



PALADIN
BLOCKCHAIN SECURITY

Smart Contract Security Assessment

Final Report

For Suzaku

26 November 2024



paladinsec.co



info@paladinsec.co

Table of Contents

Table of Contents	2
Disclaimer	3
1 Overview	4
1.1 Summary	4
1.2 Contracts Assessed	4
1.3 Findings Summary	5
1.3.1 AvalancheICTTRouter	6
1.3.2 AvalancheICTTRouterFixedFees	6
1.3.3 Interfaces (IAvalancheICTTRouter, AvalancheICTTRouterFixedFees)	7
2 Findings	8
2.1 AvalancheICTTRouter	8
2.1.1 Privileged Functions	8
2.1.2 Issues & Recommendations	9
2.2 AvalancheICTTRouterFixedFees	23
2.2.1 Privileged Functions	23
2.2.2 Issues & Recommendations	24
2.3 Interfaces (IAvalancheICTTRouter, IAvalancheICTTRouterFixedFees)	27
2.3.1 Issues & Recommendations	27

Disclaimer

Paladin Blockchain Security ("Paladin") has conducted an independent audit to verify the integrity of and highlight any vulnerabilities or errors, intentional or unintentional, that may be present in the codes that were provided for the scope of this audit. This audit report does not constitute agreement, acceptance or advocacy for the Project that was audited, and users relying on this audit report should not consider this as having any merit for financial advice in any shape, form or nature. The contracts audited do not account for any economic developments that may be pursued by the Project in question, and that the veracity of the findings thus presented in this report relate solely to the proficiency, competence, aptitude and discretion of our independent auditors, who make no guarantees nor assurance that the contracts are completely free of exploits, bugs, vulnerabilities or deprecation of technologies. Further, this audit report shall not be disclosed nor transmitted to any persons or parties on any objective, goal or justification without due written assent, acquiescence or approval by Paladin.

All information provided in this report does not constitute financial or investment advice, nor should it be used to signal that any persons reading this report should invest their funds without sufficient individual due diligence regardless of the findings presented in this report. Information is provided 'as is', and Paladin is under no covenant to the completeness, accuracy or solidity of the contracts audited. In no event will Paladin or its partners, employees, agents or parties related to the provision of this audit report be liable to any parties for, or lack thereof, decisions and/or actions with regards to the information provided in this audit report.

Cryptocurrencies and any technologies by extension directly or indirectly related to cryptocurrencies are highly volatile and speculative by nature. All reasonable due diligence and safeguards may yet be insufficient, and users should exercise considerable caution when participating in any shape or form in this nascent industry.

The audit report has made all reasonable attempts to provide clear and articulate recommendations to the Project team with respect to the rectification, amendment and/or revision of any highlighted issues, vulnerabilities or exploits within the contracts provided. It is the sole responsibility of the Project team to sufficiently test and perform checks, ensuring that the contracts are functioning as intended, specifically that the functions therein contained within said contracts have the desired intended effects, functionalities and outcomes of the Project team.

1 Overview

This report has been prepared for Suzaku on the Avalanche network. Paladin provides a user-centred examination of the smart contracts to look for vulnerabilities, logic errors or other issues from both an internal and external perspective. All dependencies have previously been audited and are not included in this report if there are no new issues.

1.1 Summary

Project Name	Suzaku
URL	
Platform	Avalanche
Language	Solidity
Preliminary Contracts	https://github.com/suzaku-network/suzaku-contracts-library/tree/8e16728491201b22ecb9b5f82e9c976bd5c031e3
Resolution	https://github.com/suzaku-network/suzaku-contracts-library/tree/f55672ffdc02270ef83aa532f6c675540c6a0792

1.2 Contracts Assessed

Contract	Live Code Match
AvalancheICTTRouter	
AvalancheICTTRouterFixedFees	
IAvalancheICTTRouter	
IAvalancheICTTRouterFixedFees	

1.3 Findings Summary

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
● Governance	0	-	-	-
● High	1	1	-	-
● Medium	3	1	1	1
● Low	6	2	1	3
● Informational	2	1	-	-
Total	12	5	2	4

