Towards Verified Price Oracles for Decentralized Exchange Protocols

Kinnari Dave
CertiK, New York, NY, USA

Vilhelm Sjöberg
CertiK, New York, NY, USA

Xinyuan Sun
CertiK, New York, NY, USA

Abstract

Various smart contracts have been designed and deployed on blockchain platforms to enable cryptocurrency trading, leading to an ever expanding user base of decentralized exchange platforms (DEXs). Automated Market Maker contracts enable token exchange without the need of third party book-keeping. These contracts also serve as price oracles for other contracts, by using a mathematical formula to calculate token exchange rates based on token reserves. However, the price oracle mechanism is vulnerable to attacks both from programming errors and from mistakes in the financial model, and so far their complexity makes it difficult to formally verify them. We present a verified AMM contract and validate its financial model by proving a theorem about a lower bound on the cost of manipulation of the token prices to the attacker. The contract is implemented using the DeepSEA system, which ensures that the theorem applies to the actual EVM bytecode of the contract. This theorem could be used as proof of correctness for other contracts using the AMM, so this is a step towards a verified DeFi landscape.

2012 ACM Subject Classification Software and its engineering → Software verification

Keywords and phrases Smart Contract Verification, Interactive Theorem Proving, Blockchain, Decentralized Finance

Digital Object Identifier 10.4230/OASIcs.FMBC.2021.1

Supplementary Material *Software (Source Code)*: `https://github.com/certikfoundation/deepsea`

Acknowledgements The work was partially supported by a Conflux Ecosystem Grant (October 2020).

1 Introduction

The last two years have seen a rapidly increasing interest in using *decentralized finance* (DeFi) instead of traditional centralized exchanges in order to trade, lend, and borrow cryptocurrencies. DeFi puts the trading logic into a smart contract on the blockchain, which increases trust and transparency, and lets anyone compose financial applications "like lego pieces". Smart contracts also enable completely new financial primitives, e.g. *flash loans* [22], risk-free lending of very large amounts which will be paid back within a single blockchain transaction. Estimates say DeFi total trading volume increased from \$0.67 billion in January 2020 to \$70 billion in January 2021, while DeFi investments reached 20.5 billion in January 2021 [16, 17].

However, protocols in decentralized finance are vulnerable to hacks. In 2020 there were at least 16 large DeFi hacks, with total losses of \$196 million. Some of these were due to mistakes in the financial model (we give an example below in Section 2.2), while in others the financial theory was sound but the contract itself was implemented incorrectly [19].

The high stakes of DeFi makes it crucial that the smart contracts executing these protocols come with formal guarantees. However, applying formal verification to them is challenging. Reasoning about the financial models often requires mathematics, e.g. real analysis, that goes beyond the capabilities of non-interactive theorem provers such as SMT solvers. And even if we can prove theorems about the financial model, we must still show that the actual program code correctly implements the model. Existing tools either try to work at the model level, or they can prove quite shallow properties about code.

In this paper, we consider one of the most widely used DeFi protocols, a Uniswap-style automated market making (AMM) contract. Various attempts [4, 3, 7] have been made at studying the AMM model mathematically and reasoning about specific cases. However, these are paper proofs and do not directly reason about the program being executed on the virtual machine.

We make use of the DeepSEA system, which has been tailored to support rigorous formal verification. The DeepSEA compiler can automatically generate a high-level Coq model for a contract, so we know that the theorem we prove in Coq will apply to the actual executed code. Achieving this requires some care, because existing work deals with the AMM model in terms of real numbers, and we must lift that proof to give bounds for the integer variables in the actual program.

AMM contracts are a basic building block of more complex financial contracts. In the future, we envision that such contracts will also be formally verified, e.g. by using the result we prove here as one lemma.

Contributions. We make the following contributions in this paper:

1. We implement a Uniswap-style AMM contract in DeepSEA (Section 3).
2. We formalize in Coq a result, previously only proven on paper [4], which establishes a lower bound on the cost to an attacker of manipulating prices quoted by the AMM contract (Section 5.2). Our formalization makes use of existing third-party math developments (Section 5.1), which shows the benefit of working inside a general-purpose proof assistant.
3. We also establish the non-depletion property of the contract, i.e it is impossible to drain the contract of all it's reserves by swapping any number of tokens. (Section 5.2). This proof requires exporting integer inequalities to reals and using ring homomorphism properties to transport them back to integers, as is evident in the *math_lemma.v* [1] module.
4. We use the auto generated Coq functions by the DeepSEA compiler frontend to link the bytecode to the formalized proof, thus establishing important financial properties of the generated bytecode (Section 5.3).

In the rest of the paper, we first explain the setting of the work (Section 2), then our specific contributions, and finally we discuss related work and conclude.

