

UNISWAP

WITH PYTHON-IMPLEMENTED SIMULATOR

REFERENCE

- ▶ Zhang, Yi, Xiaohong Chen, and Daejun Park.
"Formal Specification of Constant Product
($x \times y = k$) Market Maker Model and Implementation."
URL: <https://github.com/runtimeverification/verifiedsmart-contracts/blob/uniswap/uniswap/xy-k.pdf>,
Accessed: December 30 (2018): 2019.
- ▶ <https://github.com/lukepark327/uniswap-python>
- ▶ <https://medium.com/curg/유니스왑-이모저모-살펴보기-e2ee091f3aef>

BACKGROUNDS

BACKGROUNDS: ERC20 TOKEN

▶ 토큰

- ▶ 스마트 컨트랙트를 통해 저장되거나 수정된 상태는 신뢰할 수 있음
- ▶ 이를 이용해 쉽게 암호화폐를 제작
- ▶ 스마트 컨트랙트로 만들어진 암호화폐를 토큰(Token)이라고 칭함

BACKGROUNDS: ERC20 TOKEN

▶ ERC20

- ▶ 이더리움 스마트 컨트랙트로 만드는 암호화폐인 토큰에 대한 표준 규격
- ▶ ERC20 토큰 간에는 손쉽게 호환성을 제공할 수 있음
- ▶ 구현의 간편함과 호환성을 위해,
이더리움상에서 구현된 토큰 대다수가 ERC20 표준을 준수

BACKGROUNDS: DEFI

- ▶ 기존 금융 시스템은 은행과 같은 중앙 기관의 주도로 동작
 - ▶ 중앙 기관에서 시스템의 안전을 보장
- ▶ 디파이(DeFi)
 - ▶ 탈중앙화 금융(Decentralized Financial)의 약자
 - ▶ 중앙 기관이 없는 탈중앙화 환경에 기반하여 금융 시스템을 제공

BACKGROUNDS: DEFI

- ▶ 최근의 디파이
 - ▶ 효율성과 안전성을 위해 신뢰할 수 있는 환경인 블록체인에 기반
 - ▶ 암호화폐를 담보로 일정 금액을 대출받거나
 - ▶ 서로 다른 암호화폐간의 교환, 상품 거래, 스테이블 코인 발행 등
 - ▶ 금융 서비스를 제공

OVERVIEW

OVERVIEW

- ▶ 유니스왑(Uniswap)
 - ▶ 이더리움에서 자동화된 유동성 공급을 위한 프로토콜이자 디파이 서비스
 - ▶ 유니스왑에서는 ETH와 ERC20 토큰 간의 거래,
 - ▶ 혹은 ERC20토큰과 다른 ERC20 토큰간의 거래를 지원

OVERVIEW

- ▶ 기존의 P2P(Peer-to-peer) 거래
 - ▶ 구매자와 판매자가 반드시 한 쌍 존재해야 함
 - ▶ 이들의 희망 매매가격이 일치하는 경우에만 거래가 성사
- ▶ 유니스왑에서는 판매자라는 주체가 요구되지 않음
 - ▶ 오직 구매자와 유동성 공급자만이 존재
 - ▶ 구매자는 ETH를 토큰으로 교환하거나, 토큰을 ETH 또는 다른 토큰으로 교환하는 행위를 판매자 없이 행할 수 있음
- ▶ 이렇게 함으로써 실시간 유동성이 없는 환경에서도 거래가 가능

OVERVIEW

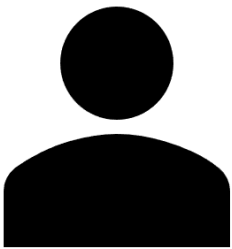
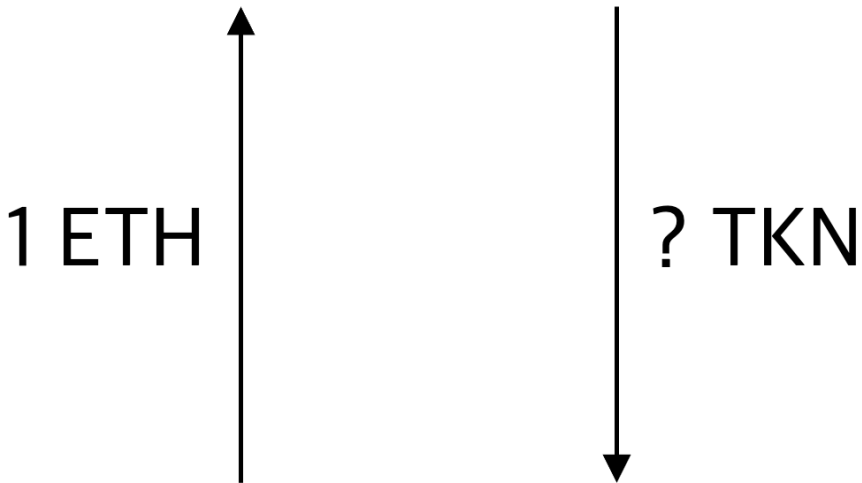
- ▶ 풀(pool)이라는 개념 덕분
 - ▶ ETH풀은 다량의 ETH를 보유하고 있으며,
 - ▶ 토큰 풀은 토큰을 보유
- ▶ ETH를 토큰으로 교환하고자 하는 구매자
 - ▶ ETH 풀에 ETH를 전송
 - ▶ 프로토콜에 의해 계산된 양의 토큰을 받음
- ▶ 반대의 경우도 마찬가지

