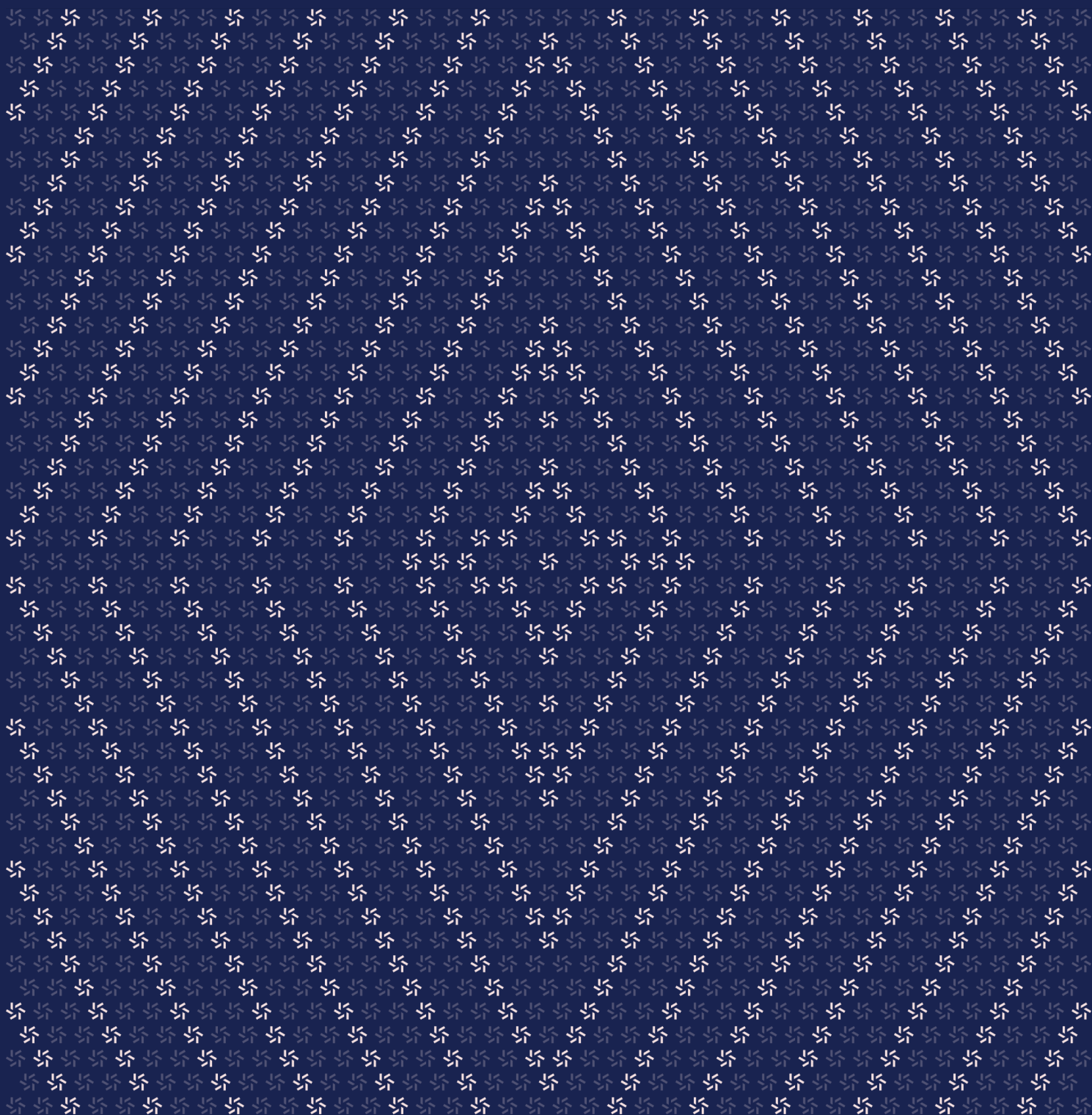


January 26, 2024

Aqua Pool

Smart Contract Security Assessment



Contents

About Zellic	4
<hr/>	
1. Executive Summary	4
1.1. Goals of the Assessment	5
1.2. Non-goals and Limitations	5
1.3. Results	6
<hr/>	
2. Introduction	6
2.1. About Aqua Pool	7
2.2. Methodology	7
2.3. Scope	9
2.4. Project Overview	9
2.5. Project Timeline	10
<hr/>	
3. Detailed Findings	10
3.1. Invariant may sometimes be calculated incorrectly	11
3.2. Incorrect operator in <code>_tweakPrice</code>	13
<hr/>	
4. Discussion	14
4.1. Additions over the CurveTwoCryptoOptimized Vyper code	15
4.2. Missing checks compared with the CurveTwoCryptoOptimized Vyper code	18
4.3. Differential fuzz testing	19
4.4. A <code>withdrawMode == 3</code> skips minting protocol fee	19