2 Background

"Market making" is the process of providing liquidity for various assets by continuously quoting of the price at which the market maker is willing to buy and the price at which the market maker is willing to sell their asset. Traditionally, these price quotes are listed in an order book, which records the current assets open for buying/selling. This requires trust in the central party managing the liquidity pools. The idea behind an Automated Market Maker protocol is to replace the central party with a smart contract that owns reserves of

[1] Available online at `https://github.com/certikfoundation/deepsea/blob/master/contracts/amm/math_lemma.v`

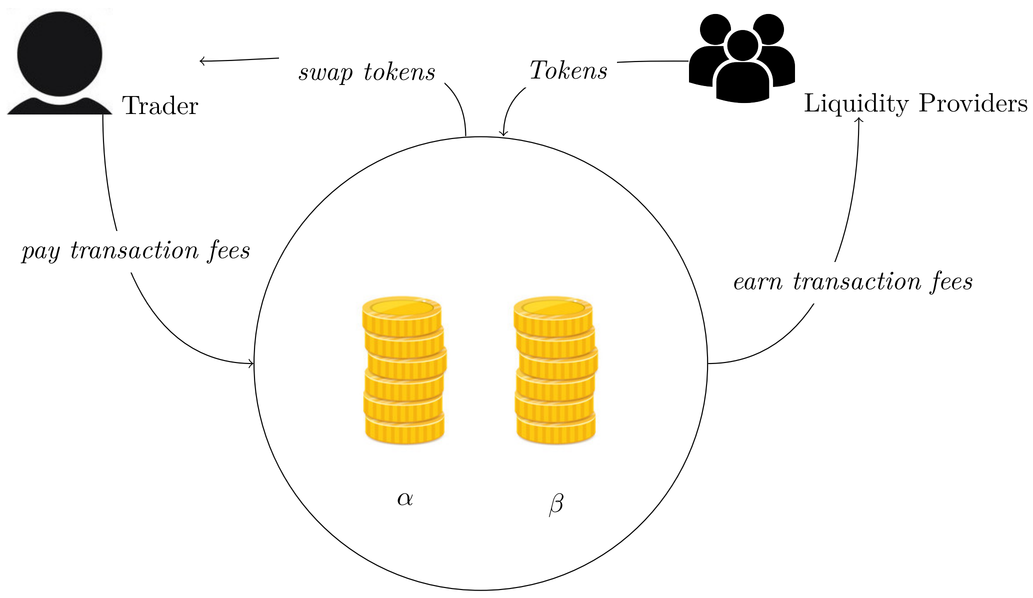

Figure 1 AMM mechanism.

two Ethereum-hosted cryptocurrencies (a.k.a tokens), and trades them at a price determined using a mathematical formula. Once these smart contracts are deployed, they can also serve as price oracles for other smart contracts.

2.1 Automated Market Makers

Automated Market Makers are decentralized exchange protocols which facilitate trading of tokens on blockchain based platforms by providing liquidity pools without an order book mechanism. These protocols allow exchange between pairs of tokens, and each token pair has a corresponding smart contract which facilitates the exchange. The exchange rate is calculated using a mathematical formula which is a function of the token reserves in the contract. We focus on the constant product market makers. This type of automated market makers satisfy the invariant:

$$x_A . y_B = k$$

where x_A, y_B are the reserves for token A and token B respectively. It follows from this formula that the marginal price of A with respect to B (the price offered by the contract for small trades) is the ratio of the token reserve of B to that of A. Since the Uniswap protocol [1] is one of the most popular implementations of the AMM mechanism, we briefly describe the functionalities it provides and its mechanism. The protocol is designed so that the smart contract implementing it interacts with two kinds of users: Liquidity Providers and Traders.

Liquidity Providers contribute to the pool of reserves of the token pair. This is enabled by issuing a liquidity token to the Liquidity Provider when they choose to contribute to the pool. The token dictates the share of the token reserves that the provider is entitled to. The provider can treat the liquidity token as an asset that can be traded. To incentivize the providers, they receive an interest proportionate to their shares, which is funded by charging traders a 0.3% transaction fee for each trade they make with the contract. The mint() method facilitates minting of liquidity tokens for the providers.

At any point, the provider is free to withdraw liquidity from the token reserves by calling the burn() method from the smart contract. On doing so, they receive the tokens they had lent to the liquidity pool plus the interest they earned from the transaction fees.

The number of liquidity tokens minted for a particular liquidity provider is determined by their share in the token reserve. There are various formulae used to calculate this. The Uniswap protocol determines the number of tokens minted using the following formula:

$$s_{minted} = \frac{x_{deposited}}{x_{starting}}.s_{starting}$$

Here, s_{minted} denotes the number of Liquidity Tokens minted for the amount $x_{deposited}$ tokens that have been deposited to the pool containing $x_{starting}$ tokens and $s_{starting}$ liquidity tokens representing contributions to that liquidity pool.

In the event that liquidity is being deposited to the reserves for the first time, the above formula doesn't work (since $x_{starting} = 0$). In this case, the number of liquidity tokens minted is given by:

$$s_{minted} = \sqrt{x_{deposited} \cdot y_{deposited}}$$

where $x_{deposited}, y_{deposited}$ denotes the pair reserves that the depositor lends to the contract. Additionally, the Uniswap protocol charges a 0.05% protocol fee as a part of the net 0.3% transaction fee charged to traders. This fee is optional and is turned off for the DeepSEA implementation of the AMM contract.