OVERVIEW

► ETH to TKN

ETH	= 10
TKN	= 500
k	= $10 * 500 = 5000$

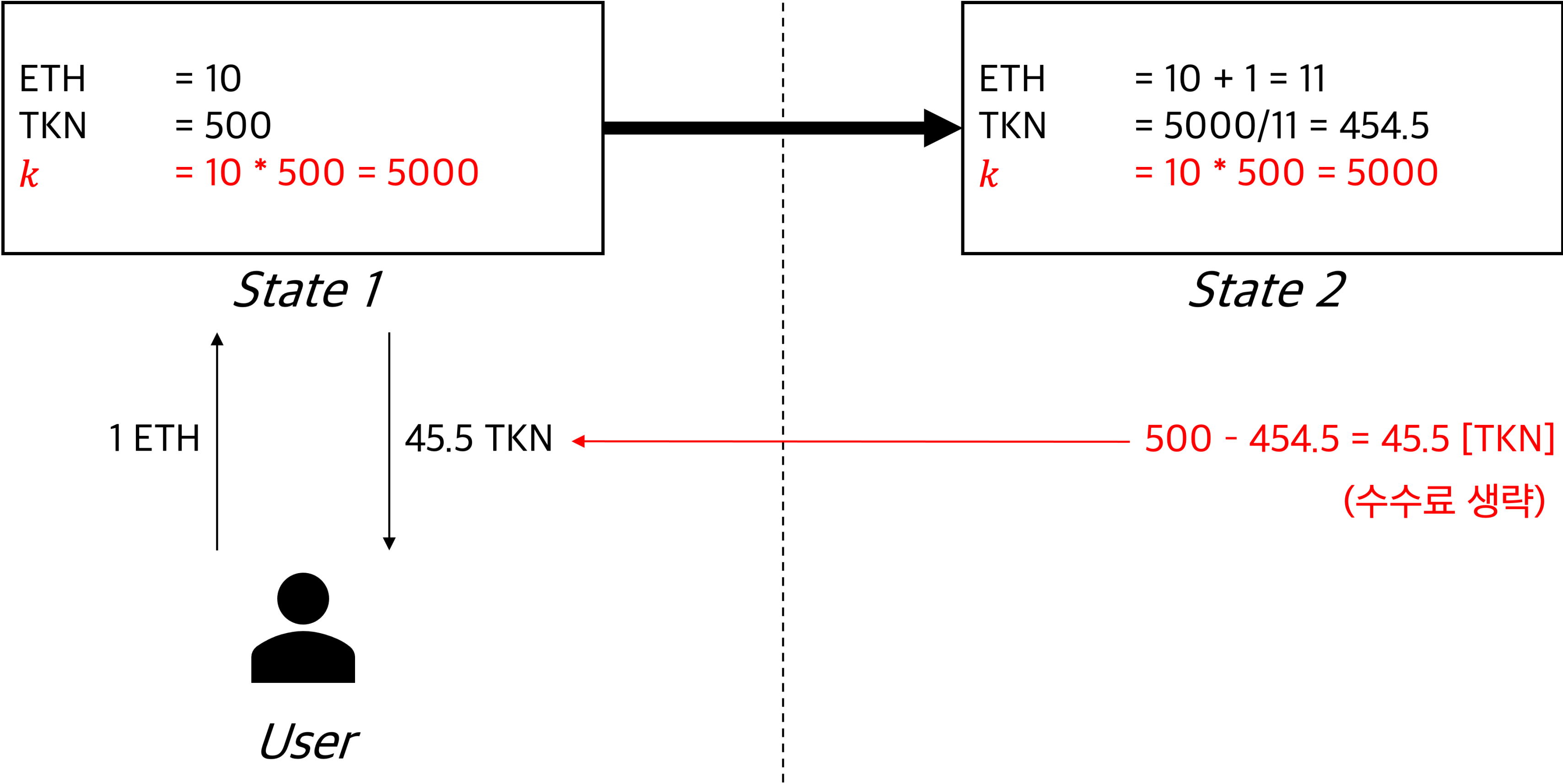
State 1



User

OVERVIEW

▶ ETH to TKN



OVERVIEW

- ▶ 유동성 공급자
 - ▶ 거래로 인해 발생할 수 있는 풀의 불균형을 조율
 - ▶ 양 풀의 자금을 공급하는 역할을 담당
- ▶ 유동성을 공급하기 위해 자금을 묶어두는 대가로
 - ▶ 0.3%의 거래 수수료를 취함

$$x \times y = k$$

MODEL

$x \times y = k$ MODEL

- ▶ 암호화폐 X 와 Y 사이의 Swap
- ▶ X 풀의 수량 x , Y 풀의 수량 y
- ▶ X 를 Y 로 교환하는 경우를 주로 다룸
 - ▶ 그 역을 동일하게 계산 가능

