`, `n` times). For more information, see "A Graduate Course in Applied Cryptography" by Boneh and Shoup, section 18.4.3.

Exploit Scenario
With a weakened hash function generating random values, it is easier for an adversary to discover inputs that generate problematic random values. For instance, if an adversary could produce inputs that hash to the value of `0`, they can cause the validator to validate invalid output notes. Additionally, if an adversary could cause the hash function to produce an identical (x_i, x_j) pair (with `i` not equal to `j`), they can also cause the validator to validate invalid output notes.

Recommendation
Short term, add an additional value as input into the hash function at each iteration, rather than solely inputting the output of the previous iteration.

Long term, check that none of the x_i values repeat or are equal to `0` to prevent similar attacks from happening.

12. Solidity compiler optimizations can be dangerous

Severity: Undetermined
Type: Undefined Behavior
Target: `Solidity`

Difficulty: Low
Finding ID: TOB-AZT-012

Description
There have been several bugs with security implications related to optimizations. Moreover, optimizations have changed in the recent past. Solidity compiler optimizations are disabled by default. It is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe. Moreover, the Common Subexpression Elimination (CSE) optimization procedure is "implemented in a very fragile manner, with manual access to indexes, multiple structures with almost identical behavior, and up to four levels of conditional nesting in the same function." Similar code in other large projects have resulted in bugs.

There are likely latent bugs related to optimization, and/or new bugs that will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations causes an unexpected security vulnerability in a contract.

Recommendation
Short term, measure the gas savings from optimizations, and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

13. Replay attack and revocation inversion on confidentialApprove

Severity: High
Type: Cryptography
Target: `ZKAssetBase.sol, aztec.js/src/signer/index.js`

Difficulty: Low
Finding ID: TOB-AZT-013

Description
The `confidentialApprove` method is used to allow third parties to spend notes on the note owner's behalf. In the `confidentialApprove` method, there is a call to `validateSignature`, which verifies that the signature giving (or revoking) permission to spend is valid and was signed by the owner. However, when verifying the signature, the `_status` (indicating giving or revoking of permission) is not actually tied to the signature (see figure 13.1), so the same signature used to revoke permission can be used to restore permission. This problem is compounded because there is no mechanism in place to detect the resubmission of previously used signatures.

```
function confidentialApprove(
        bytes32 _noteHash,
        address _spender,
        bool _status,
        bytes memory _signature
    ) public {
        ( uint8 status, , , ) = ace.getNote(address(this), _noteHash);
        require(status == 1, "only unspent notes can be approved");
        bytes32 _hashStruct = keccak256(abi.encode(
                NOTE_SIGNATURE_TYPEHASH,
                _noteHash,
                _spender,
                status
        ));

        validateSignature(_hashStruct, _noteHash, _signature);
        confidentialApproved[_noteHash][_spender] = _status;
    }
```

Figure 13.1: The `confidentialApprove` method in `ZKAssetBase.sol`. The `_status` value is not included in the `_hashStruct` used to verify the signature.

Exploit Scenario
This can be exploited to wrongfully give or revoke permissions. Say an owner decides to revoke permission that was previously given to a third party. That party can resubmit either the original signature giving permission or the latest signature revoking permission (where they switch the permission back to true) and wrongfully reinstate their permission to spend notes.

Further, another malicious party (who is neither the owner nor third party) can revoke any permission given to third parties. If an owner submits a signature giving permission to a third party, this malicious party can easily resubmit this signature and switch the `_status` to `false`. Again, since `_status` is not tied to the signature, they will accept this malicious submission.

Recommendation
Short term, update `confidentialApprove` to tie the `_status` value into the signature used to give/revoke permission to spend. In order for valid signatures to work, this will also require updating `signer/index.js` in `aztec.js` to tie this value into the signature (presently, the `_status` value is always set to `true`; see Figure 13.2). In addition, we recommend maintaining state to prevent the replay of previous signatures.

```
signer.signNoteForConfidentialApprove =
  (verifyingContract, noteHash, spender, privateKey) => {
    const domain = signer.generateZKAssetDomainParams(verifyingContract);
    const schema = constants.eip712.NOTE_SIGNATURE;
    const status = true;
    const message = {
        noteHash,
        spender,
        status,
    };

    const { unformattedSignature } = signer.signTypedData(domain, schema, message, privateKey);
    const signature = `0x${unformattedSignature.slice(0, 130)}`;
    return signature;
};
```

Figure 13.2: The `signNoteForConfidentialApprove` method in `signer/index.js`. The `status` value is always set to `true`.