5.	Assessment Results	20
5.1.	Disclaimer	21

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the [#1 CTF \(competitive hacking\) team](#) worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io and follow [@zellic_io](#) on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io.



1. Executive Summary

Zellic conducted a security assessment for SyncSwap Labs from January 22nd to January 26th, 2024. During this engagement, Zellic reviewed Aqua Pool's code for security vulnerabilities and functional issues introduced when porting the mathematics of Curve twoCrypto V2's pool.

1.1. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is the math and logic consistent with Curve's twoCrypto V2 pool, aside from the modifications documented by SyncSwap Labs?
- Are the modifications introduced by SyncSwap Labs safe?

It is important to note that we have generated a base framework for fuzzing the pool's components, for ease of comparing the Vyper and Solidity implementations and for testing the pool's math. However, due to time constraints, we could not fully implement this framework. As of the time of writing this report, the framework only incorporates the mint (i.e., `add_liquidity`) function. Together with this report, we will deliver the framework to SyncSwap Labs so that they can continue to extend it to test the pool's math and ensure that no inconsistencies are introduced in the future.

1.2. Non-goals and Limitations

We did not assess the following areas or questions that were outside the scope of this engagement:

- Does the math in the Curve twoCrypto V2 pool (e.g., for swaps and rebalancing) — which the in-scope contract is ported from — work as intended in its design specification, and is it free of functional or security bugs?
- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

During this assessment, lack of time and the complexity of the math prevented us from fully understanding relevant Curve twoCrypto V2 math (particularly that relating to rebalancing) and assessing smart contract risks. We were asked to focus on ensuring the pool's math was ported without error, so we prioritized this goal.






1.3. Results

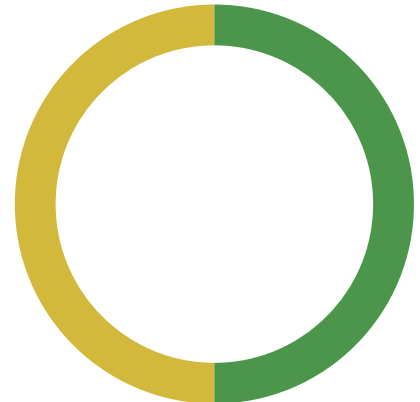
During our assessment on the scoped Aqua Pool contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and one was of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for SyncSwap Labs's benefit in the Discussion section ([4.7](#)) at the end of the document.

We strongly advise SyncSwap Labs to invest some resources into implementing differential fuzz testing, which showed the potential to detect additional inconsistencies between SyncSwap and Curve in our preliminary exploration.

Breakdown of Finding Impacts

Impact Level	Count
 Critical	0
 High	0
 Medium	1
 Low	1
 Informational	0



2. Introduction

2.1. About Aqua Pool

SyncSwap is a seamless and efficient decentralized exchange on Ethereum ZK rollups. Powered by zero-knowledge technology, SyncSwap brings more people easy-to-use and low-cost DeFi with complete Ethereum security. It aims to provide a seamless and efficient decentralized exchange on the zkSync Era network.

Aqua Pool is a pool with automated concentrated liquidity to support exchange between two tokens on Ethereum ZK rollup networks (such as zkSync Era, Linea, and Scroll).

It is a fork of Curve Finance's V2 twoCrypto pool; the main logics are expected to be the same, and the math library reuses the original Vyper library from Curve. However, the interfaces (such as member and event naming, fee management, and precision) are changed to adopt the SyncSwap interface. The whole repo is built for SyncSwap V2, and the most significant change from V1 is the introduction of the Crypto Pool (or called "Aqua Pool") and the removal of the Vault contract. (This means pool funds are now stored in the individual pool contract instead of the singleton vault contract.)