2.2 Oracles

The AMM mechanism also supports a price oracle function. The first protocol designed by Uniswap supports an on-chain price oracle which computes prices using the constant product market maker formula and reports instantaneous prices when queried. Other contracts can e.g. issue loans of one token guaranteed by a collateral in another token, and use the reported price to calculate how much the collateral is worth. However, the mechanism of reporting instantaneous prices is highly susceptible to attacks, in particular in combination with flash loans. Let us consider an example of an oracle attack which happened on 18th February 2020 [5]:

▶ **Example 1.** The bZx protocol is a lending protocol which facilitates decentralized borrowing and lending of assets on Ethereum. The Kyber network on the other hand is an on-chain AMM protocol similar to Uniswap. An attacker flash-borrowed 7500 ETH from the bZx protocol, then called the Kyber protocol to swap a net amount of 900 ETH with 155,994 sUSD. This affects the reserves of ETH and sUSD in the Kyber protocol thus affecting the prices reported by it. The attacker later relies on bZx quering the faulty Kyber oracle to borrow ETH against sUSD at a cheap rate. To get the sUSD required to perform this exchange on bZx, the attacker buys sUSD from an unrelated contract at a normal rate. They used the Synthetix depot contract, which had larger reserves and therefore did not change price as much as Kyber. They used 3518 ETH from their borrowed ETH to get 943,837 sUSD. Now, they borrow 6796 ETH from bZx with a collateral of only 1,099,841 sUSD. They are able to do this because of the price manipulation on the Kyber oracle which bZx queries. Finally, they are able to transfer back the 7500 ETH borrowed from bZx to repay the flash loan. In effect, bZx lost $600k in equity.

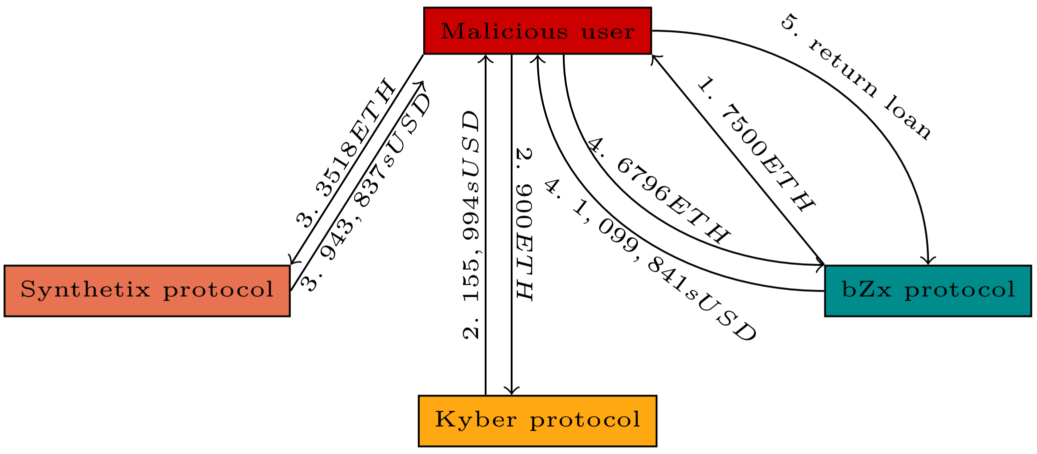

Figure 2 bZx protocol attack.

How can such attacks be avoided? There are several partial solutions. The lending contract can try to avoid being called inside a flash-loan transaction (limiting the amount of funds available for oracle manipulations), or use a "slippage check"[2] to detect if a manipulation is in progress. The oracle can report a time-weighted average instead of the instantaneous price, to smooth out spikes (this approach is adopted by the Uniswap v2 protocol). We believe the ideal solution, which we build towards in this paper, is to *prove* that attacks are impossible by calculating the cost of such manipulation to the attacker as a function of various parameters of the contract such as token reserves. Once this is achieved, these parameters can be modified to make the cost of manipulation high enough that the attack can not be carried out using the funds available to the attacker.

2.3 The DeepSEA system

DeepSEA (Deep Simulation of Executable Abstractions), is a programming language and system that links high-level specifications in Coq [20] to executable code.The original version [18] compiled programs into C, while a new version [10] compiles Ethereum contracts to Ethereum Virtual Machine (EVM) bytecode.

The DeepSEA compiler works in two steps. The front end parses and type-checks the input to create a typed intermediate representation. From the intermediate representation it then generates two things. First, a set of Coq Gallina functions that serves as a high-level model of the program, one function for each method in the contract. Since Gallina is a pure functional language, monads are used to capture effects. The end-user can load this model into their own Coq project, and prove theorems about the contract just as they would about any program written in Coq. Second, there is a backend similar to the CompCert compiler [12], which goes through a series of phases of intermediate representations and generates an EVM bytecode file. Crucially, there is a proof in Coq (although it is not yet complete) that this compilation is done correctly, which will give a high degree of certainty that results proven about the high-level specifications also hold for the bytecode.