Long term, adjust the signature scheme so it is not detachable in this way, in order to prevent similar replay attacks in the future.

14. scalingFactor allows values leading to potentially undesirable behavior

Severity: Informational
Type: Data Validation
Target: `NoteRegistryManager.sol`

Difficulty: Hard
Finding ID: TOB-AZT-014

Description
The `NoteRegistryManager` contract defines how notes will be managed. When creating a note registry, a `scalingFactor` is defined, which represents the value of an AZTEC asset relative to another asset. Both the `scalingFactor` and `totalSupply`, which represents the total amount of notes, are `uint256` types which are operated on via `SafeMath` functions. When a transaction is performed that requires the supply to be increased (e.g., when another asset is converted into an AZTEC asset), the `scalingFactor` is used to determine how much to increase the supply.

Since the `totalSupply` is a `SafeMath` `uint256`, the supply will be capped at the maximum `uint256` value. By design, `SafeMath` ensures that once the `totalSupply` reaches this maximum value, every transaction that results in an increase in the `totalSupply` will result in a `revert`. Since the maximum `unit256` value is very large, this will only be problematic when the `scalingFactor` value is set to very small values or `0`. However, there is currently no mechanism in place to prevent potentially unsafe `scalingFactor` values.

Exploit Scenario
A note registry for an asset is incorrectly set up with a very small `scalingFactor` value. A user then employs a few tokens of another asset to cap out the supply of this asset. As a result, no other users can obtain this asset, and all transactions that would increase supply will `revert`.

Recommendation
Short term, add checks in the `NoteRegistryManager` to prevent unsafe `scalingFactor` values from being used.

Long term, ensure all call sites where `SafeMath` is used are carefully verified, as it can trap certain state transitions.

15. MultiSigWallet can call invalid contracts

Severity: Medium
Type: Data Validation
Target: `MultiSigWallet.sol`

Difficulty: Low
Finding ID: TOB-AZT-015

Description
Transactions are added to the `MultiSig` wallet via `submitTransaction`. Except for a zero check, there is currently no mechanism to verify that the `destination` address corresponds to a valid contract.

```
modifier notNull(address _address) {
    require(_address != address(0x0));
    _;
}

function addTransaction(address destination, uint value, bytes memory data)
    internal
    notNull(destination)
    returns (uint transactionId) {
    // ...
}
```

Figure 15.1: The `addTransaction` method in `MultiSigWallet.sol`.

Furthermore, contract validity is not checked in `external_call`, and it is documented that the low level instruction `call` will return `true` even if the target contract does not exist. When `external_call` is executed by `executeTransaction` with an invalid contract, the wallet will mark the transaction as executed.

```
function external_call(address destination,
                       uint value, uint dataLength,
                       bytes memory data) internal returns (bool) {
    bool result;
    assembly {
        let x := mload(0x40)
        let d := add(data, 32)
        result := call(
            sub(gas, 34710),
            destination,
            value,
            d,
            dataLength,
            x,
            0
        )
    }
    return result;
}
```

Figure 15.2: The `external_call` method that does not contain a validity check.

Exploit Scenario
A user of the multisignature wallet may submit transactions to a self-destructed contract without knowing of its destruction. The transaction will eventually get executed in `executeTransaction`, and the expected behavior of the wallet will not occur, which can cause unintentional bugs.

Recommendation
Short term, add two checks: one in `addTransaction` to prevent the creation of bad transactions, and one in `executeTransaction` or `external_call` to prevent the race condition of the target contract's self-destruct.

Long term, monitor issues discovered with `MultiSigWallet` and any other code imported from external sources.

16. Time locking can overflow

Severity: Low
Type: Data Validation
Target: `MultiSigWalletWithTimeLock.sol`

Difficulty: High
Finding ID: TOB-AZT-016

Description
The `MultiSigWithTimeLock` contract supports a time locking feature with a wallet settable time lock. However, the time lock check is an unbounded addition which can overflow and cause the check to always pass.

```
modifier pastTimeLock(uint256 transactionId) {
    require(
        block.timestamp >= confirmationTimes[transactionId] + secondsTimeLocked,
        "TIME_LOCK_INCOMPLETE"
    );
    _;
}
```

Figure 16.1: The `pastTimeLock` method in the `MultiSigWithTimeLock` contract.