$x \times y = k$ MODEL: SWAP

- ▶ 거래 수수료가 없는 상황에서, X 와 Y 사이의 거래는 다음을 만족
 - ▶ 거래 전과 후의 상태 모두 $x \times y = k$ 를 유지
 - ▶ 이러한 특성 때문에 k 를 ‘상수 곱(constant product)’으로 칭함
- ▶ Δx 개의 토큰 X 를 Δy 개의 토큰 Y 와 거래하려는 상황
 - ▶ $x \times y = k = (x + \Delta x) \times (y - \Delta y)$

$x \times y = k$ MODEL: SWAP

▶ 거래 이후의 토큰 X 의 개수 x' 과 Y 의 개수 y'

▶ $x' = x + \Delta x = (1 + \alpha)x = \frac{1}{1 - \beta}x$

▶ $y' = y - \Delta y = \frac{1}{1 + \alpha}y = (1 - \beta)y$

▶ 여기서 $\alpha = \frac{\Delta x}{x}$, $\beta = \frac{\Delta y}{y}$

▶ 풀의 기존 잔액 대비 변화량의 비(ratio)

$x \times y = k$ MODEL: SWAP

▶ 거래 수수료를 고려

- ▶ 풀의 유동성 공급을 촉진하고 균형을 맞추기 위해 ‘거래 수수료’ 개념이 존재
- ▶ 모든 유니스왑 거래에 일정 비율로 부과
- ▶ 유동성 공급자들 각자가 풀에 기여한 지분만큼 분할 수여

▶ 수식에서 ρ 로 표기

- ▶ 0.3%의 수수료는 $\rho = 0.003$ 에 해당
- ▶ $0 \leq \rho < 1$

$x \times y = k$ MODEL: SWAP

- ▶ 거래 수수료를 고려

- ▶ $y' = y - \Delta y$ 계산 시 변화량의 비 $\alpha = \frac{\Delta x}{x}$ 에 대해 거래 수수료를 부과

- ▶ α 대신에 $\alpha(1 - \rho)$ 를 사용

- ▶ 단순화를 위해 $\gamma = 1 - \rho$ 기호를 사용

$x \times y = k$ MODEL: SWAP

▶ 거래 수수료를 고려

$$\text{▶ } y' = y - \Delta y = \frac{1}{1 + \alpha\gamma} y = (1 - \beta)$$

$$\text{▶ } \Delta y = \frac{\alpha\gamma}{1 + \alpha\gamma} y$$

▶ α 를 β 에 대해 정리하면

$$\text{▶ } \alpha = \frac{\beta}{(1 - \beta)\gamma}$$

$x \times y = k$ MODEL: SWAP

▶ 거래 수수료를 고려

$$\text{▶ } x' = x + \Delta x = (1 + \alpha)x = \left(1 + \frac{\beta}{(1 - \beta)\gamma}\right)x$$

$$\text{▶ } \Delta x = \frac{\beta}{(1 - \beta)\gamma}x$$

$x \times y = k$ MODEL: SWAP

- ▶ 거래 수수료가 없는 상황 ($\gamma = 1$)
 - ▶ $x \times y = x' \times y'$
- ▶ 거래 수수료가 있는 상황 ($0 < \gamma < 1$)
 - ▶ $x \times y < x' \times y'$
 - ▶ k 값이 소폭 증가
 - ▶ 이러한 기작으로부터 유동성 공급자가 유동성을 회수할 때 이익을 발생시킴

$x \times y = k$ MODEL: SWAP

▶ 교환 금액 계산

▶ Δx 개의 토큰 X 로 구매할 수 있는 토큰 Y 의 개수 Δy

$$\Delta y = \frac{\alpha \gamma}{1 + \alpha \gamma} y, \alpha = \frac{\Delta x}{x}$$

▶ Δy 개의 토큰 Y 를 사기 위해 필요한 토큰 X 의 개수 Δx

$$\Delta x = \frac{\beta}{(1 - \beta)\gamma} x, \beta = \frac{\Delta y}{y}$$

$x \times y = k$ MODEL: SWAP

- ▶ 교환 금액 계산
- ▶ 다음 상태 x', y', k' 의 변화
 - ▶ $x < x'$
 - ▶ $y > y'$
 - ▶ $k < k'$

$x \times y = k$ MODEL: LIQUIDITY

- ▶ 유동성 공급자
 - ▶ 특정 비율(가령 0.3%)의 거래 수수료
 - ▶ 유동성에 기여한 지분만큼 가져감
- ▶ 지분을 평가하기 위해
 - ▶ 유동성 토큰(Liquidity Token, LT)이라는 별도의 유틸리티 토큰을 활용
 - ▶ 유동성 토큰의 현재까지의 총 발행량을 l 로 표기

$x \times y = k$ MODEL: LIQUIDITY

▶ 유동성 토큰 발행

▶ 양측 풀에 유동성이 공급될 때 새로운 유동성 토큰이 발행

▶ 시스템 상태는 다음과 같이 전이

▶ $(x, y, l) \rightarrow (x', y', l')$

▶ $x' = (1 + \alpha)x, y' = (1 + \alpha)y, l' = (1 + \alpha)l$

▶ $\alpha = \frac{\Delta x}{x}$

$x \times y = k$ MODEL: LIQUIDITY

- ▶ 유동성 토큰 발행
- ▶ 유동성 공급자
 - ▶ Δx 와 $\Delta y = y' - y$ 를 풀에 묶어두고
 - ▶ 그 대가로 $\Delta l = l' - l$ 만큼의 유동성 토큰을 발행받음
 - ▶ 이때 비율 $x : y : l$ 은 유지
 - ▶ $x : y : l = x' : y' : l'$

