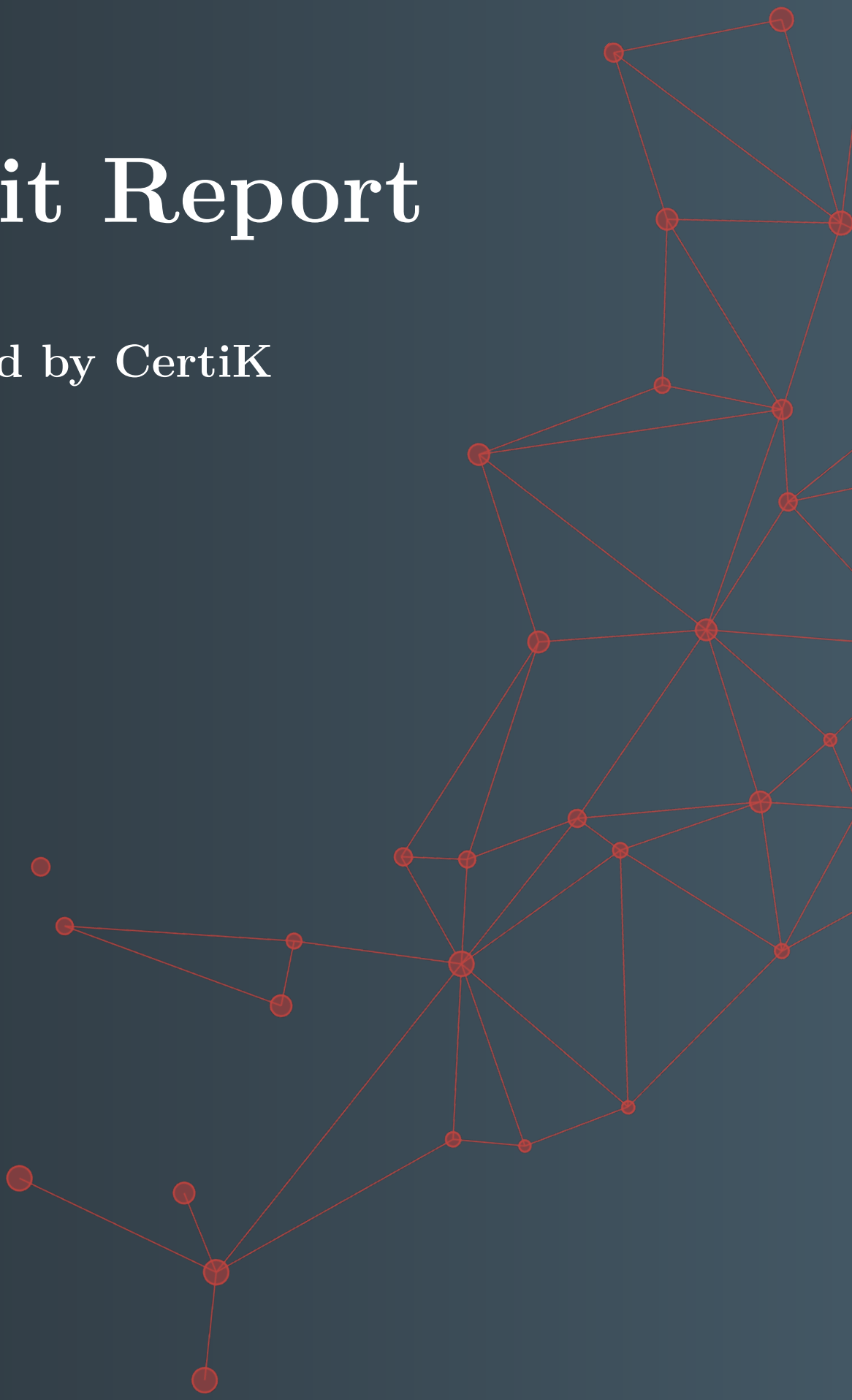
CERTIK

Audit Report

Produced by CertiK

for Torus



Contents

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and Torus (the "Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the "Agreement").

About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK's mission of every audit is to apply different approaches and detection methods, ranging from manual, static, and dynamic analysis, to ensure that the project is checked against known attacks and potential vulnerabilities. CertiK leverages a team of seasoned engineers and security auditors to apply testing methodologies and verifications on the project, in turn creating a more secure and robust software system.

CertiK has served more than 100 clients with high quality auditing and consulting services, ranging from stablecoins such as Binance's BGBP and Paxos Gold to decentralized oracles such as Band Protocol and Tellor.