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4.7) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Aqua Pool Contracts

Repository	https://github.com/syncswap/core-contracts-aqua ↗
Version	core-contracts-aqua: f64e79025ba7be08c49487bcae410ff4a6baf84c
Program	SyncSwapCryptoPool
Types	Vyper, Solidity
Platform	EVM-compatible

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of eight and a half person-days. The assessment was conducted over the course of four calendar days.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
✈ Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Filippo Cremonese
✈ Engineer
fcremo@zellic.io ↗

Vlad Toie
✈ Engineer
vlad@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 22, 2024 Start of primary review period

January 26, 2024 End of primary review period

3. Detailed Findings

3.1. Invariant may sometimes be calculated incorrectly

Target	SyncSwapCryptoPool		
Category	Coding Mistakes	Severity	Medium
Likelihood	Medium	Impact	Medium

Description

The `_tweakPrice` function adjusts the `priceOracle`, `lastPrices` and `priceScale` variables, used in the calculation of the prices within the pool. This function is an essential component of the pool, and is called whenever there is an imbalance during any of the liquidity operations, such as swap, mint or burn.

In `_tweakPrice`, the Vyper code calculates `D_unadjusted`:

```
# ----- If new_D is set to 0, calculate it -----

D_unadjusted: uint256 = new_D
if new_D == 0: # <----- _exchange sets new_D to 0.
    D_unadjusted = MATH.newton_D(A_gamma[0], A_gamma[1], _xp, K0_prev)
```

The corresponding Solidity code-names this value `newInvariant`:

```
// ----- If `newInvariant` is 0, calculate it -----

if (newInvariant == 0) {
    newInvariant = MATH.computeD(a, gamma, xp0, xp1, _k0_prev);
}
```

The invariant that this pool uses is essentially somewhere between the constant-product invariant(like in Uniswap, Balancer, etc.) and the StableSwap(which is essentially Curve v1).

Further down the function, Vyper calculates a value and stores to a new variable `D` this time (not `D_unadjusted`):

```
# ----- Update D with new xp.
D: uint256 = MATH.newton_D(A_gamma[0], A_gamma[1], xp, 0)
```

However, Solidity overwrites `newInvariant` (which was `D_unadjusted`) with this value instead of defining a new variable too:

```
// ----- Update D with new xp.  
  
// Calculate "extended constant product" invariant xCP and virtual price.  
newInvariant = MATH.computedD(a, gamma, _xp0, _xp1, 0);
```

Finally, at the end of the function, Vyper stores `D_unadjusted` to `self.D`. Solidity stores `newInvariant` (which is not `D_unadjusted` but `D` in Vyper) to `invariantLast` (i.e., the equivalent of `self.D` in Vyper).

Impact

For any cases in the Vyper code where `D != D_unadjusted` at the end of the function, the Solidity code will store the wrong value. Per SyncSwap Labs,

This can cause wrong value to be updated to `invariantLast/D` if the pool didn't got enough profit to rebalance.

Recommendations

Use an ephemeral variable when determining whether or not to rebalance profits.

Remediation

This issue has been acknowledged by SyncSwap Labs, and a fix was implemented in commit [ff28d01a](#).

3.2. Incorrect operator in `_tweakPrice`

Target	SyncSwapCryptoPool		
Category	Coding Mistakes	Severity	Low
Likelihood	Low	Impact	Low

Description

The `_tweakPrice` function adjusts the `priceOracle`, `lastPrices` and `priceScale` variables, used in the calculation of the prices within the pool. This function is an essential component of the pool, and is called whenever there is an imbalance during any of the liquidity operations, such as swap, mint or burn.

The Vyper code (in `tweak_price` in the Curve twoCrypto V2 pool) has the following code:

```
# If A and gamma are not undergoing ramps (t < block.timestamp),
# ensure new virtual_price is not less than old virtual_price,
# else the pool suffers a loss.
if self.future_A_gamma_time < block.timestamp:
    assert virtual_price > old_virtual_price, "Loss"
```

The inverse of `>` in the assertion is `<=`. However, the Solidity code (in `_tweakPrice` in SyncSwapCryptoPool) ports the above code as follows, using the `<` operator:

```
if (_futureTime < block.timestamp) {
    if (it.virtualPrice < it.oldVirtualPrice) {
        revert Loss();
    }
}
```

Impact

The above code may not revert in all cases intended by the original Vyper code, and it is not logically equivalent.

