AZTEC

Security Assessment

October 9th, 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

From August 26th through September 6th, 2019, AZTEC engaged with Trail of Bits to review the security of the AZTEC trusted setup protocol. Trail of Bits conducted this assessment over the course of four person-weeks with two engineers using commit hash 230a1d8a for the AztecProtocol/Setup repository.

The audit 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).

Over the course of the audit, Trail of Bits discovered two high-severity issues. These issues (TOB-AZT-007 and TOB-AZT-008) have a high difficulty of exploitation as they rely on low-probability events; however, if they were exploited, it would be very dangerous to the AZTEC protocol. In addition, five low-severity issues were reported that pertain to AWS configurations (TOB-AZT-002 through TOB-AZT-006).

We recommend that AZTEC be vigilant in their cryptographic implementation, even for low-probability events, as these can still be dangerous. We also encourage AZTEC to familiarize themselves with AWS configuration best practices. Lastly, we recommend that AZTEC integrate our tooling, DeepState, to continue to expand test coverage.

Trail of Bits reviewed the fixes proposed by AZTEC for the issues presented in this report. These fixes were either implemented or their risk was accepted. For more details on the review of AZTEC's fixes, see Appendix D.

Project Dashboard

Application Summary

Name	AZTEC
Version	AztecProtocol/Setup commit: 230a1d8a
Type	C++, TypeScript

Engagement Summary

Dates	August 26th through September 6th 2019
Method	Whitebox
Consultants Engaged	2
Level of Effort	4 person-weeks

Vulnerability Summary

Total High-Severity Issues	2	■■
Total Low-Severity Issues	5	■■■■■
Total	7	

Category Breakdown

Configuration	5	■■■■■
Cryptography	2	■■
Total	7	

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 aimed to reduce the complexity of constructing and verifying proofs. Specifically, the complexity of AZTEC's range proofs does not scale with the size of the range. However, this achievement requires 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 AZTEC's `setup-tools` and `setup-mpc-server`. The assessment included both manual and dynamic analyses, and used DeepState integration to perform dynamic test coverage of the C++ portions of the codebase.

Specifically, we sought to answer the following questions:

- Is the multi-party computation protocol specified in the documentation supplied by AZTEC cryptographically secure?
- Does the implementation of the multi-party computation protocol in `setup-tools` and `setup-mpc-server` comply with the documentation?
- If the implementation does not comply with its corresponding documentation, does this introduce any vulnerabilities?
- Can we use dynamic testing to detect unsafe behavior in both `C++`?

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.

Recommendations Summary

Short Term

❑ **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 the trusted setup. TOB-AZT-008

Long Term

❑ **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

❑ **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

Findings Summary

#	Title	Type	Severity
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

2. AWS filesystem encryption is not configured

Severity: Low
Difficulty: Low
Type: Configuration
Finding ID: TOB-AZT-002
Target: `setup-mpc-server\terraform\main.tf`

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
Difficulty: Low
Type: Configuration
Finding ID: TOB-AZT-003
Target: `setup-mpc-server\terraform\main.tf`

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
Difficulty: Low
Type: Configuration
Finding ID: TOB-AZT-004
Target: `setup-iac\main.tf`

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
Difficulty: Low
Type: Configuration
Finding ID: TOB-AZT-005
Target: `setup-iac\main.tf`

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
Difficulty: Low
Type: Configuration
Finding ID: TOB-AZT-006
Target: `setup-iac\main.tf`

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
Difficulty: High
Type: Cryptography
Finding ID: TOB-AZT-007
Target: `verifier.cpp`

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
Type: Cryptography
Target: `setup/setup.cpp, verify/verifier.cpp`

Difficulty: High
Finding ID: TOB-AZT-008

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`.

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.

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

Long term, add 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`.

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 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

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

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.

D. Fix Log

Trail of Bits reviewed the fixes proposed by AZTEC for the issues presented in this report. The fixes submitted by AZTEC can be found at the following locations:

- *TOB_AZT_003*: https://github.com/AztecProtocol/Setup/blob/master/setup-mpc-server/terraform/main.tf
- *TOB_AZT_004*: https://github.com/AztecProtocol/Setup/blob/master/setup-iac/main.tf
- *TOB_AZT_005*: https://github.com/AztecProtocol/Setup/blob/master/setup-iac/main.tf
- *TOB_AZT_006*: https://github.com/AztecProtocol/Setup/blob/master/setup-iac/main.tf
- *TOB_AZT_007*: Fixed via manual review of signatures after ceremony performed
- *TOB_AZT_008*: https://github.com/AztecProtocol/Setup/commit/e5a2eaa7b2300a71de75b847956bc733880b2407

AZTEC provided fixes for all of our issues, except for TOB_AZT_002. We report all issues to be fixed, except for TOB_AZT_004 and TOB_AZT_006, which we marked as open and partially fixed, respectively. We wish to discuss both of these issues, as well as additional issues within the Terraform descriptions, with AZTEC in the near future.

#	Title	Severity	Status
2	AWS file system encryption is not configured	Low	Risk Accepted
3	AWS instance is associated with a public IP address	Low	Fixed
4	AWS ALB load balancer allows plaintext traffic	Low	Open
5	AWS ALB load balancer has access logs disabled	Low	Fixed
6	AWS security group allowing all traffic exists	Low	Partial
7	Failure to validate MPC output can lead to double spending attack	High	Fixed
8	Random value generated can be zero	High	Fixed

Detailed Fix Log

Finding 2: AWS file system encryption is not configured.
Risk Accepted.

Finding 3: AWS instance is associated with a public IP address.
Fixed; the AZTEC team removed the `associate_public_ip_address` *attribute from the AMI configuration.*

Finding 4: AWS ALB load balancer allows plaintext traffic.
Open; however, the traffic is only used for redirect. Whilst HTTP downgrade attacks can exist, other mitigating controls such as HTTP Strict Transport Security (HSTS).

Finding 5: AWS ALB load balancer has access logs disabled.
Fixed; access logs are now enabled within the terraform configuration.

Finding 6: AWS security group allowing all traffic exists.
Partial; components have been moved with a VPC communication limited through a gateway. However, arbitrary communication to the upstream NAT gateway is still permissible. Review by the assessment team shows that this VPC-segmentation should be sufficient as a mitigating control to the original issue.

Finding 7: Failure to validate MPC output can lead to double spending attack.
Fixed. The AZTEC team stated that:

> *This will be verified by checking Boneh-Boyen signatures for 1 after ceremony.*

Finding 8: Random value generated can be zero.
Fixed.