Classification of Issues

Severity	Description
● Governance	Issues under this category are where the governance or owners of the protocol have certain privileges that users need to be aware of, some of which can result in the loss of user funds if the governance's private keys are lost or if they turn malicious, for example.
● High	Exploits, vulnerabilities or errors that will certainly or probabilistically lead towards loss of funds, control, or impairment of the contract and its functions. Issues under this classification are recommended to be fixed with utmost urgency.
● Medium	Bugs or issues that may be subject to exploit, though their impact is somewhat limited. Issues under this classification are recommended to be fixed as soon as possible.
● Low	Effects are minimal in isolation and do not pose a significant danger to the project or its users. Issues under this classification are recommended to be fixed nonetheless.
● Informational	Consistency, syntax or style best practices. Generally pose a negligible level of risk, if any.

1.3.1 AvalancheCTTRouter

ID	Severity	Summary	Status
01	HIGH	bridgeAmount is incorrectly reduced by primary and secondary fees in all bridge functions	✓ RESOLVED
02	MEDIUM	registerDestinationTokenBridge does not allow destination chain to be the current chain	ACKNOWLEDGED
03	MEDIUM	Users are overcharged if a token with a fee on transfer is used	PARTIAL
04	MEDIUM	Incorrect use of array length in removeDestinationChainID	✓ RESOLVED
05	LOW	Router assumes that a token will have only one remote bridge contract per chain	ACKNOWLEDGED
06	LOW	Router forces users to pay the primary fee in the same currency as the bridged token	✓ RESOLVED
07	LOW	Users can disadvantage themselves by providing insufficient gasLimit	ACKNOWLEDGED
08	LOW	Transactions can revert even when sufficient gasLimit is provided	ACKNOWLEDGED
09	LOW	All native bridge functions give more allowances to the bridge source than needed	PARTIAL
10	INFO	Typographical issues	✓ RESOLVED

1.3.2 AvalancheCTTRouterFixedFees

ID	Severity	Summary	Status
11	LOW	Primary and secondary fees set by the owner can cause the user to overpay or underpay the fees	✓ RESOLVED
12	INFO	Add a custom getter for destinationChainTokenToMinBridgeFees	

1.3.3 Interfaces

(IAvalancheCTTRouter, AvalancheCTTRouterFixedFees)

No issues found.



2 Findings

2.1 AvalancheICTTRouter

AvalancheICTTRouter is a bridge contract that enables asset transfers between Avalanche EVM L1 chains through canonical Inter-Chain Token Transfer (ICTT) contracts.

The bridge has the following features:

- Supports bridging both ERC20 tokens and native tokens
- Allows for both simple transfers and transfers with additional function calls on the destination chain
- Supports multi-hop bridging (transfers through intermediate chains)

2.1.1 Privileged Functions

- registerSourceTokenBridge
- registerDestinationTokenBridge
- removeSourceTokenBridge
- removeDestinationTokenBridge
- transferOwnership
- renounceOwnership

2.1.2 Issues & Recommendations

Issue #01	bridgeAmount is incorrectly reduced by primary and secondary fees in all bridge functions
-----------	--------------------------------------------------------------------------------------------------

Severity

 HIGH SEVERITY

Description

Primary and secondary fees incentivize relayers to process the user's message. Primary fees are transferred straight away from the router by the ICTT protocol.

The secondary fees work differently. They incentivize the relayer that processes the message (in the case of a multi-hop) from the home chain to the end remote chain destination. Because the secondary fee is charged on a different chain (the intermediary home chain) instead of the originating chain where the user initiated the router bridge transaction, the secondary fee is deducted from the bridged amount on the intermediary home chain.

As the following snippet shows, the secondary fee is transformed into a primary fee on the intermediary home chain.

```
TokenHome.sol:L532:538
else if (transferrerMessage.messageType ==
TransferrerMessageType.MULTI_HOP_SEND) {
    MultiHopSendMessage memory payload =
        abi.decode(transferrerMessage.payload,
(MultiHopSendMessage));

    (uint256 homeAmount, uint256 fee) =
_processMultiHopTransfer(
        sourceBlockchainID, originSenderAddress,
payload.amount, payload.secondaryFee
    );
```

In `bridgeERC20`, `bridgeAndCallERC20`, `bridgeNative` and `bridgeAndCallNative`, the user defines the amount they want to bridge and they or the owner defines how much the primary or the secondary fees should be.