Recommendations

Use the `<=` operator instead of `<`.

```
if (_futureTime < block.timestamp) {  
    if (it.virtualPrice < it.oldVirtualPrice) {  
        if (it.virtualPrice <= it.oldVirtualPrice) {  
            revert Loss();  
        }  
    }  
}
```

Remediation

This issue has been acknowledged by SyncSwap Labs, and a fix was implemented in commit [ff28d01a](#).

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Additions over the CurveTwoCryptoOptimized Vyper code

As initially stated, our security review focused on the changes made by the SyncSwap Labs team when translating the Vyper version of [Curve's TwocryptoOptimized pool](#) to Solidity. Below, we list the most notable changes to the overall architecture of the original code.

1. Possibility of callbacks

SyncSwap allows callbacks to contracts that implement a predefined interface. For example, in swap, the following operation occurs:

```
function swap(bytes memory _data, address _sender, _callback,
    _callbackData) {
    // ...
    if (_callback != address(0)) {
        // Fills additional values for callback params.
        params.sender = _sender;
        params.callbackData = _callbackData;

        // Calculates fee amount for callback.
        params.feeIn = Math.mulDivUnsafeFirstLast(params.amountIn,
            params.swapFee, 1e5);

        ICallback(_callback).syncSwapBaseSwapCallback(params);
    }
}
```

This addition allows for more flexibility in the pool's usage and is not present in the Curve Vyper code. We note, however, that due to the nature of reentrant capabilities of callbacks, this addition may introduce additional attack vectors in the future and should be carefully considered.

The actual interface is defined in the ICallback interface and is as follows:

```
// SPDX-License-Identifier: AGPL-3.0-or-later

pragma solidity >=0.5.0;

/// @dev The callback interface for SyncSwap base pool operations.
```

```
/// Note additional checks will be required for some callbacks, see below
/// for more information.
/// Visit the documentation https://syncswap.gitbook.io/api-documentation/
/// for more details.
interface ICallback {

    struct BaseMintCallbackParams {
        address sender;
        address to;
        uint reserve0;
        uint reserve1;
        uint balance0;
        uint balance1;
        uint amount0;
        uint amount1;
        uint fee0;
        uint fee1;
        uint newInvariant;
        uint oldInvariant;
        uint totalSupply;
        uint liquidity;
        uint24 swapFee;
        bytes callbackData;
    }

    function syncSwapBaseMintCallback(BaseMintCallbackParams calldata
    params)
    external;

    struct BaseBurnCallbackParams {
        address sender;
        address to;
        uint balance0;
        uint balance1;
        uint liquidity;
        uint totalSupply;
        uint amount0;
        uint amount1;
        uint8 withdrawMode;
        bytes callbackData;
    }

    function syncSwapBaseBurnCallback(BaseBurnCallbackParams calldata
    params)
    external;

    struct BaseBurnSingleCallbackParams {
        address sender;
        address to;
```



```

        address tokenIn;
        address tokenOut;
        uint balance0;
        uint balance1;
        uint liquidity;
        uint totalSupply;
        uint amount0;
        uint amount1;
        uint amountOut;
        uint amountSwapped;
        uint feeIn;
        uint24 swapFee;
        uint8 withdrawMode;
        bytes callbackData;
    }

    /// @dev Note the `tokenOut` parameter can be decided by the caller, and
    the correctness is not guaranteed.
    /// Additional checks MUST be performed in callback to ensure the
    `tokenOut` is one of the pools tokens if the sender
    /// is not a trusted source to avoid potential issues.
    function syncSwapBaseBurnSingleCallback(BaseBurnSingleCallbackParams
    calldata params)
    external;

    struct BaseSwapCallbackParams {
        address sender;
        address to;
        address tokenIn;
        address tokenOut;
        uint reserve0;
        uint reserve1;
        uint balance0;
        uint balance1;
        uint amountIn;
        uint amountOut;
        uint feeIn;
        uint24 swapFee;
        uint8 withdrawMode;
        bytes callbackData;
    }

    /// @dev Note the `tokenIn` parameter can be decided by the caller, and
    the correctness is not guaranteed.
    /// Additional checks MUST be performed in callback to ensure the
    `tokenIn` is one of the pools tokens if the sender
    /// is not a trusted source to avoid potential issues.
    function syncSwapBaseSwapCallback(BaseSwapCallbackParams calldata
    params)

```

```
external;  
}
```

2. Usage of a Router contract

The pool is designed to be used through a Router contract, which is responsible for calling the pool's functions as well as performing additional checks (such as slippage checks).