$x \times y = k$ MODEL: LIQUIDITY

- ▶ 유동성 토큰 발행
- ▶ k 는 $k = x \times y$ 에서 $k' = x' \times y'$ 으로 증가 ($k < k'$)
- ▶ $\frac{k'}{k} = \left(\frac{l'}{l}\right)^2$

$x \times y = k$ MODEL: LIQUIDITY

▶ 유동성 토큰 소각

▶ 유동성 공급자가 묶어둔 돈을 다시 회수할 때 유동성 토큰이 소각

▶ 시스템 상태는 다음과 같이 전이

▶ $(x, y, l) \rightarrow (x', y', l')$

▶ $x' = (1 - \lambda)x, y' = (1 - \lambda)y, l' = (1 - \lambda)l$

▶ $\lambda = \frac{\Delta l}{l}$

$x \times y = k$ MODEL: LIQUIDITY

- ▶ 유동성 토큰 소각
- ▶ 유동성 공급자
 - ▶ 유동성 토큰을 $\Delta\lambda$ 만큼 소각함으로써
 - ▶ $\Delta x = x' - x$ 와 $\Delta y = y' - y$ 를 출금
 - ▶ 이때 비율 $x : y : l$ 은 유지
 - ▶ $x : y : l = x' : y' : l'$

$x \times y = k$ MODEL: LIQUIDITY

- ▶ 유동성 토큰 소각
- ▶ k 는 $k = x \times y$ 에서 $k' = x' \times y'$ 으로 감소 ($k > k'$)
- ▶ $\frac{k'}{k} = \left(\frac{l'}{l}\right)^2$

IMPLEMENTATION

IMPLEMENTATION

- ▶ 구현상의 이슈
 - ▶ 정수로 다뤄야 함
 - ▶ Floor 또는 Ceil을 이용한 integer rounding
- ▶ 오차 발생
 - ▶ 없던 가치가 mint 되어서는 안 됨
 - ▶ 가치가 큰 폭으로 변동해서는 안 됨

IMPLEMENTATION

▶ _get_input_price

- ▶ 입력한 수량에 맞는 상대 재화를 계산

```
32 def _get_input_price(self, delta_X, X, Y, bool_fee=True):
33     # Validity check
34     if (X == 0) or (Y == 0):
35         raise Exception("invalid X: {} or Y: {}".format(X, Y))
36
37     alpha = float(delta_X / X)
38     gamma = (1. - self.fee) if bool_fee else (1.)
39
40     delta_Y = floor(alpha * gamma / (1. + alpha * gamma) * Y)
41
42     # Validity check
43     if delta_Y >= Y:
44         raise Exception("invalid delta_Y. {} >= {}".format(delta_Y, Y))
45
46     return delta_Y
```

IMPLEMENTATION

▶ _get_output_price

▶ 원하는 상대 재화에 필요한 입력값을 구함

```
48     def _get_output_price(self, delta_Y, Y, X, bool_fee=True):
49         # Validity check
50         if (X == 0) or (Y == 0):
51             raise Exception("invalid X: {} or Y: {}".format(X, Y))
52         if delta_Y >= Y:
53             raise Exception("invalid delta_Y. {} >= {}".format(delta_Y, Y))
54
55         beta = float(delta_Y / Y)
56         gamma = (1. - self.fee) if bool_fee else (1.)
57
58         delta_X = floor(beta / ((1. - beta) * gamma) * X) + 1
59         return delta_X
```

IMPLEMENTATION

► ETH_to_ERC20 (ERC20_to_ETH)

```
61     def ETH_to_ERC20(self, delta_ETH, bool_fee=True, bool_update=True):
62         ETH_prime = self.ETH + delta_ETH
63         delta_ERC20 = self._get_input_price(delta_ETH, self.ETH, self.ERC20, bool_fee=bool_fee)
64
65         if bool_update:
66             ERC20_prime = self.ERC20 - delta_ERC20
67             self._update(ETH_prime, ERC20_prime) # Pool update
68         return delta_ERC20
69
70
71
72
73
74
75
76
77
78
79     def ERC20_to_ETH(self, delta_ERC20, bool_fee=True, bool_update=True):
80         ERC20_prime = self.ERC20 + delta_ERC20
81         delta_ETH = self._get_input_price(delta_ERC20, self.ERC20, self.ETH, bool_fee=bool_fee)
82
83         if bool_update:
84             ETH_prime = self.ETH - delta_ETH
85             self._update(ETH_prime, ERC20_prime) # Pool update
86         return delta_ETH
```

IMPLEMENTATION

► ETH_to_ERC20_exact (ERC20_to_ETH_exact)

```
70     def ETH_to_ERC20_exact(self, delta_ERC20, bool_fee=True, bool_update=True):
71         delta_ETH = self._get_output_price(delta_ERC20, self.ERC20, self.ETH, bool_fee=bool_fee)
72         ETH_prime = self.ETH + delta_ETH
73
74         if bool_update:
75             ERC20_prime = self.ERC20 - delta_ERC20
76             self._update(ETH_prime, ERC20_prime) # Pool update
77         return delta_ETH
78
79     def ERC20_to_ETH_exact(self, delta_ETH, bool_fee=True, bool_update=True):
80         delta_ERC20 = self._get_output_price(delta_ETH, self.ETH, self.ERC20, bool_fee=bool_fee)
81         ERC20_prime = self.ERC20 + delta_ERC20
82
83         if bool_update:
84             ETH_prime = self.ETH - delta_ETH
85             self._update(ETH_prime, ERC20_prime) # Pool update
86         return delta_ERC20
```