[2] Slippage is defined to be the change in price in the time period between a trade order being placed and it's execution. Hence causing traders to settle for a price different from the one they initially requested. An ongoing attack would cause such slippage.

Contracts written in DeepSEA are structured similarly to Solidity contracts, as a set of objects which contain state (storage) variables and methods which can modify the state. In DeepSEA the objects are further organized into "layers", which can express the modular structure of large systems.

3 DeepSEA AMM

The smart contract written in DeepSEA to support AMMs[3] uses the Uniswap v2 protocol as a blueprint. Instead of dividing the functionality of the protocol into two basic types of smart contracts (as is done in the Uniswap protocol), the DeepSEA contract combines the functionality of the router contracts and that of the core contracts into a single contract with two sets of methods corresponding to the above classification.

In the DeepSEA setup, the entire contract is defined as a layer AMM on top of an underlay layer called the AMMLIB. The AMMLIB layer consists of three objects: two ERC20 tokens which are to be swapped and a liquidity token. The AMM layer acts as the interface for the contract. This layer consists of an object of type AMMInterface, which defines the methods that provide all the functionalities of the protocol. The methods in this object signature are given as follows:

- simpleSwap0: This method allows the transfer of one token to the contract to be exchanged for the other, and returns the amount of the second token to be received in return.
- mint: This method allows the transfer of liquidity to a liquidity pool for a liquidity provider.
- burn: This method allows a liquidity provider to withdraw liquidity from a pool.
- sync: This method is a recovery mechanism method to prevent the market for the given pair from being stuck in case of low reserves.
- skim: This method prevents any user from depositing more tokens in any reserve than the maximum limit, to prevent overflow.
- k: This method tracks the product of the reserves.
- quote0: This method returns the equivalent amount of the second token, given an amount of the first token and current reserves in the contract.
- getAmountOut0: This method returns the maximum possible amount of a token than can be gained in exchange for a particular input amount of the other token and that of the reserves.
- getAmountIn0: This method returns the amount of a given token that must be input in order to obtain the desired amount of the other token under the given reserves.

Compared to the Uniswap protocol, we have made a few simplifications. Unlike Uniswap, which offers the option of switching on/off the protocol fee, the DeepSEA contract does not model protocol fees. Moroever, instead of using the above mentioned square root formula to calculate the share of minted liquidity tokens for a liquidity provider, the DeepSEA contract uses the product and burns the first 1000 coins, as in Uniswap v2. The price oracle mechanism is based on Uniswap v1, and the DeepSEA contract does not support flash swaps (i.e., getting flash loans of the assets in the liquidity pool). In the future we may add these features, in order to make our contract completely ABI-compatible with the original. However, the DeepSEA AMM contract already offers all the core functionality offered by the Uniswap protocol, and it contains everything that is relevant to the specification that we are verifying. As such, our proof is an example of verifying a realistic contract.

[3] Available online at `https://github.com/certikfoundation/deepsea/blob/master/contracts/amm/amm.ds`

4 Mathematical Analysis of Automated Market Makers

While AMM contracts hold a great deal of promise for the future of Decentralized Finance, the stability of the markets generated using smart contracts remains a concern. To address such concerns and provide a rigorous comparison with Centralized Finance, Angeris et al. carried out a mathematical analysis [4]. They define the conditions on the Uniswap price in terms of the market price so that no arbitrage opportunities arise. Moreover, they show that it is impossible to drain the contract of all it's liquidity reserves, and go on to model risk in the constant product market maker model. Here we give a brief description of their results, which we mechanize uisng the Coq proof assistant and connect to the DeepSEA AMM contract.

4.1 Manipulating Prices

Since the AMM contract calculates the prices of tokens based on liquidity reserves using a mathematical formula, an attacker can potentially trade with the contract to alter reserves in order to manipulate the prices, as illustrated in Section 2.2. Angeris et al. prove that the cost of such a manipulation is proportionate to the reserves, thus confirming the intuition that large liquidity reserves lead to stable prices.

Consider an AMM contract which facilitates trading of two tokens α, β. Let the token reserves in this contract before the swap to be analysed is performed be given by R_α, R_β respectively. Further assume that the swap involved a deposit of Δ_β β coins and a corresponding withdrawal of Δ_α α coins. We consider the cost of manipulating the market price in the event of the reference market price being infinitely liquid (i.e when $\Delta_\beta = m_p \Delta_\alpha$).

Suppose that the attacker performs this swap to manipulate the reported price by a fraction ϵ. This gives us the intended price $m_u$as a function of ϵ and m_p:

$$m_u = (1+\epsilon)m_p$$

Since the AMM under consideration uses the constant product market maker formula, an alternative computation of the new price is obtained to be the inverse ratio of the token reserves:

$$m_u = \frac{R_\beta + \Delta_\beta}{R_\alpha - \Delta_\alpha}$$

Thus, giving us the equation $\frac{R_\beta+\Delta_\beta}{R_\alpha-\Delta_\alpha} = (1+\epsilon)m_p$.