Exploit Scenario
Alice and Eve are owners of the `MultiSigWalletWithTimelock`. Alice wishes to set the time lock to be unlocked at a date which is virtually never going to be encountered, allowing the wallet to be abandoned. In this case, Alice submits a transaction to execute changeTimeLock with a large number. Eve knows this will cause an overflow, allowing immediate execution of previously locked transactions. Eve is now able to execute previously locked transactions.

Recommendation
Use `SafeMath` to avoid overflow on sensitive mathematical operations.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal

	implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. DeepState Testing Enhancements

DeepState provides an interface for symbolic execution and fuzzing engines through a unit test-like structure. As part of this engagement, Trail of Bits integrated this framework with AZTEC's pre-existing C++ trusted MPC setup test suite to greatly enhance test coverage. DeepState incorporates support for exhaustive input testing and test case generation, and includes auxiliary tools to help triage results. AZTEC can use DeepState to incorporate greater security guarantees into their testing and continuous integration process.

```
/* Streaming_ReadG1ElemsToBuffer
 *
 *      Tests reading arbitrary buffer input to G1 elements
 *      and then comparing to newly written output buffer.
 */
TEST(Streaming, ReadG1ElemsToBuffer)
{
    constexpr size_t N = 10;
    constexpr size_t element_size = sizeof(Fq) * 2;
    constexpr size_t buffer_size = N * element_size;

    libff::init_alt_bn128_params();

    std::vector<G1> result;
    std::vector<G1> elems;
    result.reserve(N);
    elems.reserve(N);

    // buffers for reading/writing G1 elements
    char out_buffer[buffer_size];
    char * buffer = DeepState_CStrUpToLen(buffer_size);
    LOG(TRACE) << "Input buffer :" << buffer <<
        " of size: " << buffer_size;

    // read from elements out of buffer
    streaming::read_g1_elements_from_buffer(elems, buffer, element_size);
    for (size_t i = 0; i < N; i++)
    {
        elems[i].to_affine_coordinates();
        result.emplace_back(elems[i]);
    }
    streaming::write_g1_elements_to_buffer(result, out_buffer);

    ASSERT(memcmp(buffer, out_buffer, buffer_size))
        << "Input buffer: " << buffer << " is not equal to output buffer: " << out_buffer;
}
```

Figure B.1: An example unit test for checking correctness in validating G1 elements read and written to a buffer using helpers from `setup-tools/src/aztec_common/streaming_g1.cpp`.

DeepState evaluates defined security properties with industry-standard program analysis tools, including AFL, libFuzzer, Honggfuzz, Eclipser, Angora, Angr, and Manticore. Our integration work was focused primarily on the fuzz testing capabilities of DeepState, and

used them to test AZTEC protocol's cryptographic primitives and multi-precision arithmetic operations.

Build and Setup

We converted AZTEC's GTest suite in `setup-tools`. We integrated DeepState with CMake to create a `deepstate_tests` binary alongside the GoogleTest `setup_tests` binary. This allowed us to test on local and Docker environments while compiling with fuzzer-based compilers.

To set up a local build in order to run the DeepState tests:

```
# inside Setup/src/setup-tools
$ mkdir build && cd build/
$ cmake -DSIMULATE_PARTICIPANT=ON ..
$ make

# runs all the available DeepState tests concretely
$ ./test/deepstate_tests

# run a specific test with input file
$ ./test/deepstate_tests --input_test_file some_input
--input_which_test Streaming_ReadG1ElemsFromBuffer
```

This build setup also allows us to build the tests with compile-time instrumentation using our fuzzer-based compilers of choice:

```
$ CC=afl-gcc CXX=afl-g++ cmake -DSIMULATE_PARTICIPANT=ON ..
# ..or with honggfuzz:
$ CC=hfuzz-gcc CXX=hfuzz-g++ cmake -DSIMULATE_PARTICIPANT=ON ..
```

Not only are we still able to run our tests with concrete input like we do with GoogleTest, we can also invoke our aforementioned fuzzer executors in exhaustive test inputs, and perform crash replay. For example, here we can call our AFL executor after instrumenting our build with AFL:

```
# output by default stored in {FUZZER}_out
$ deepstate-afl -i my_seeds --which_test
Streaming_ReadG1ElemsFromBuffer ./test/deepstate_tests

# replay any crashes generated in output directory for specific test
$ ./test/deepstate_tests --input_test_files_dir AFL_out/crashes
--input_which_test Streaming_ReadG1ElemsFrombuffer
```

We encourage you to use our diverse set of fuzzer executors to test a variety of heuristics. This also includes an auxiliary ensembler fuzzing tool, which provides support for parallel and ensemble-based fuzzing with our pre-existing executors.

