# Introduction

A price oracle is any tool used to view price information about a given asset. When you look at stock prices on your phone, you are using your phone as a price oracle. Similarly, the app on your phone relies on devices to retrieve price information - likely several, which are aggregated and then displayed to you, the end-user. These are price oracles as well.

When building smart contracts that integrate with DeFi protocols, developers will inevitably run into the price oracle problem. What is the best way to retrieve the price of a given asset on-chain?

Many oracle designs on Ethereum have been implemented on an ad-hoc basis, with varying degrees of decentralization and security. Because of this, the ecosystem has witnessed numerous high-profile hacks where the oracle implementation is the primary attack vector. Some of these vulnerabilities are discussed [here](https://samczsun.com/taking-undercollateralized-loans-for-fun-and-for-profit/).

While there is no one size fits all solution, Uniswap V2 enables developers to build highly decentralized and manipulation-resistant on-chain price oracles, which may solve many of the demands necessary for building robust protocols.

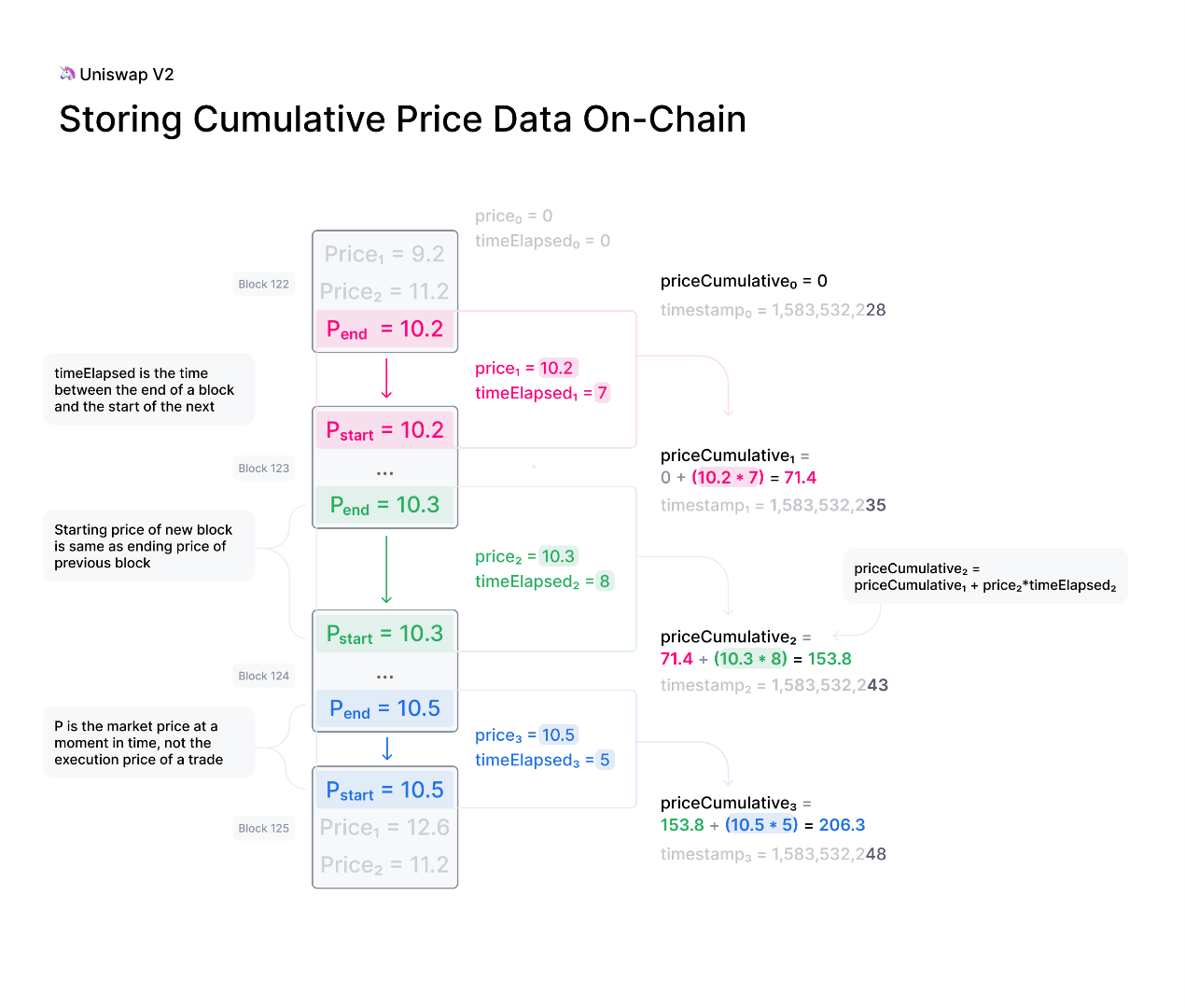