The issue lies in the amount that is sent towards the source bridge. This amount is calculated in all functions in the following way:

```
uint256 bridgeAmount = adjustedAmount - (primaryFeeAmount +
secondaryFeeAmount);
```

Let's say the amount is 100 and both primary and secondary fees are 10.

The router takes from the user 100 and then calculates the bridge amount:

```
bridgeAmount = 100 - (10 + 10)
```

```
bridgeAmount = 80
```

When we send or `sendAndCall` we will define the bridge amount as 80. Then the source bridge will transfer from the router the primary fee (10) and then it will only check if the secondary fee is less than the bridged amount, if not it will revert. Notice it will not take the secondary fee from the router.

On the home chain, it will take the defined secondary fee (10) from the bridged amount (80) and the final bridged amount is going to be 70.

This leaves the secondary fee inside the router and there is also an outstanding allowance from the router towards the source bridge for this amount.

Recommendation	By taking into account a separate issue (that the primary fee could be in a different currency), consider <code>bridgeAmount</code> to not be reduced by the primary or the secondary fees.
-----------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Resolution



The recommendation has been adopted for `AvalancheICTTRouterFixedFees` (the `secondaryFeeAmount` is no longer deduced from the `adjustedAmount`). The issue is solved for `AvalancheICTTRouter` with the implementation of issue #06 recommendations (let the user specify the fee token).

Issue #02**registerDestinationTokenBridge does not allow destination chain to be the current chain****Severity** MEDIUM SEVERITY**Description**

ICTT multi-hop transactions flow in the following way: current subnet -> home chain (hop) -> destination subnet.

When ICTT handles multi-hop transactions, it validates the multi-hop inputs with `_validateMultiHopInput`:

[TokenRemote.sol::L659-L675](#)


```
function _validateMultiHopInput(
    bytes32 destinationBlockchainID,
    address destinationTokenTransferrerAddress,
    address multiHopFallback
) private view {
    TokenRemoteStorage storage $ = _getTokenRemoteStorage();
    if (destinationBlockchainID == $_blockchainID) {
        require(
            destinationTokenTransferrerAddress !=
            address(this),
            "TokenRemote: invalid destination token
            transferrer address"
        );
    }
    require(multiHopFallback != address(0), "TokenRemote:
    zero multi-hop fallback");
}
```

Based on the snippet above, the destination subnet can be the same as the current one only if the destination bridge is different from the one used now.

`registerDestinationTokenBridge` reverts if the destination chain id is the same as the current one, thus the router is blocking users from making multi-hop transactions to the current chain.

Recommendation

Remove the `destinationChainID == routerChainID` if-statement in `registerDestinationTokenBridge`.

Resolution ACKNOWLEDGED

The team acknowledged this issue as the Router is intended to bridge to other chains only as it relies exclusively on canonical bridges and one token can have just one canonical bridge per chain.

Issue #03**Users are overcharged if a token with a fee on transfer is used****Severity** MEDIUM SEVERITY**Description**

In the `bridgeERC20` and `bridgeAndCallERC20` functions, the fees are taken first from the amount that the user wants to bridge then a `safeTransferFrom` is executed and the amount is then adjusted if the token is a token with a fee on transfer.

```
uint256 primaryFeeAmount = (amount *  
primaryRelayerFeeBips) / 10_000;
```

```
uint256 secondaryFeeAmount = (amount *  
secondaryRelayerFeeBips) / 10_000;
```

```
uint256 adjustedAmount =  
SafeERC20TransferFrom(IERC20(tokenAddress),  
amount);
```

The issue is that the fees are taken from the initial amount which might be bigger than the actual amount that was transferred in, causing the user to pay more in fees.

Recommendation

Consider taking the fee from `adjustedAmount` instead of `amount`.

Resolution PARTIALLY RESOLVED

This issue was fixed in `AvalancheICTTRouter` but not in `AvalancheICTTRouterFixedFees`.