Test Cases

Trail of Bits converted the existing GTest suite into DeepState tests, and added new test cases to improve the potential for discovering unexpected behaviors and maintaining high code coverage. These tests vary from both lower-level operations to higher-level MPC setup and verification.

Existing unit tests were converted to `Boring_*` tests, which do not provide support for exhaustive input testing, but are instead used with concrete test vectors to verify function correctness.

```
/* Streaming_BoringWriteBigIntToBuffer
 *
 *      Concrete test that checks and verifies concrete
 *      bignum values and their resultant endianness.
 */
TEST(Streaming, BoringWriteBigIntToBuffer)
{
    libff::bigint<4> input;

    // generate bigints with concrete ulong vectors
    input.data[3] = (mp_limb_t)0xffeeddccbbaa9988UL;
    input.data[2] = (mp_limb_t)0x7766554433221100UL;
    input.data[1] = (mp_limb_t)0xf0e1d2c3b4a59687UL;
    input.data[0] = (mp_limb_t)0x78695a4b3c2d1e0fUL;

    // write bigints to buffer
    char buffer[sizeof(mp_limb_t) * 4];
    streaming::write_bigint_to_buffer<4>(input, &buffer[0]);

    // cast buffer to libgmp bignum types.
    mp_limb_t expected[4];
    expected[0] = *(mp_limb_t *)(&buffer[0]);
    expected[1] = *(mp_limb_t *)(&buffer[8]);
    expected[2] = *(mp_limb_t *)(&buffer[16]);
    expected[3] = *(mp_limb_t *)(&buffer[24]);

    // compare output with original inputs with flipped endianess
    ASSERT_EQ(expected[3], (mp_limb_t)0x8899aabbccddeeffUL);
    ASSERT_EQ(expected[2], (mp_limb_t)0x0011223344556677UL);
    ASSERT_EQ(expected[1], (mp_limb_t)0x8796a5b4c3d2e1f0UL);
    ASSERT_EQ(expected[0], (mp_limb_t)0x0f1e2d3c4b5a6978UL);
}
```

Figure B.2: Example `Boring` test written for validating bignums written using `streaming::write_bigint_to_buffer`.

Each original `Boring` test is also supplemented with a test case for actual program analysis, as it incorporates our `DeepState_*` input methods to read from our executors. The `Boring` test in Figure B.2 also has the following test optimal for fuzz testing:

```
/* Streaming_WriteBigIntToBuffer
 *
 *      Tests arbitrary input as bignum values and checks
 *      for resultant endianness when reconverted to a libgmp bignum.
 */
TEST(Streaming, WriteBigIntToBuffer)
{
    libff::bigint<1> input;
    mp_limb_t expected[1];

    // generate input value casted to unsigned long
    unsigned long bigint_in = (unsigned long) DeepState_UInt64();
    input.data[0] = (mp_limb_t) bigint_in;
    LOG(TRACE) << "Unsigned long input: " << bigint_in;
    ASSERT_EQ(input.as_ulong(), bigint_in)
        << input.as_ulong() << " does not equal input " << bigint_in;

    // write bigint to buffer, store in libgmp output
    char buffer[sizeof(mp_limb_t)];
    streaming::write_bigint_to_buffer<1>(input, &buffer[0]);
    expected[0] = *(mp_limb_t *)(&buffer[0]);

    // compare ulong input with libgmp output, with swapped endianess
    ASSERT_EQ(input.as_ulong(), __builtin_bswap64(expected[0]))
        << input.as_ulong() << " does not equal " << __builtin_bswap64(expected[0]);
}
```

Figure B.3: Corresponding `streaming::write_bigint_to_buffer` test for validating bignums.

Our current tests cover:

- Serialization of a buffer and multi-precision integers
- Serialization and deserialization of curve and field elements to buffer or file
- Serialization and deserialization of manifests into transcript files
- Computing polynomial for range proof
- Verification key validation
- Transcript and manifest validation
- Re-implementation of higher-level entry points as tests

Coverage Measurement

In addition to integrating these test cases for continuous assurance with automated testing, we have also investigated how they have improved overall code coverage in the MPC setup codebase. We have provided support for coverage measurement using GNU's gcov, which works well out-of-the-box with CMake and its CTest functionality for running

unit tests. In addition, we visualized our coverage output from gcov with gcovr, which generated HTML reports that summarized our measurement findings.