This swap is made in order to achieve the new price as the reported price by the oracle when queried. Since it is performed by the attacker, Angeris et al. calculate it's cost to the attacker and establish a lower bound on this cost. The attacker deposited Δ_β amount of β tokens in the contract and received Δ_α amount of α tokens from the contract. The price of the α tokens relative to the β tokens before the manipulation is given by m_p. Thus, the net cost of this manipulation to the attacker is simply given by the expression:

$$\Delta_\beta - m_p.\Delta_\alpha$$

After some algebraic simplifications, this cost can be calculated as a function of ϵ and fixed contract parameters:

$$C(\epsilon) = \Delta_\beta - m_p\Delta_\alpha = R_\beta(\sqrt{1+\epsilon} + (\sqrt{1+\epsilon})^{-1} - 2)$$

The cost of manipulation theorem establishes that there is a minimal positive cost to the attacker which is propertional to the reserves of the token added.

▶ **Theorem 2** (Cost of manipulation). *The cost of manipulating the exchange rate of tokens by a fraction ϵ ($C(\epsilon)$) by performing a token swap is bounded below by a function of ϵ depending on it's range:*

- $0 \leq \epsilon \leq 1$: $C(\epsilon) \geq R_\beta \frac{\epsilon^2}{2}\ inf_{0 \leq \epsilon' \leq 1} C''(\epsilon') = (\frac{1}{32\sqrt{2}}) R_\beta \epsilon^2$
- $\epsilon \geq 1$: $C(\epsilon) \geq \kappa R_\beta \sqrt{\epsilon}$

where $\kappa = 3/2 - \sqrt{2}$.

4.2 Nondecreasing k and no depletion

The cost of manipulation result is the most interesting result, because it has implications for the correctness of clients of the oracle and because it uses more advanced math. In addition, Angeris et al. also prove a simpler invariant which gives some additional confidence that the contract is implemented correct. We formalize this proof also.

In particular, they prove that it is impossible to drain the contract of all it's token reserves, irrespective of how many tokens the attacker can use for the swap. This is proved by showing that the net reserves of tokens in the contract are strictly bounded away from 0. The result is stated as follows:

▶ **Theorem 3** (Nondepletion). *At all times,* $R_\alpha + R_\beta > 0$.

The proof for this property follows by the AM-GM inequality, and the *increasing k* invariant. The increasing k invariant establishes that when the protocol charges trading fees, the product of the reserves of the tokens in the contract is strictly increasing over each swap operation.

These properties are related to the notion of *path independence*([9]). For an arbitrary type of AMM, one could worry that it would be possible to exploit the liquidity providers by making repeated trades, e.g. selling token A when the price is low and then buying back when the price is high, and maybe eventually draining the entire reserve. The above invariant shows that constant-product market makers are immune to such attacks. If the product k was exactly constant, then the prices would only depend on the net amount of A traded, not on the exact "path" of buys and sells. In practice k is increasing because of trading fees, which means it is never advantageous to strategically split trades into multiple transactions.

5 Mechanizing results for the DeepSEA AMM contract

In this section we describe our mechanization of the properties stated in Section 4.

5.1 Importing third-party Coq libraries

In order to reason about various bounds on expressions used in the result established in Section 4.1, we chose to use the Coq-interval [15] library. The library supports a high degree of automation, to establish approximate preliminary bounds on certain standard functions like the square root function, polynomials, trigonometric functions, the exponential function and the logarithm. It uses Taylor models (as defined in [14]) to establish such bounds.

However, this library by itself doesn't prove to be sufficient, since it relies on Coquelicot [8] to prove certain results and doesn't provide automation to use them. In order to setup an environment compatible with these results, we use Coquelicot as well.

Additionally, we rely on the injections of natural numbers and integers into reals,(in particular, to establish the *increasing k* invariant from Section 4.2) and the proven ring homomorphism properties of these injection in the Coq standard library, to argue about integers in bytecode inside reals and then transport established inequalities over reals back to inequalities over integers.

5.2 Proof Outline

The formalization[4] of the above properties of the constant product market maker protocol in the Coq proof assistant requires the use of real analysis results.