Within `bridgeERC20()` and `bridgeAndCallERC20()` in the `AvalancheICTTRouterFixedFees` contract, `primaryFeeAmount` is calculated based on `adjustedAmount` (the actual transferred value) but `secondaryFeeAmount` is calculated based on the input argument `amount`. It should be based on `adjustedAmount` instead of `amount`.

Issue #04**Incorrect use of array length in removeDestinationChainID****Severity** MEDIUM SEVERITY**Location**

```
function _removeDestinationChainID(address token, bytes32
chainID) internal {
    uint256 chainsNumber =
tokenToDestinationChainsIDList[token].length;
    for (uint256 i; i < chainsNumber; ++i) {
        if (tokenToDestinationChainsIDList[token][i] ==
chainID) {
            tokenToDestinationChainsIDList[token][i] =
tokenToDestinationChainsIDList[token]
[tokensList.length - 1];
            tokenToDestinationChainsIDList[token].pop();
            break;
        }
    }
}
```

Description


In the `_removeDestinationChainID` function, there is an incorrect usage of `tokensList.length` instead of the correct array length, which should be `tokenToDestinationChainsIDList[token].length`. This will most probably result in DoS when trying to remove tokens.

Recommendation

Consider correcting the array length assigned from `tokenList.length` to `tokenToDestinationChainsIDList[token].length`.

Resolution RESOLVED

This function was removed when the team changed the type of the variables from array to `EnumerableSet` when resolving a part of issue #10.

Issue #05**Router assumes that a token will have only one remote bridge contract per chain****Severity** LOW SEVERITY**Description**

ICCT allows a token to have more than one remote token transferrer. This is visible in `_registerRemote` within ICTT's [TokenHome.sol contract](#).

The remote contract registration is permissionless and anybody can register a new remote contract for a specific token.


Possible use cases of having multiple remote contracts on one chain could include having a native and a ERC20 version of the token or having the token in different decimals etc.

Recommendation

Consider modifying the codebase to support multiple source and destination bridges to be registered, removed and used.

Resolution ACKNOWLEDGED

The team commented: "We acknowledge this issue as the Router is intended to work with canonical contracts only, thus having a single way to bridge one token to a destination chain. It does not really makes sense 'having a native and an ERC20 version of the token' because if the token is the native token on the destination chain, we can use a wrapper contract to transform it to ERC20."

Issue #06**Router forces users to pay the primary fee in the same currency as the bridged token****Severity** LOW SEVERITY**Description**

ICCT allows users to define the `primaryFeeTokenAddress` and this token can be different from the bridged token amount.


With the current implementation, the users of the router cannot use another currency for the primary fee.

Recommendation

Consider allowing users to pay the primary fee with a different currency than the bridged one in both ERC20 and Native bridge functions.

Resolution RESOLVED

The recommendation has been adopted for `AvalancheICTTRouter` (a new input was added to the bridge functions: `primaryFeeTokenAddress`. This allows users to select the token they want to pay the primary fee). For `AvalancheICTTRouterFixedFees`, the same token (or its wrapped version) is used as `primaryFeeToken`—this is intended.

Issue #07**Users can disadvantage themselves by providing insufficient gasLimit****Severity** LOW SEVERITY**Description**

In `bridgeAndCallERC20` and `bridgeAndCallNative`, the user can specify the `requiredGasLimit` and `recipientGasLimit`.

There is nothing to prevent the user from defining a very small gas limit. In such cases, the relayer will attempt to execute the user transaction, the execution will fail, the relayer will get their primary/secondary fee and the message will be added to a list of failed transactions. From that point on, if the user wants their transaction to be re-executed, they must call `TeleporterMessenger.retryMessageExecution()` and pay for all the gas, effectively donating the primary/secondary fee to the relayer.

Recommendation

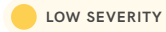
Consider reverting if the `gasLimit` is less than a minimal amount in `bridgeAndCallERC20`, `bridgeAndCallNative` or `registerDestinationTokenBridge`.