In order to enable coverage measurement with the trusted setup codebase, we have provided support for options to use with CMake to properly collect coverage information:

```
# compile our tests with the necessary flags to generate GCOV data
files
$ cmake -DGENCOV=ON DSIMULATE_PARTICIPANT=ON ..
$ make

# run GTest coverage reporting, and store generated *.gcov files in
cov_out/
$ make cov
$ cd ..
$ gcovr -r . --html --html-details -o
/path/to/gtest_report/coverage.html

# run DeepState coverage reporting, and store generated files in
dcov_out
$ cd build/ && make dcov
$ cd ..
$ gcovr -r . --html --html-details -o
/path/to/deep_report/coverage.html
```

Once gcovr executes, coverage results will be stored in HTML files in the specified output path. Opening `coverage.html` will show a report of line- and branch-based coverage for relevant code files and library dependencies, as well as a cumulative report for all the files.

Tables B.1 and B.2 show results from our coverage measurement efforts, reporting both line and branch coverage. They depict significant improvements for several libraries within the codebase, including increases in `aztec_common`, which provides several notable functions for serialization and deserialization for both cryptographic types and actual transcripts. We also extended tests to cover previously uncovered code paths, including operations in generator polynomial computation, range proof validation, and actual transcript setup in `src/setup/*.cpp`.

File	Original GTest Coverage	DeepState Coverage
src/range/range_multi_exp.hpp	41.2%	100.0%
src/generator/compute_generator_polynomial.tcc	0.0%	100.0%

src/aztec_common/streaming.cpp	45.8%	100.0%
src/aztec_common/streaming_transcript.cpp	56.9%	94.7%
src/setup/setup.cpp	0.0%	70.0%
src/setup/utils.hpp	44.8%	98.3%

Table B.1: Line-based coverage improvement for largely untested files from the `setup-tools/` codebase.

File	**Original GTest Coverage**	**DeepState Coverage**
src/range/range_multi_exp.hpp	20.0%	45.0%
src/generator/compute_generator_polynomial.tcc	0.0%	46.2%
src/aztec_common/streaming.cpp	24.4%	54.9%
src/aztec_common/streaming_transcript.cpp	28.0%	41.5%
src/setup/setup.cpp	0.0%	32.4%
src/setup/utils.hpp	28.6%	78.6%

Table B.2: Branch-based coverage improvement for largely untested files from the `setup-tools/` codebase.

Continuous Integration

We recommend adopting the following procedure to maintain continuous assurance of the trusted setup codebase during the development cycle. This is crucial to catch bugs, maintain a high level of code coverage, and validate that functionality is preserved with no breaking changes.

Our recommended procedure is as follows:

1. For every test, an initial fuzzing campaign should be completed in order to build up an initial input corpora per test.
2. Once new changes are added to the codebase, ensure that any corresponding `Boring` tests continue to pass, validating both its functionality and whether it is still deterministically evaluating to the correct output.
3. Run a targeted fuzzing campaign for the test for at least 24 hours, periodically verifying that inputs are reaching new states and that new seeds are being generated.
4. Update the initial corpus with any newly generated seeds.

Over time, the built up sets of corpora for each test will represent thousands of CPU hours of refinement, and can be used not just for program analysis tools, but also for fast validation with DeepState besides the `Boring` tests:

```
# will run every single generated test case from my corpus against the
specified test
$ ./test/deepstate_tests --input_file_dirs my_corpus/
--input_which_test Streaming_WriteReadValidateTranscripts
```

Store the generated corpora in an access-controlled location, since they are potentially crashing inputs and may pose a risk if discovered and used by outside parties.

Finally, consider integrating with Google's oss-fuzz, which provides automatic fuzz testing using Google's extensive testing infrastructure. oss-fuzz, which supports fuzzing a variety of languages with AFL and LibFuzzer, provides the benefit of triaging and reporting bugs without further human intervention. It's also able to re-configure fuzz tests to work with new upstream changes.

C. Documentation Discrepancies

Trail of Bits discovered issues in the documentation supplied by AZTEC throughout the course of the assessment. These issues took the form of both inconsistencies between the documentation and implementation and omissions in the documentation. We recommend that the documentation reflect the implementation as accurately as possible, as this is where the majority of our cryptographic findings arose. This could also be a source of confusion for any users implementing the AZTEC protocol. We document these issues here.