Additionally, the transfer of tokens is done through the Router contract atomically with the pool's operations. From the user's perspective, the Router contract is the only contract that interacts with the pool, and the user only needs to interact with the Router contract.

3. Interface differences

We note that when compared to the default CurveTwoCryptoOptimized Vyper code, the Solidity version has a drastically different interface. Functions, events, and variable names are different. Some functions also have different parameters.

4. Precision differences

The fee precision has been changed from $1e10$ to $1e5$.

4.2. Missing checks compared with the CurveTwoCryptoOptimized Vyper code

Some invariants are not checked in the Solidity code, but they are checked in the original Vyper code.

For example, the Solidity code does not check the `frac` invariant in `tweakPrice`. The [original Vyper](#) code performs the following check:

```
for k in range(N_COINS):  
    frac: uint256 = xp[k] * 10**18 / D # <----- Check validity of  
    assert (frac > 10**16 - 1) and (frac < 10**20 + 1) # p_new.
```

Additionally, the Solidity code does not check that the `token_amount` is less than the total `token_supply`, as the [original Vyper](#) does.

Similarly, throughout the contract, other checks are missing. We note, however, that we could not identify any direct security implications of these missing checks.

4.3. Differential fuzz testing

Differential fuzz testing is a form of fuzzing where the effects of the execution of two (or more) programs are compared to detect differences. During the course of this engagement, we started developing a harness to perform this kind of testing. Even though we were not able to complete the harness due to time constraints, we shared our work with SyncSwap Labs as a basis on which they could build a comprehensive test. The current version of the harness already detects some differences in behavior in the `mint / add_liquidity` functions, which given the same inputs, do not seem to match in behavior in at least two ways: there are cases where SyncSwap reverts while Curve does not. Additionally, the amount of LP tokens minted to the user is often slightly different, with the SyncSwap codebase having a tendency to mint a lesser amount in absolute terms. We could not identify the root cause of these differences with certainty. While these inconsistencies do not necessarily represent an issue, we strongly advise SyncSwap Labs to identify and document the causes of the differences in behavior.

In order to set up the fuzz harness, we set up a Foundry test, which ultimately instantiates one SyncSwap Crypto pool and one Curve pool. Instantiating the pools requires deployment of other helper infrastructure, including factory contracts, mock ERC-20 tokens, math library contracts, and so on. To make meaningful comparisons, the two pools need to be deployed using the same curve parameters. Additionally, we removed fees by making targeted modifications to the code of the pools, which was otherwise left unmodified.

Deploying the Curve code required identifying how to obtain the corresponding bytecode. This can be done by manually compiling the code and embedding it as a hexadecimal byte string in the source as well as via the use of FFI cheat codes, which allow to invoke the Vyper compiler at test runtime.

The harness is engineered to contain one fuzzable entry point for each feature exposed by the contract — the primary features being minting and burning of LP tokens and swapping. The harness performs the required operations to do the same operation on both contracts; this entails, for example in the case of minting/adding liquidity, minting the required tokens to be added as liquidity, transferring them or granting the needed approval, and invoking the SyncSwap and Curve contracts. After the operation is performed, the execution results and contract states are compared to detect differences.

4.4. A `withdrawMode == 3` skips minting protocol fee

Note that when `withdrawMode` is 3, the burn function skips minting the protocol fee:

```
// Mints protocol fees. In emergency case, skip it with withdraw mode.
if (params.withdrawMode != 3) {
    params.totalSupply = (
```

```
        _mintProtocolFee(params.totalSupply - params.liquidity,  
        _newInvariant, priceScale)  
    );  
}
```

SyncSwap Labs noted that this behavior is intentional.

5. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Ethereum Mainnet.

During our assessment on the scoped Aqua Pool contracts, we discovered two findings. No critical issues were found. One finding was of medium impact and one was of low impact. SyncSwap Labs acknowledged all findings and implemented fixes.

5.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.