Resolution ACKNOWLEDGED

The `requiredGasLimit` input has been removed from all functions. It is set by the Router owner during `registerDestinationTokenBridge`. It is up to the owner to ensure that it is sufficient, based on the destination chain specs. `recipientGasLimit` is highly dependent on the function called on the destination chain, and this cannot be predicted. The team decided that it is out of scope of the Router to check this value. Link to PR: <https://github.com/suzaku-network/suzaku-contracts-library/pull/36>

While this issue was acknowledged, we must bring attention to the issue that removing `requiredGasLimit` was not a good idea. It cannot be assumed that a native call and an ERC20 call will cost the same in terms of gas. That means that the gas cost for one of the calls will be overestimated. We strongly recommend for the `requiredGasLimit` argument to be added back for bridge and call functions and just set a minimum gas limit amount instead to ensure that the users will not use low gas amounts on these functions.

Severity



Description

According to the [teleporter docs](#), even if users provide enough gas for the execution of their call, only 63/64 of the gas will be used for the execution of the call due to EIP-150. Here is a breakdown of the gas in TeleporterMessenger._handleInitialMessageExecution() to see how much gas has to be specified in requiredGasLimit:

```
function _handleInitialMessageExecution(
    bytes32 messageID,
    bytes32 sourceBlockchainID,
    TeleporterMessage memory message
) private {

    require(gasleft() >= message.requiredGasLimit,
        "TeleporterMessenger: insufficient gas");

    if (message.destinationAddress.code.length == 0) {
        _storeFailedMessageExecution(messageID,
            sourceBlockchainID, message);
        return;
    }

    bytes memory payload = abi.encodeCall(
        ITeleporterReceiver.receiveTeleporterMessage,
        (sourceBlockchainID,
            message.originSenderAddress, message.message)
    );

    bool success =
        _tryExecuteMessage(message.destinationAddress,
            message.requiredGasLimit, payload);

    if (!success) {
        _storeFailedMessageExecution(messageID,
            sourceBlockchainID, message);
        return;
    }

    emit MessageExecuted(messageID, sourceBlockchainID);
}
```

After the `gasleft() >= message.requiredGasLimit` check, there is 100 000 gas. Then this amount is reduced due to the gas needed to execute `message.destinationAddress.code.length == 0` and then generating the payload—payload gas consumption is arbitrary and it is based on the length of the provided data.

Then we reach the actual call execution and the provided gas limit of this call is 100 000 but in reality the gas we have up to this point could be 85 000 (for the example) so the call will be able to use $85\,000 * 63 / 64 = 83\,671$.

Recommendation To quote the teleporter documentation:

“Based on the current implementation, a message must have a `requiredGasLimit` of over 1,200,000 gas for this to be possible. In order to avoid this case entirely, it is recommended for applications sending Teleporter messages to add a buffer to the `requiredGasLimit` such that 63/64ths of the value passed is sufficient for the message execution.”

Consider increasing the provided gas limit by the user

- with 1/64 of the amount so the forwarded gas for the call execution equals the desired gas limit
- by increasing the gas limit based on the size of the payload since before the execution of the call the payload and other parameters get encoded as a call (`bytes memory payload = abi.encodeCall()`).

Resolution

ACKNOWLEDGED

`requiredGasLimit` is set by the Router owner during `registerDestinationTokenBridge`. It is up to the owner to ensure that it is sufficient, based on the destination chain specs.

Link to PR: <https://github.com/suzaku-network/suzaku-contracts-library/pull/36>

Issue #09**All native bridge functions give more allowances to the bridge source than needed****Severity** LOW SEVERITY**Description**

Bridge native functions of the router increase the allowance for the feeToken (wrapped native token) with `msg.value` but the actual amount that needs to be approved is the `primaryFeeAmount` because this is the only amount that the ICTT transferrer will try to transfer from the router.

The main amount is sent to the transferrer as a native currency.

Example `bridgeNative` function:

```
SafeERC20.safeIncreaseAllowance(IERC20(feeToken),  
bridgeSource, msg.value)
```

Recommendation

Record the difference in the router balance before and after the wrapping of the native currency `primaryFeeAmount` and increase the transferrer's allowance only with that amount.