To prove the lower bound in Theorem 2 in the first case, we use the Taylor series approximation for continuous and twice differentiable functions. We state and prove the Taylor series approximation for the function, $\sqrt{1+\epsilon} + 1/\sqrt{1+\epsilon} - 2$ in the interval $(0, 1]$. The lemma is stated as follows:

```
Lemma taylor_m : 0 < eps <= 1 ->
exists eta,
 (0 <> eps -> (0 < eta < eps \/ eps < eta < 0)) /\
 sqrt (1 + eps) + 1/sqrt(1 + eps) -2 =
 (((2 - eta) / (8* ((1 + eta)^2) * sqrt (1 + eta))) * eps^2).
```

We use the general version of the Taylor Lagrange theorem formalized in the Coq-interval library to prove the above lemma. [14] The statement of the theorem is as follows:

```
Section TaylorLagrange.
Variables a b : R.
Variable n : nat.
Notation Cab x := (a <= x <= b) (only parsing).
Notation Oab x := (a < x < b) (only parsing).
Variable D : nat -> R -> R.
Notation Tcoeff n x0 := (D n x0 / (INR (fact n))) (only parsing).
Notation Tterm n x0 x := (Tcoeff n x0 * (x - x0)^n) (only parsing).
Notation Tsum n x0 x := (sum_f_R0 (fun i => Tterm i x0 x) n) (only parsing).
Section TL.

Hypothesis derivable_pt_lim_Dp :
  forall k x, (k <= n)%nat -> Oab x ->
  derivable_pt_lim (D k) x (D (S k) x).

Hypothesis continuity_pt_Dp :
  forall k x, (k <= n)%nat -> Cab x ->
  continuity_pt (D k) x.
Variables x0 x : R.
Theorem Taylor_Lagrange :
  exists xi : R,
  D 0 x - Tsum n x0 x =
  Tcoeff (S n) xi * (x - x0)^(S n)
  /\ (x0 <> x -> x0 < xi < x \/ x < xi < x0).
End TL.
End TaylorLagrange.
```

In the above setting, the function $D : nat \to \mathbb{R} \to \mathbb{R}$ represents the series of a function and it's *nth* derivative, where $D0$ is the function itself and Dn is it's *(n-1)th* derivative. *Tsum n x0 x* is the sum of the first n terms of the Taylor series of the function $D0$. Since, a necessary condition for the Taylor Lagrange theorem to hold is that if the approximation is of the *nth* order then each of the *kth* order derivatives of the function should be continuous and

[4] Available online at `https://github.com/certikfoundation/deepsea/blob/master/contracts/amm/cst_man.v`

differentiable for all $k \leq n$, the section in the Taylor.v module includes a hypothesis, which we must prove in order to apply the theorem. We define the following function and use it in place of the above function D for our application of the Taylor Lagrange theorem:

```
Definition T_f_1 (n : nat) :=
match n with
 | 0%nat => (fun x => sqrt(1+x) + (1/sqrt (1 + x)))
 | 1%nat => (fun x => 1/(2* sqrt(1+x)) - (1/ (2 * (1 + x) * (sqrt (1 + x)))))
 | 2%nat => (fun x => (2 - x)/ (4 * ((1 + x)^2) * sqrt(1 +x)))
 | _ => (fun x => 0%R)
end.
```

The required hypothesis are stated and proved as the following lemmas:

```
Lemma deriv_lim_T_f : forall (k : nat) (x : R),
(k <= 1)%nat ->
0 < x < 1 ->
derivable_pt_lim (T_f_1 k) x (T_f_1 (S k) x).

Lemma cont_lim_T_f : forall (k : nat) (x : R),
(k <= 1)%nat ->
0 <= x <= 1 -> continuity_pt (T_f_1 k) x.
```

Once, we have the Taylor approximation, we establish a lower bound on the remainder term using the powerful interval tactic. This can be done since we have a range in which ϵ lies for the first lower bound on the cost of manipulation (i.e $0 \leq \epsilon \leq 1$). The lemma for the lower bound on the remainder term is stated as follows:

```
Lemma lower_bnd : forall eta, 0 <= eta <= 1 ->
(2 - eta) / (8 * ((1 + eta)^2) * sqrt (1 + eta)) >= 1 / 48.
```

Note that, the lower bound in ([4]) is $1/32\sqrt{2}$. Since the interval tactic works based on approximations [15], it cannot be used to prove exact irrational bounds. Hence, we approximate $\sqrt{2}$ with 3/2 to get a lower bound of 1/48. This lower bound can be made closer to $1/32\sqrt{2}$, by using a finer approximation.This gives us the first part of the lower bound in Theorem 2.

To prove the second part, we use a different approach from the one used in [4]. The lower bound for the case when $\epsilon \geq 1$ involves proving the following inequality:

$$x + x^{-1} - 2 \geq \kappa x$$

where $x = \sqrt{1+\epsilon}$, $\kappa = 3/2 - \sqrt{2}$. Here again we approximate κ by 5/100 to facilitate the use of the interval tactic. Instead of using the analysis of quadratic equations approach as in ([4]), we use a simpler way. After a small algebraic manipulation, and accounting for approximations the above inequality can be re-written as the following lemma:

```
Lemma eps_sq : eps >= 1 ->
(sqrt(1 + eps) - 1)^2  - ((5/100) * (1 + eps)) >= 0.
```

To show this, we use the fact that if the derivative of a continuous function defined on a connected domain is positive, then the function is increasing. This coupled with the observation that the value of the l.h.s of the above inequality evaluated at 1 is positive, gives us the proof. Thus we have the lower bound on the cost of manipulation for the second case :

```
Lemma cst_func_ge_1 : eps >= 1 ->
sqrt ( 1 + eps) + (1 / sqrt (1 + eps)) -2 >= ((5/100) * sqrt (1 + eps)).
```