Executive Summary

Torus is a user-friendly, secure, and non-custodial key management system for DApps. The Distributed Key Generation protocol allows more nodes to participate in the process, hence prevents centralized one point of failure. To that end, the sole objective of the audit is to verify Torus' implementation of the DKG against the provided specifications. A series of thorough security assessments have been carried out, the goal of which is to help Torus protect their users by finding and fixing known vulnerabilities that could cause unauthorized access, loss of funds, cascading failure, and/or other vulnerabilities. Alongside each security finding, recommendations on fixes and best practices have also been given.

Testing Summary

SECURITY LEVEL

DKG Audit

This report has been prepared as a product of the DKG Audit request by Torus.

This audit was conducted to discover issues and vulnerabilities in the source code of DKG package implementation.

TYPE	DKG Implementation
SOURCE CODE	github.com/torusresearch/torus-public
LANGUAGE	Golang
REQUEST DATE	June 16, 2020
REVISION DATE	July 13, 2020
METHODS	A comprehensive examination has been performed using Whitebox Analysis. In detail, Dynamic Analysis, Static Analysis, and Manual Review were utilized.

Review Notes

Overview

A primary focus for the audit is to have a thorough look at the messages processing functions of the package. Specifically we analyze how the keygen nodes are defined and how their state changes are triggered by messages, the goal of which is to check the implementation against the specs and therefore minimize the possibilities of unintentional state behaviors taking place.

We inspected every module within the scope to ensure that:

1. The message routes the message to the correct processing function.
2. The functions process corresponding messages correctly according to the scope.
3. The messages are sent to the correct nodes.

Scope of Work

- The audit work was scoped to a specific commit 51b9965d of the source code per the agreement
- The code was verified against the specifications and literature provided by the client. The most prominent sources are:
    - Secure Distributed Key Generation for Discrete-Log Based Cryptosystems.
    - Distributed Key Generation in the Wild.

    - AVSS and PSS
- Files within the scope include: keygen_nofsm.go, keygen_nofsm_test.go, p2p_messages.go, types.go
- Node state transitions in each function were carefully verified against their specification
- Go programming best practices were enforced to improve general performance and minimize the chances of run-time panicking

Audit Summary

In total we found one minor issue and several other smaller shortcomings that Torus' developers should be aware of. Overall, we found Torus' smart contracts follow best practices and the implementation of DKG closely adheres to the provided specifications, whose goal is to decentralize the key generation operation. Moreover other desirable properties of DKG are also achieved, namely:

- Liveness: all wait states that a node enters are eventually satisfied.
- Correctness: all honest nodes decide on the same value of selected sets of AVSS.
- Efficiency: the overall DKG has uniformly bounded communication complexity.
- Secrecy: no malicious node can compute the private key, otherwise it would break the discrete algorithm.

Audit Revisions

On 16.07.2020 the Torus developer team has updated their codebase in the commit cd85d2a. We really appreciate the team's quick reaction and communication. After additional review we conclude that all issues listed in this report have been correctly alleviated following our recommendations.

Message Processing Evaluation

Messages handlers

Function	Description	PASS
processShareMessage	• Initialize the whole DKG process as the dealer node • Create the secrets S and Sprime • From the secrets choose two random bivariate polynomials F and Fprime, whose slices are then sent to other node • Set keygen Phase to started • Create and send Send messages	✔
processSendMessage	• Make sure that the incoming message is from the correct dealer node • Verify the polynomials from the dealer node by AVSSVerifyPoly • Compute points on the polynomials then send them in Echo messages to other nodes	✔
processEchoMessage	• Verify the incoming points by AVSSVerifyPoint • If the number of Echo messages is sufficient interpolate the points to get polynomials • Compute points on these polynomials then send them in Ready messages to other nodes	✔
processReadyMessage	• Verify incoming points by AVSSVerifyPoint • If the number of Ready messages is sufficient create and send the Complete message.	✔
processCompleteMessage	• If the number of complete messages is sufficient broadcast the propose message	✔

processNIZKPMessage	• Verify the share commitment with VerifyShareCommitment • Verify the zero-knowledge proof with VerifyNIZKPK • Broadcast the Pubkey message	✔

Messages routers

FUNCTION	Description	PASS
ProcessMessage	• Retrieve relevant information from node storage • Route node-to-node message to relevant function to process	✔
ProcessBroadcastMessage	• Retrieve relevant information from node storage • Route broadcasted message to relevant function to process	✔

Messages verification

FUNCTION	Description	PASS
AVSSVerifyPoly	• Verify that the incoming polynomials from the dealer • Utilize a commitment scheme and homomorphicity of the encoding function, i.e. exponentiation in the group of elliptic curve	✔
AVSSVerifyPoint	• Verify that the incoming points from other nodes • Utilize a commitment scheme and homomorphicity of the encoding function, i.e. exponentiation in the group of elliptic curve	✔