If the deposit is not 1:1, then the transaction might fail or outstanding allowances could still be possible.

Consider changing the following lines:

```
SafeERC20.safeIncreaseAllowance(IERC20(feeToken),  
bridgeSource, primaryFeeAmount);  
WrappedNativeToken(payable(feeToken)).deposit{value:  
primaryFeeAmount}();
```

to:

```
WrappedNativeToken wrappedFeeToken =  
WrappedNativeToken(payable(feeToken));  
uint256 wrappedNativeBalance =  
wrappedFeeToken.balanceOf(address(this));  
wrappedFeeToken.deposit{value: primaryFeeAmount}();  
wrappedNativeBalance =  
wrappedFeeToken.balanceOf(address(this)) -  
wrappedNativeBalance;  
  
if (wrappedNativeBalance > 0) {  
    SafeERC20.safeIncreaseAllowance(IERC20(feeToken),  
bridgeSource, wrappedNativeBalance);  
}
```

Description

If the owner uses `registerSourceTokenBridge` to change the source bridge address of a token that already has a registered bridge, then the token address will get duplicated in `tokenList`. Consider using `EnumerableSet` from OpenZeppelin to have an easier token accounting.

Consider using a constant variable for the maximum amount of basis points instead of the hardcoded value of `10_000` throughout the codebase.

Consider validating if `primaryRelayerFeeBips` or `secondaryRelayerFeeBips` arguments are not more than `10_000` in all bridge functions and in `updateRelayerFeesBips`. The sum of these 2 fees should not be more than 100%.

`_removeToken` and `_removeDestinationChainID` loop until they find the item they want to remove. This pattern should be avoided because of increasing gas costs when an item is at the end of a very large array. In extreme cases, the external functions that use these functions can be DOSed if the array becomes so large that the gas becomes bigger than the block gas limit. Consider computing the to-be-removed item index off-chain and providing just the index in order to remove the item from the array on chain. `EnumerableSet` from OpenZeppelin should solve this issue.

Using the tertiary operator will provide better handling of the secondary fee, e.g.

```
uint256 secondaryFeeAmount = destinationBridge.isMultihop
    ? (msg.value * secondaryRelayerFeeBips) / 10_000
    : 0;
```

Consider adding the fees as well to the following events:

- BridgeERC20
- BridgeAndCallERC20
- BridgeNative
- BridgeAndCallNative

—

In case of non-multihop operations, the secondary fee is set to 0. By following the same logic, consider adding setting `multiHopFallback` to `address(0)` as well since the transaction will revert if it is not `address(0)`.

—

BridgeERC20, BridgeAndCallERC20, BridgeNative and BridgeAndCallNative events should contain the fee token address as well.

Recommendation Consider resolving the typographical issues.

Resolution



A new issue was added during the resolution round.

2.2 **AvalancheICTTRouterFixedFees**

AvalancheICTTRouterFixedFees is a variation of the AvalancheICTTRouter but the fees are set by the owner of the contract.

The fees applied to the transfers are deducted as follow:


- Primary fee is taken on the source chain, the chain that initiates the transaction.
- Secondary fee is taken only if the transaction is a multi-hop. A multi-hop is a transaction that goes through multiple chains to destination. This fee is taken on the intermediary chain's bridge amount, which is the initial amount without the primary fee.

2.2.1 **Privileged Functions**

- `updateRelayerFeesBips`
- `registerSourceTokenBridge`
- `registerDestinationTokenBridge`
- `removeSourceTokenBridge`
- `removeDestinationTokenBridge`
- `transferOwnership`
- `renounceOwnership`

2.2.2 Issues & Recommendations

NOTE: All the issues described in the `AvalancheICTTRouter` apply to this contract as well. The following issue is an additional issue for this contract only.