IMPLEMENTATION

- ▶ required_ERC20_for_liquidity
 - ▶ 풀 양측 모두 유동성을 공급해야 하므로
 - ▶ ETH를 기준으로
 - ▶ 유동성 공급에 필요한 ERC20 토큰 수량 계산

```
99     def required_ERC20_for_liquidity(self, delta_ETH):
100         alpha = float(delta_ETH / self.ETH)
101
102         ERC20_prime = floor((1. + alpha) * self.ERC20) + 1
103         return ERC20_prime - self.ERC20
```


IMPLEMENTATION

▶ _mint

▶ 유동성 공급

```
137     def _mint(self, delta_ETH, delta_ERC20, bool_update=True): # add_liquidity
138         alpha = float(delta_ETH / self.ETH)
139
140         ETH_prime = self.ETH + delta_ETH
141         ERC20_prime = floor((1. + alpha) * self.ERC20) + 1
142         LT_prime = floor((1. + alpha) * self.LT)
143         delta_LT = LT_prime - self.LT
144
145         if bool_update:
146             self._update(ETH_prime, ERC20_prime, LT_prime) # Pool update
147         return delta_LT
```


IMPLEMENTATION

▶ _burn

▶ 유동성 제거

```
149     def _burn(self, delta_LT, bool_update=True): # remove_liquidity
150         alpha = float(delta_LT / self.LT)
151
152         ETH_prime = ceil((1. - alpha) * self.ETH)
153         ERC20_prime = ceil((1. - alpha) * self.ERC20)
154         LT_prime = self.LT - delta_LT
155         delta_ETH = self.ETH - ETH_prime
156         delta_ERC20 = self.ERC20 - ERC20_prime
157
158         if bool_update:
159             self._update(ETH_prime, ERC20_prime, LT_prime) # burn LT
160         return delta_ETH, delta_ERC20
```

ARBITRAGE

ARBITRAGE

- ▶ 유니스왑에서의 거래
 - ▶ 한 풀의 총량을 증가시키고 반대 풀의 총량을 감소시키는 형태
 - ▶ 풀의 균형이 깨지는 것
- ▶ 많은 양의 편향된 거래로 인해 풀의 균형이 극심하게 깨진다면
 - ▶ 유니스왑을 통한 교환비(ratio)와 외부 거래소를 통한 교환비가 크게 차이나게 됨
 - ▶ 시세 차익을 이용해 이득을 취하는 차익 거래(arbitrage)가 가능

ARBITRAGE

- ▶ 차익 거래(Arbitrage)
- ▶ X 풀과 Y 풀의 균형이 깨졌을 경우
 - ▶ 차익 거래자들은 X 또는 Y 를 교환을 통해 확보하는 것으로 이득을 취함

ARBITRAGE

- ▶ 최고 이득으로 Y 를 교환하기 위해 필요한 X 의 양
 - ▶ X 를 Y 로 교환하는 상황

ARBITRAGE

- ▶ 최고 이득으로 Y 를 교환하기 위해 필요한 X 의 양
- ▶ 이득(Gain)
 - ▶ $Gain_{\Delta x} = \Delta y - \Delta x - fee_x$
 - ▶ X 와 Y 가치의 단위를 맞춰준 후 진행

ARBITRAGE

- ▶ 최고 이득으로 Y 를 교환하기 위해 필요한 X 의 양
- ▶ $Gain_{\Delta x} = \Delta y - \Delta x - fee_x$
 - ▶ Δy 와 Δx 의 관계식을 이용, 변수로 Δx 만을 가지는 식으로 정리
 - ▶ Δx 에 대해 미분한 후, 0이 되는 지점을 찾음