Combining them gives us the following mechanization of Theorem 2 :

```
Definition cost_of_manipulation_val := (IZR (reserve_beta s)) *
(sqrt (1 + eps) + (1/ sqrt (1 + eps)) - 2).

Theorem cost_of_manipulation_min : eps >= 0 ->
cost_of_manipulation_val >= (IZR (reserve_beta s)) * (5/100) * sqrt (eps) \/
cost_of_manipulation_val >= (IZR (reserve_beta s)) * (1/48) * (eps^2).
```

We also mechanize the no-depletion property from Section 4.2. The formalized version of Theorem 3 holds for the AMM contract written in DeepSEA:

```
Theorem no_depletion_reserves : (IZR (reserve_beta s'')) +
(IZR (reserve_alpha s'')) > 0.
```

Proving this result requires establishing the increasing product invariant over integers. This requires some careful reasoning over reals before the result is transported back to integers. The increasing product invariant is stated as follows:

```
Lemma increasing_k : Z.lt (compute_k s) (compute_k s'').
```

The above lemma states that the product of the reserves always increases with each swap operation. Thus, we establish two independent important algorithmic properties of the bytecode corresponding to the DeepSEA AMM contract.

5.3 Connection to the DeepSEA contract

The results we formalized in the Coq Proof assistant about the cost of manipulation of the market price are for the AMM contract written in DeepSEA. The *cst_man.v* module imports the Coq files generated by the DeepSEA compiler, so that computations can be made using the variables of the contract. We want to know how prices are affected by a single call to the `simpleSwap0` function, we do so by adding the following hypothesis to our file:

```
Hypothesis del_alp : runStateT (AutomatedMarketMaker_simpleSwap0_opt
toA (make_machine_env a)) s = Some (r' , s'').
```

Here `AutomatedMarketMaker_simpleSwap0_opt` is an automatically generated Coq function which represents the behaviour of the contract method. The hypothesis says that a call to it, exchange of the token α for the input token β ,completed without reverting and left us in a new contract state `s'`.

This is done to calculate Δ_α. Once the values $R_\alpha, R_\beta, \Delta_\alpha, \Delta_\beta$ are obtained from the contract, the fraction by which the exchange price can be manipulated (i.e ϵ) is computed using the formula:

$$\frac{R_\beta + \Delta_\beta}{R_\alpha - \Delta_\alpha} = (1+\epsilon)\frac{\Delta_\beta}{\Delta_\alpha}$$

Now the results stated in 4.1 are formalized for the ϵ obtained from above and its possible values.

6 Related Work

We are only aware of one mechanized proof applied to a DeFi contract: Park et al.'s verification of the original Uniswap AMM using the KEVM Framework [23]. They prove that the functions implemented by the contract bytecode conforms to a high-level specification (the constant-product formulas), and they do not prove any financial correctness properties. In other words, the end result of the verification is a set of high-level functions similar to what DeepSEA generates automatically and we take as our starting point; however, this is done for the already existing and deployed contract, while DeepSEA requires you to re-implement the contract in the DeepSEA language.

As for paper proofs, we already discussed Angeris and Chitra's theorem [4], which we mechanize in this paper. Angeris et al. [3] consider generalizations of the Uniswap formula and further desirable properties. Bartoletti et al set out to understand DeFi protocols by writing down (on paper) abstract models of AMMs [7] and lending pools [6] as transition systems, and then proving theorems such as demand-sensitivity and non-depletion about them. In future work, we aim to provide machine-checked proofs of many such properties, as we have already done for non-depletion.

An alternative to formal proof is to apply model checking [21, 19] or graph search with contraints [24] to find DeFi hacks. These works manually translate and simplify the set of contracts into a language the model checker can deal with, and can then make a best effort to find exploits. Because model checking is automatic (so it requires less user effort) we believe this can be a useful complement.

The kind of oracle we consider provides pricing information based directly on on-chain trades. This should be distinguished from oracles that aggregates data from off-chain sources and posts them on the blockchain. Analyses of the latter [11, 13] show that they, too, have problems with spurious data spikes and possible attacks.

7 Conclusions and Future Work

We take the first step in formally verifying financial properties of the contract at the algorithmic level. This is enabled by the Coq functional model that is automatically generated by the DeepSEA compiler. Not only is the compilation to bytecode verified, but we also have a formalization of the desirable properties of the Constant Product Market Maker model which is directly tied to the DeepSEA AMM contract. Hence we have a verified specification and verified code.