Inconsistencies

- In the AZTEC whitepaper, the JoinSplit algorithm generates random values $(x_0,...,x_n)$, where x_0 is the output of a hash function taken over the input commitments, and the subsequent x_i values are equal to x_0^i. In the implementation, x_0 is computed in the same manner; however, the subsequent x_i values are equal to $H^i(x_0)$, where $H^i(-)$ represents the hash function `H` nested `i` times.
- In the AZTEC whitepaper, pairing checks are used to validate commitments (γ,σ). Specifically, the whitepaper calls for checking that $e(\gamma,t_2) = e(\sigma,g_2)$. However, the implementation actually checks that $e(\gamma,t_2)\cdot e(\sigma^{-1},g_2)=1$.
- In the AZTEC whitepaper, the language used to describe the trusted setup parameters seems to be inconsistent with the actual trusted setup protocol given in the separate documentation. Specifically, the whitepaper claims that `h` and `y` are both chosen uniformly randomly. However, in the actual protocol, `h` depends on `y`, as it is computed by evaluating a polynomial of `y` in the exponent of `g`. It is therefore misleading to say that `h` is chosen uniformly randomly from `G`.

Omissions

- The documentation concerning the trusted setup protocol mentions that each participant's transcripts need to be individually verified. However, they mention no mechanism for each participant to authenticate themselves and their transcript. In the trusted setup implementation, each user generates a random Ethereum account and uses this account to sign their transcript hash as authentication. This information should be included in the documentation with a corresponding security proof.
- As addressed in TOB-AZT-008, verification fails if the random value generated during verification is either zero or one. This information should also be reflected in the documentation.
- The whitepaper presents a JoinSplit protocol and an optimized version of that same protocol. After presenting the optimized protocol, a justification for its security is given, but there's no formal proof of security.

D. Solidity Testability

Trail of Bits has developed Echidna, an EVM fuzzer and we strongly recommend its use on AZTEC's Solidity code. Echidna will automatically generate test cases for the exposed ABI of the Solidity contract that attempt to violate defined security properties.

For example, Echidna would find the following contract falsifiable with two sufficiently large `uint256` values. Methods that begin with `echidna_*` easily define security properties.

```
contract SafeAdd {
  bool failed = false;
  function try_add(uint256 a, uint256 b) public {
    uint256 c = a + b;
    if (c < a || c < b) {
      failed = true;
    }
  }
  function echidna_add() public returns (bool) {
    return !failed;
  }
}
```

Figure D.1: An example contract.

To demonstrate its use, we wrote an Echidna property for TOB-AZT-009. Echidna is able to fuzz the exposed ABI and verify that none of the functions we can call with the contract are able to set the admin address to 0. It is a very short contract, detailed below.

```
pragma solidity ^0.5.0;
import './AdminUpgradeabilityProxy.sol';
contract AdminUpgradeTest is AdminUpgradeabilityProxy {
  constructor() AdminUpgradeabilityProxy(address(this), address(this), "") public { }
  function echidna_admin() public returns (bool) {
    return _admin() != address(0);
  }
}
```

Figure D.2: An Echidna test validating that `address(0)` can never be set.

By running `echidna-test AdminUpgradeTest.sol`, a developer can obtain reasonable assurance that the `AdminUpgradeabilityProxy` contract cannot be forced into an invalid admin state by a malicious user.

The design of AZTEC's Solidity contracts impedes further testing via Echidna:

1. There is no Solidity utility to generate proofs. The AZTEC codebase has high unit test coverage in JavaScript. However, higher confidence in the security of the contracts could be achieved by writing an Echidna test that can assert that every valid proof can be validated, not just the examples in the unit test suite. Because the proofs are generated in JavaScript and not Solidity, we cannot use Echidna to generate several proofs to be tested.
2. There is no valid Solidity ABI to test against. Echidna uses the ABI output of the Solidity compiler to infer the signatures of contract functions. While an ABI specification is provided for each validator contract via the interface contracts, the parameter format does not allow for direct "plug & play" fuzzing. Each validator requires the calldata in a specific format, and since the fuzzer does not have access to this format, it will only produce random invalid bytes. It is possible to write a public ABI that will assert that most proofs do not validate because each calldata specification is static, but this is significantly less interesting.

AZTEC should add property-based testing to their Solidity contract. Unit tests catch cases that have already been considered, but property tests discover edge cases that nobody has thought to consider. With a Solidity proof generator contract, such testing will become more feasible and enable greater assurance that contracts adhere to specifications.