- ▶
$$\Delta x = \frac{\sqrt{x}\sqrt{y}\sqrt{\gamma} - x}{\gamma}$$

ARBITRAGE

▶ 최고 이득으로 X 를 교환하기 위해 필요한 Y 의 양

▶ $Gain_{\Delta y} = \Delta x - \Delta y - fee_y$

▶ Δy 에 대해 미분한 후, 0이 되는 지점을 찾음

▶
$$\Delta y = \frac{\sqrt{x}\sqrt{y}\sqrt{\gamma} - y}{\gamma}$$

EVALUATION

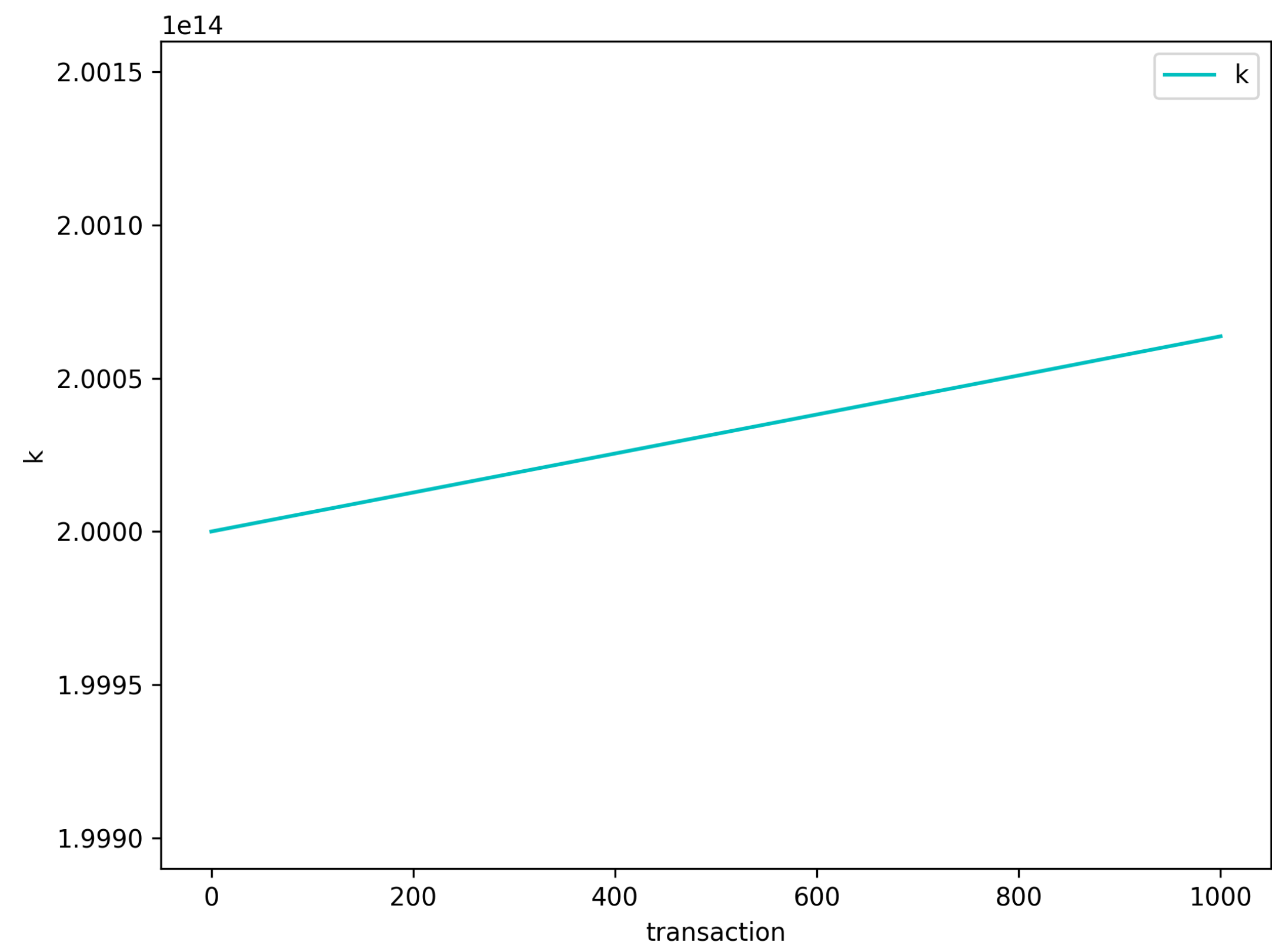
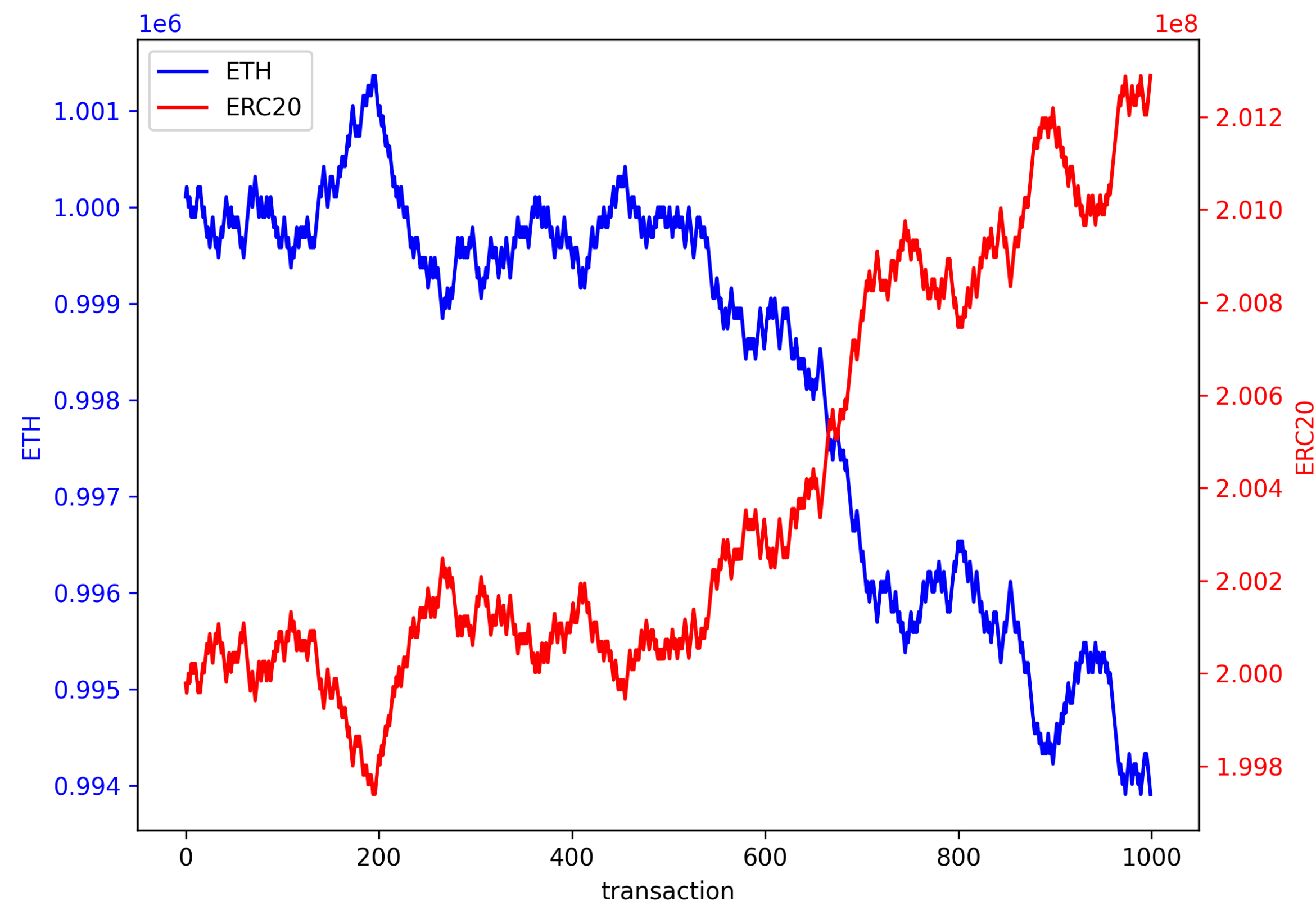
EVALUATION