In the future, we want to extend this line of work in two directions. First, we want to consider theorems about more advanced kinds of oracles. The non-manipulation result we formalized is still the state of the art when it comes to mathematical analysis of oracles, but more recent developments have made it partially obsolete. As we explained in Section 5.2, the assumption is that the attacker makes a loss because he bought tokens at a too-high price. But in a scenario involving flash-loans this is not necessarily the case, because in a single transaction the attacker can carry out the manipulation and then immediately sell back the tokens again. There is not enough time for other market participants to exploit the incorrect price through arbitrage. Similar considerations apply if the attacker is colluding with a miner to control the order of transactions in a block. Newer oracles such as Uniswap v2 [1] and Uniswap v3 [2] are more robust and hacker-resistant than the simple instantaneous-price oracle that we consider, because they average the price over a longer time period. Theoretical results about such models however still remain to be established.

Second, it will be interesting to prove the correctness of a *client* of an oracle, e.g. to give bounds on when a lending protocol can become undercollateralized. Just like the DeFi applications themselves are built from "money lego", we hope that they can be verified by composing together theorems about the individual components.

References

1 Hayden Adams, Noah Zinsmeister, and Dan Robinson. Uniswap v2 core. 2020, 2020. URL: `https://uniswap.org/whitepaper.pdf`.

2 Hayden Adams, Noah Zinsmeister, Moody Salem, River Keefer, and Dan Robinson. Uniswap v3 core, 2021.

3 Guillermo Angeris and Tarun Chitra. Improved price oracles: Constant function market makers. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 80–91, 2020.

4 Guillermo Angeris, Hsien-Tang Kao, Rei Chiang, Charlie Noyes, and Tarun Chitra. An analysis of uniswap markets. *arXiv preprint*, 2019. `arXiv:1911.03380`.

5 Korantin Auguste. The bzx attacks explained. Blog post. `https://www.palkeo.com/en/projects/ethereum/bzx.html#second-transaction.`, 2020. (Accessed on 05/23/2021).

6 Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. Sok: Lending pools in decentralized finance. *CoRR*, abs/2012.13230, 2020. `arXiv:2012.13230`.

7 Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. A theory of automated market makers in defi. *arXiv preprint*, 2021. `arXiv:2102.11350`.

8 Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for coq. *Mathematics in Computer Science*, 9(1):41–62, 2015.

9 Vitalik Buterin. On path independence. Blog post, 2017. URL: `https://vitalik.ca/general/2017/06/22/marketmakers.html`.

10 CertiK Foundation. DeepSEA. (Accessed on 05/23/2021). URL: `https://github.com/certikfoundation/deepsea`.

11 Wanyun Catherine Gu, Anika Raghuvanshi, and Dan Boneh. Empirical measurements on pricing oracles and decentralized governance for stablecoins. *Available at SSRN 3611231*, 2020.

12 Xavier Leroy. The CompCert verified compiler. `http://compcert.inria.fr/`, 2005–2021.

13 Bowen Liu, Pawel Szalachowski, and Jianying Zhou. A first look into defi oracles. *arXiv preprint*, 2020. `arXiv:2005.04377`.

14 Érik Martin-Dorel, Laurence Rideau, Laurent Théry, Micaela Mayero, and Ioana Pasca. Certified, efficient and sharp univariate taylor models in coq. In *2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 193–200. IEEE, 2013.

15 Guillaume Melquiond. Coq-interval. *Retrieved June*, 17:2017, 2011.

16 miscellanous. Cryptocurrency statistics. Blog post, 2020. URL: `https://duneanalytics.com/queries/4494/8769`.

17 miscellanous. Defi statistics. Blog post, 2020. URL: `https://cointelegraph.com/news/defi-hacks-and-exploits-total-285m-since-2019-messari-reports`.

18 Vilhelm Sjöberg, Yuyang Sang, Shu-chun Weng, and Zhong Shao. DeepSEA: a language for certified system software. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–27, 2019.

19 Xinyuan Sun, Shaokai Lin, Vilhelm Sjöberg, and Jay Jie. How to exploit a defi project (extended talk abstract). Talk at the 1st Workshop on Decentralized Finance (DeFi), colocated with Financial Cryptography and Data Security 2021 (fc21), March 2021.

20 The Coq Development Team. The Coq proof assistant. `https://coq.inria.fr/`. Accessed: 28/5/2019.

21 Palina Tolmach, Yi Li, Shang-Wei Lin, and Yang Liu. Formal analysis of composable defi protocols. *CoRR*, abs/2103.00540, 2021. `arXiv:2103.00540`.

22 Dabao Wang, Siwei Wu, Ziling Lin, Lei Wu, Xingliang Yuan, Yajin Zhou, Haoyu Wang, and Kui Ren. Towards understanding flash loan and its applications in defi ecosystem. *CoRR*, abs/2010.12252, 2020. `arXiv:2010.12252`.

23 Daejun Park Yi Zhang, Xiaohong Chen. Formal specification of constant product market maker model and implementation, 2018. URL: `https://github.com/runtimeverification/verified-smart-contracts/blob/uniswap/uniswap/x-y-k.pdf`.

24 Liyi Zhou, Kaihua Qin, Antoine Cully, Benjamin Livshits, and Arthur Gervais. On the just-in-time discovery of profit-generating transactions in defi protocols. *arXiv preprint*, 2021. `arXiv:2103.02228`.