# Uniswap V2 solution

Uniswap V2 includes several improvements for supporting manipulation-resistant public price feeds. First, every pair measures (but does not store) the market price at the beginning of each block, before any trades take place. This price is expensive to manipulate because it is set by the last transaction, whether it is a mint, swap, or burn, in a previous block.

**To set the measured price to one that is out of sync with the global market price, an attacker has to make a bad trade at the end of a previous block** , typically with no guarantee that they will arbitrage it back in the next block. Attackers will lose money to arbitrageurs unless they can “selfishly” mine two blocks in a row. This type of attack presents several challenges and [has not been observed to date](https://arxiv.org/abs/1912.01798).

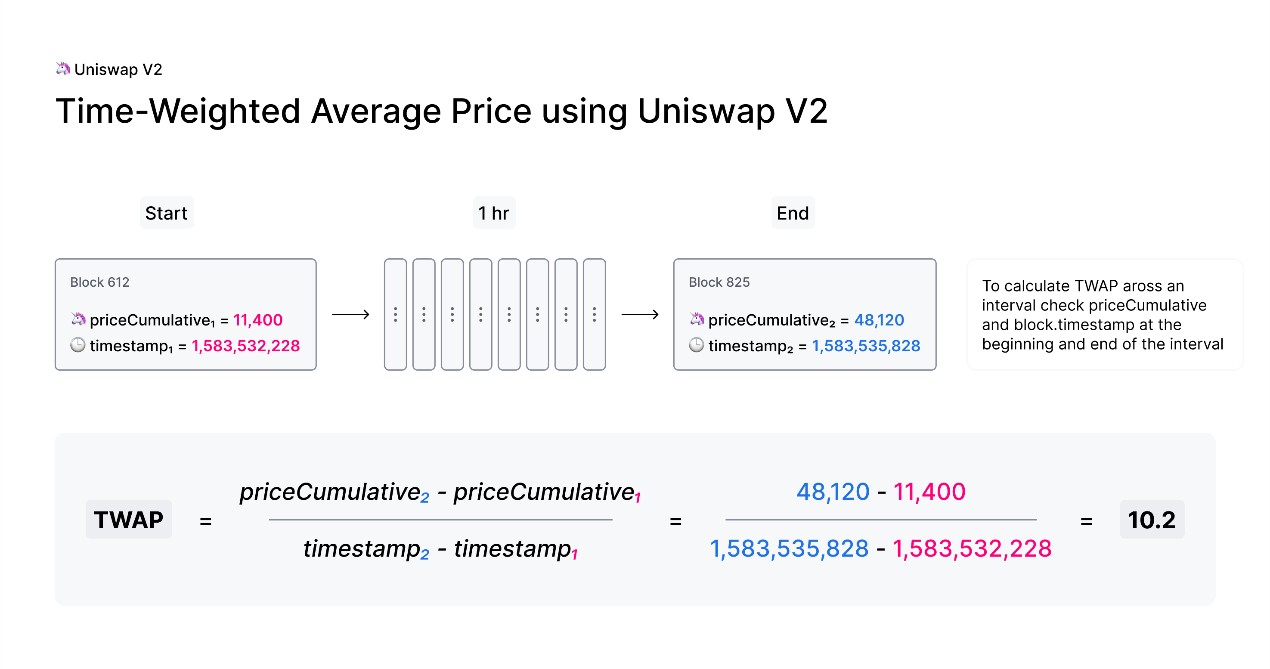
Unfortunately, this alone is not enough. If significant value settles based on the price resulting from this mechanism, an attack’s profit will likely outweigh the loss.

Instead, Uniswap V2 adds this end-of-block price to a single cumulative-price variable in the core contract weighted by the amount of time this price existed. **This variable represents a sum of the Uniswap price for every second in the entire history of the contract.**



This variable can be used by external contracts to track accurate time-weighted average prices (TWAPs) across any time interval.