Issue #11	Primary and secondary fees set by the owner can cause the user to overpay or underpay the fees
Severity	 LOW SEVERITY
Description	<p>The primary and secondary fees are calculated based on a percentage set by the router owner, applied to the amount of ERC20 or native tokens provided.</p> <p>Relayers of the teleporter protocol are incentivized by these fees to execute transfers and calls. The primary fee supports transfers and calls to the first destination chain (either the home chain or a remote chain), while the secondary fee is deducted from the transferred amount on the home chain when routing to a final remote chain in multi-hop transactions.</p> <p>A key issue is that the primary and secondary fee percentages are owner-configured, which can lead to situations where the fees are either too high or insufficient for the required transaction costs.</p> <p>For example, consider a <code>bridgeAndCallERC20</code> transaction: if the payload to be executed on the destination chain (remote chain -> home chain -> final destination chain) is large, it may require a higher gas allocation. If the secondary fee does not cover this, the relayer may lack sufficient incentive to execute the message. In such cases, users may need to add funds via <code>TeleporterMessenger.addFeeAmount()</code> to increase the fee to ensure execution.</p> <p>If the user tries to provide enough primary and secondary fees in this case, they could be forced to send way more ERC20 or native tokens in order for his call to be executed.</p> <p>This percentage-based fee structure can be problematic; for instance, a 5% fee on a 1000 WAVAX swap versus a 1000 USDC swap results in very different absolute values, potentially affecting the relayer's incentive or the transaction's feasibility.</p>

Recommendation Calculating approximate primary and secondary fees is complex and may require additional logic. A potential solution is to allow users more flexibility in setting these fees:

- **User-Defined Percentage Fees:** Allow users to decide whether fees should be a percentage of the transaction amount. In this case, the owner could set default percentage fees while also enforcing a minimum fee to increase the likelihood of transaction success.
- **Fixed Fee Option:** Let users specify fees as fixed amounts instead of percentages. For this, the owner could set a minimum fee per token, ensuring sufficient coverage if the percentage-based option is not chosen. This minimum fee could be established for known tokens.

Example (not meant for use without testing):

```
if (feeType == FeeType.Percentage) {  
    // Calculate percentage-based fees  
    primaryFeeAmount = (amount *  
primaryRelayerFeeBips[token]) / 10_000;  
    secondaryFeeAmount = (amount *  
secondaryRelayerFeeBips[token]) / 10_000;  
  
    // Enforce minimum fees  
    if (primaryFeeAmount < minimumPrimaryFeeAmount[token]) {  
        primaryFeeAmount = minimumPrimaryFeeAmount[token];  
    }  
    if (secondaryFeeAmount <  
minimumSecondaryFeeAmount[token] &&  
destinationBridge.isMultihop) {  
        secondaryFeeAmount =  
minimumSecondaryFeeAmount[token];  
    }  
} else if (feeType == FeeType.Fixed) {  
    // Use fixed fee amounts  
    primaryFeeAmount = primaryFixedFeeAmount[token];  
    secondaryFeeAmount = destinationBridge.isMultihop ?  
secondaryFixedFeeAmount[token] : 0;  
} else {  
    revert("Invalid fee type");  
}
```

Resolution

The team commented: "We decided to create a new struct in the `IAvalancheICTTRouterFixedFees` interface, `MinBridgeFees`, to manage the minimum amount the bridge can support. We added a new mapping to the `AvalancheICTTRouterFixedFees` contract to map the minimal primary and secondary relayer fees to a token on the destination chain. We reimplemented the `registerDestinationTokenBridge` function to add the minimal fees input and update the mapping.

We created a new error to revert the bridge functions if the amount bridged is too low when comparing the fees collected from this amount and the minimal fees set up by the owner for the token and the destination chain."

Issue #12**Add a custom getter for
`destinationChainTokenToMinBridgeFees`****Severity****Description**

Some block explorers are not able to easily handle mixed mappings and structs—it is better to add an explicit get function for this mapping for third parties to integrate it easily.

Recommendation

Consider adding a custom getter function that takes as parameters a `bytes32` and an address and returns a `MinBridgeFees` object.

Resolution

2.3 Interfaces (IAvalancheICTTRouter, IAvalancheICTTRouterFixedFees)

IAvalancheICTTRouter and IAvalancheICTTRouterFixedFees are the interfaces of AvalancheICTTRouter and AvalancheCTTRouterFixedFees.

2.3.1 Issues & Recommendations

No issues found.





PALADIN
BLOCKCHAIN SECURITY