VerifyShareCommitment	• Verify that the mixed public key forms a valid share with respect to D bar.	✔
VerifyNIZKPK	• Using zero-knowledge proof to verify that the node has a valid secret share without revealing the private key.	✔

Audit Findings

Exhibit 1

TITLE	TYPE	SEVERITY	LOCATION
Repeated “Share” message	Security	Minor	p2p_messages.go line 21

Description:

The “Share” message which initializes the key generation operation is meant to be sent from the node to itself only once. However a malicious node can try to recreate the content of the “Share” message and send it to the intended dealer node. It then triggers the random generation of secrets S, S0, bivariate polynomials f, f0 and overwrites the earlier ones.

Recommendations:

Put a check in the function to make sure the incoming “Share” message has not been processed.

Alleviation:

There is an external check for this situation outside of the keygen package, the audited scope, but we think it is definitely better for that check to also be inside the package.

Exhibit 2

TITLE	TYPE	SEVERITY	LOCATION
Threshold number of “Complete” messages	Implementation	Informational	p2p_messages.go line 511

Description:

In the function processCompleteMessage() the number of “Complete” messages is checked at line 511 and has to be a certain threshold. In the code this threshold is K + T , but in the specification it is K.

Recommendations:

Either change K + T to K in the code or K to K + T in documentation.

Alleviation:

For the K vs K + T threshold, either is fine since there are K + T honest nodes so it always hits both thresholds. Using K as the threshold will let it potentially complete slightly faster (depending on the variance between completion rates of the individual AVSSs), whereas using K + T increases the number of nodes that contribute to the initial randomness. We will update the spec to use K + T since that's what we intend to continue using.

Exhibit 3

TITLE	TYPE	SEVERITY	LOCATION
Repeated “Send” message	Security	Informational	p2p_messages.go line 89

Description:

In the specification the "Send" message from a certain dealer should only be processed once, but neither `processSendMessage()` nor `ProcessMessage()` checks for repeated message in the code. We believe this cannot be exploited in any meaningful way other than unnecessary repeated execution.

Recommendations:

Put a check in the function to make sure the incoming “Send” message has not been processed.

Alleviation:

The developer has updated the code to follow our recommendation in a newer commit.

Exhibit 4

TITLE	TYPE	SEVERITY	LOCATION
"Echo" message counter	Error handling	Informational	p2p_messages.go line 235

Description:

In the function `processEchoMessage()` the "Echo" message counter `EC` is increased at line 235. This value is then checked in the subsequent if condition at line 236, whose branch can fail at line 258 and line 284, so the "Ready" messages will not be generated and sent. Next time the function `processEchoMessage()` is triggered, the value `EC` will be increased by 1 again, hence the if condition will never be satisfied and the node would be left out of the whole process.

Recommendations:

Make sure to decrease the counter `EC` when an error occurs.

Alleviation:

If the function `processEchoMessage()` exits prematurely it means the node malfunctions, so it is OK that it is left out during the current key generation operation.

Exhibit 5

TITLE	TYPE	SEVERITY	LOCATION
"Ready" message counter	Error handling	Informational	p2p_messages.go line 235

Description:

In the function `processReadyMessage()` the "Ready" message counter `RC` is increased at line 387. This value is then checked in the subsequent if condition at line 388, whose branch can fail at lines 411, 437, so the "Ready" messages will not be generated and sent, or in if condition at line 451, whose branch can fail at line 471, so the "Complete" message will not be generated and sent. Next time the function `processReadyMessage()` is triggered, the value `RC` will be increased by 1 again, hence the if condition will never be satisfied and the node would be left out of the whole process.

Recommendations:

Make sure to decrease the counter `RC` when an error occurs.

Alleviation:

If the function `processReadyMessage()` exits prematurely it means the node malfunctions, so it is OK that it is left out during the current key generation operation.

Exhibit 6

TITLE	TYPE	SEVERITY	LOCATION
Unnecessary loop execution	Ineffectual Code	Informational	p2p_messages.go lines 252-260, 405-413

Description:

In the function `processEchoMessage()` the piece of code at lines 252-260 is not related to `newNode`, loop iterator, hence can be put in front of the loop for more efficient execution.

In the function `processReadyMessage()` the piece of code at line 405-413 is not related to `newNode`, loop iterator, hence can be put in front of the loop for more efficient execution.

Recommendations:

Put the aforementioned code in front of the loops.

Alleviation:

The developer team has updated the code to follow our recommendation in a newer commit.