The TWAP is constructed by reading the cumulative price from an ERC20 token pair at the beginning and at the end of the desired interval. The difference in this cumulative price can then be divided by the length of the interval to create a TWAP for that period.



TWAPs can be used directly or as the basis for moving averages (EMAs and SMAs) as needed.

A few notes:

* For a 10-minute TWAP, sample once every 10 minutes. For a 1-week TWAP, sample once every week.
* For a simple TWAP, the cost of manipulation increases (approx. linear) with liquidity on Uniswap, as well as (approx. linear) with the length of time over which you average.
* The Cost of an attack is relatively simple to estimate. Moving the price 5% on a 1-hour TWAP is approximately equal to the amount lost to arbitrage and fees for moving the price 5% every block for 1 hour.

There are some nuances that are good to be aware of when using Uniswap V2 as an oracle, especially where manipulation resistance is concerned. The [whitepaper](https://docs.uniswap.org/whitepaper.pdf) elaborates on some of them. Additional oracle-focused developer guides and documentation will be released soon.

In the meantime, check out our [example implementation](https://github.com/Uniswap/uniswap-v2-periphery/blob/master/contracts/examples/ExampleOracleSimple.sol) of a 24 hr TWAP Oracle built on Uniswap V2!

## Manipulation resistance[​](https://docs.uniswap.org/contracts/v2/concepts/core-concepts/oracles#manipulation-resistance)

The cost of manipulating the price for a specific time period can be roughly estimated as the amount lost to arbitrage and fees every block for the entire period. For larger liquidity pools and over longer time periods, this attack is impractical, as the cost of manipulation typically exceeds the value at stake.

Other factors, such as network congestion, can reduce the cost of attack. For a more in-depth review of the security of Uniswap V2 price oracles, read the [security audit section on Oracle Integrity](https://uniswap.org/audit.html#org87c8b91).

# Building an oracle

To learn more about building oracles check out [building an Oracle](https://docs.uniswap.org/contracts/v2/guides/smart-contract-integration/building-an-oracle) in the developer guides.

Building an Oracle

To build a price oracle on Uniswap V2, you must first understand the requirements for your use case. Once you understand the kind of price average you require, it is a matter of storing the cumulative price variable from the pair as often as necessary, and computing the average price using two or more observations of the cumulative price variables.

Understanding requirements

To understand your requirements, you should first research the answer to the following questions:

* Is data freshness important? I.e.: must the price average include the current price?
* Are recent prices more important than historical prices? I.e.: is the current price given more weight than historical prices?

Note your answers for the following discussion.

Oracle Strategies

Fixed windows[​](https://docs.uniswap.org/contracts/v2/guides/smart-contract-integration/building-an-oracle#fixed-windows)

In the case where data freshness is not important and recent prices are weighted equally with historical prices, it is enough to store the cumulative price once per period (e.g. once per 24 hours.)

Computing the average price over these data points gives you 'fixed windows', which can be updated after the lapse of each period. We wrote an example oracle of this kind [here](https://github.com/Uniswap/uniswap-v2-periphery/blob/master/contracts/examples/ExampleOracleSimple.sol).

This example does not limit the maximum size of the fixed window, i.e. it only requires that the window size is greater than 1 period (e.g. 24 hours).

Moving averages[​](https://docs.uniswap.org/contracts/v2/guides/smart-contract-integration/building-an-oracle#moving-averages)

In the case where data freshness is important, you can use a sliding window in which the cumulative price variable is measured more often than once per period.

There are at least [two kinds of moving averages](https://www.investopedia.com/terms/m/movingaverage.asp#types-of-moving-averages) that you can compute using the Uniswap cumulative price variable.

[Simple moving averages](https://www.investopedia.com/terms/s/sma.asp) give equal weight to each price measurement. We have built an example of a sliding window oracle [here](https://github.com/Uniswap/uniswap-v2-periphery/blob/master/contracts/examples/ExampleSlidingWindowOracle.sol).

[Exponential moving averages](https://www.investopedia.com/terms/e/ema.asp) give more weight to the most recent price measurements. We do not yet have an example written for this type of oracle.

You may wish to use exponential moving averages where recent prices are more important than historical prices, e.g. in case of liquidations. However, note that putting more weight on recent prices makes the oracle cheaper to manipulate than weighting all price measurements equally.

Computing average prices[​](https://docs.uniswap.org/contracts/v2/guides/smart-contract-integration/building-an-oracle#computing-average-prices)

To compute the average price given two cumulative price observations, take the difference between the cumulative price at the beginning and end of the period, and divide by the elapsed time between them in seconds. This will produce a [fixed point unsigned Q112x112](https://en.wikipedia.org/wiki/Fixed-point_arithmetic#Notation) number that represents the price of one asset relative to the other. This number is represented as a uint224 where the upper 112 bits represent the integer amount, and the lower 112 bits represent the fractional amount.

Pairs contain both price0CumulativeLast and price1CumulativeLast, which are ratios of reserves of token1/token0 and token0/token1 respectively. I.e. the price of token0 is expressed in terms of token1/token0, while the price of token1 is expressed in terms of token0/token1.

Getting the latest cumulative price

If you wish to compute the average price between a historical price cumulative observation and the current cumulative price, you should use the cumulative price values from the current block. If the cumulative price has not been updated in the current block, e.g. because there has not been any liquidity event (mint/burn/swap) on the pair in the current block, you can compute the cumulative price counterfactually.

We provide a library for use in oracle contracts that has the method [UniswapV2OracleLibrary#currentCumulativePrices](https://github.com/Uniswap/uniswap-v2-periphery/blob/master/contracts/libraries/UniswapV2OracleLibrary.sol#L16) for getting the cumulative price as of the current block. The current cumulative price returned by this method is computed *counterfactually*, meaning it requires no call to the relative gas-expensive #sync method on the pair. It is correct regardless of whether a swap has already executed in the current block.

Notes on overflow

The UniswapV2Pair cumulative price variables are designed to eventually overflow, i.e. price0CumulativeLast and price1CumulativeLast and blockTimestampLast will overflow through 0.

This should not pose an issue to your oracle design, as the price average computation is concerned with differences (i.e. subtraction) between two separate observations of a cumulative price variable. Subtracting between two cumulative price values will result in a number that fits within the range of uint256 as long as the observations are made for periods of max 2^32 seconds, or ~136 years.

blockTimestampLast is stored only in a uint32. For the same reason as described above, the pair can save a storage slot, and many SSTORES over the life of the pair, by storing only block.timestamp % uint32(-1). This is feasible because the pair is only concerned with the time that elapses between each liquidity event when updating the cumulative prices, which is always expected to be less than 2^32 seconds.

When computing time elapsed within your own oracle, you can simply store the block.timestamp of your observations as uint256, and avoid dealing with overflow math for computing the time elapsed between observations. This is how the [ExampleSlidingWindowOracle](https://github.com/Uniswap/uniswap-v2-periphery/blob/master/contracts/examples/ExampleSlidingWindowOracle.sol) handles observation timestamps.

Integrating the oracle[​](https://docs.uniswap.org/contracts/v2/guides/smart-contract-integration/building-an-oracle#integrating-the-oracle)

To integrate an oracle into your contracts, you must ensure the oracle's observations of the cumulative price variable are kept up to date. As long as your oracle is up to date, you can depend on it to produce average prices. The process of keeping your oracle up to date is called 'maintenance'.

Oracle maintenance[​](https://docs.uniswap.org/contracts/v2/guides/smart-contract-integration/building-an-oracle#oracle-maintenance)

In order to measure average prices over a period, the oracle must have a way of referencing the cumulative price at the start and end of a period. The recommended way of doing this is by storing these prices in the oracle contract, and calling the oracle frequently enough to store the latest cumulative price.

Reliable oracle maintenance is a difficult task, and can become a point of failure in times of congestion. Instead, consider building this functionality directly into the critical calls of your own smart contracts, or incentivize oracle maintenance calls by other parties.

No-maintenance option[​](https://docs.uniswap.org/contracts/v2/guides/smart-contract-integration/building-an-oracle#no-maintenance-option)

It is possible to avoid regularly storing this cumulative price at the start of the period by utilizing storage proofs. However, this approach has limitations, especially in regard to gas cost and maximum length of the time period over which the average price can be measured. If you wish to try this approach, you can follow [this repository by Keydonix](https://github.com/Keydonix/uniswap-oracle/).

Keydonix has developed a general purpose price feed oracle built on Uniswap v2 that supports arbitrary time windows (up to 256 blocks) and doesn't require any active maintenance.