▶ 실험 환경

- ▶ 초기 X 토큰의 수량은 1,000,000 개, 초기 Y 토큰의 수량은 200,000,000 개
 - ▶ 블록체인 내부 환율은 $x : y = 1 : 200$ 으로 시작됨
- ▶ 초기 유동성 토큰 발행량은 1,000,000
- ▶ 거래 수수료는 0.3%
- ▶ 차익 거래자는 1,000,000,000 Y 의 가치에 해당하는 초기 자금을 가짐
- ▶ 차익 거래자가 활용하는 외부 환율은 $x : y = 1 : 200$ 으로 시작됨

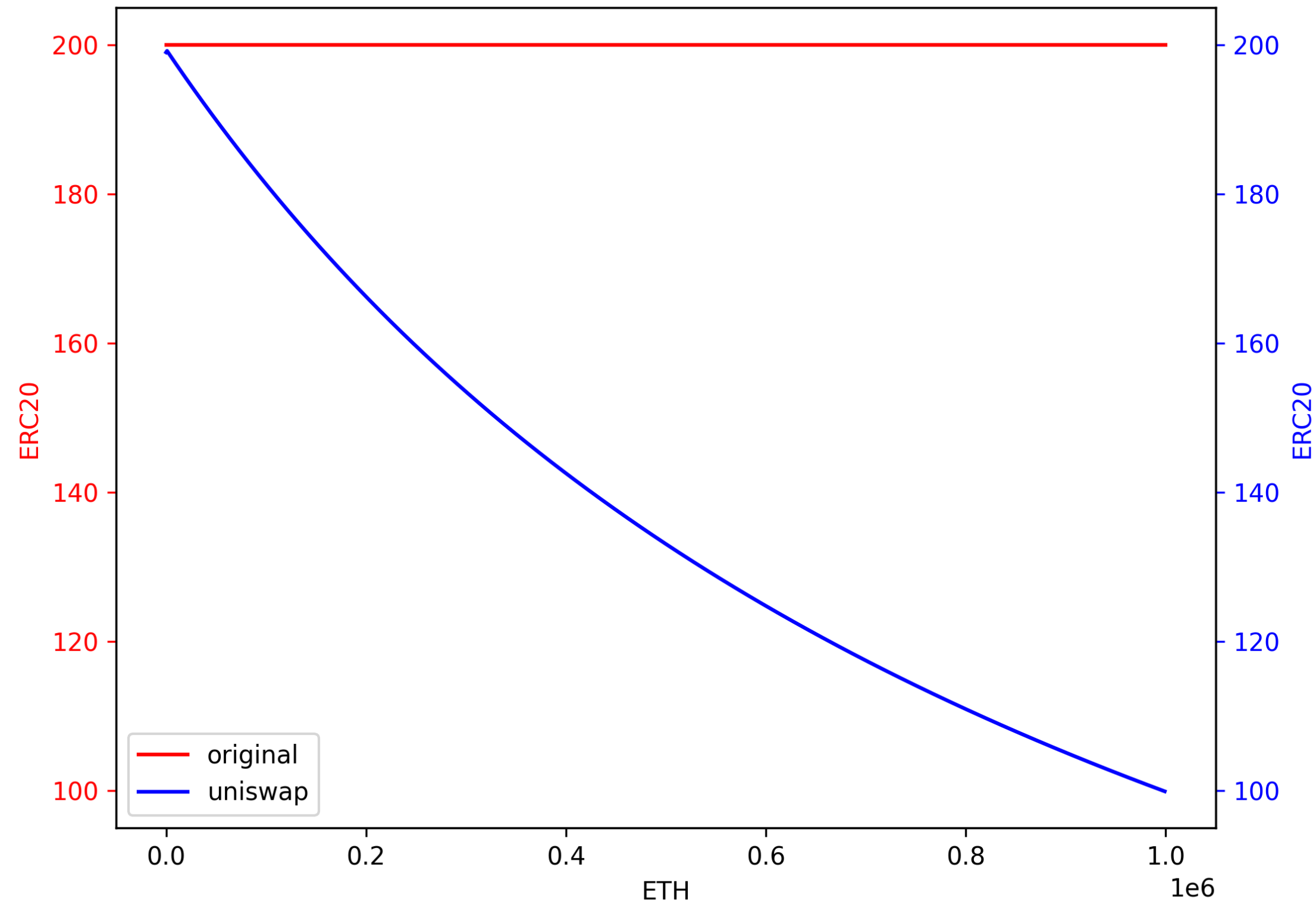
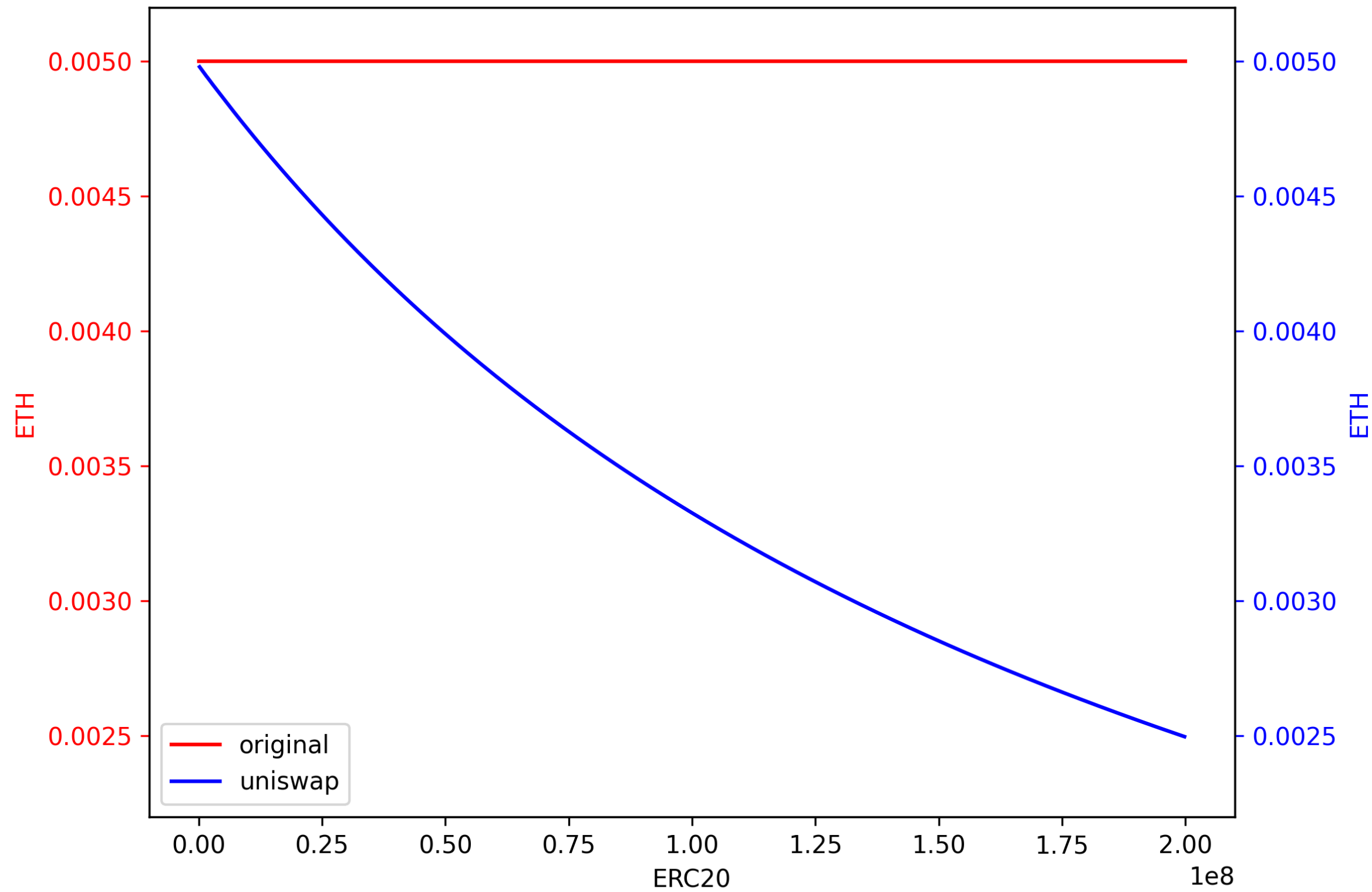
EVALUATION: UNISWAP PROTOCOL

▶ 교환



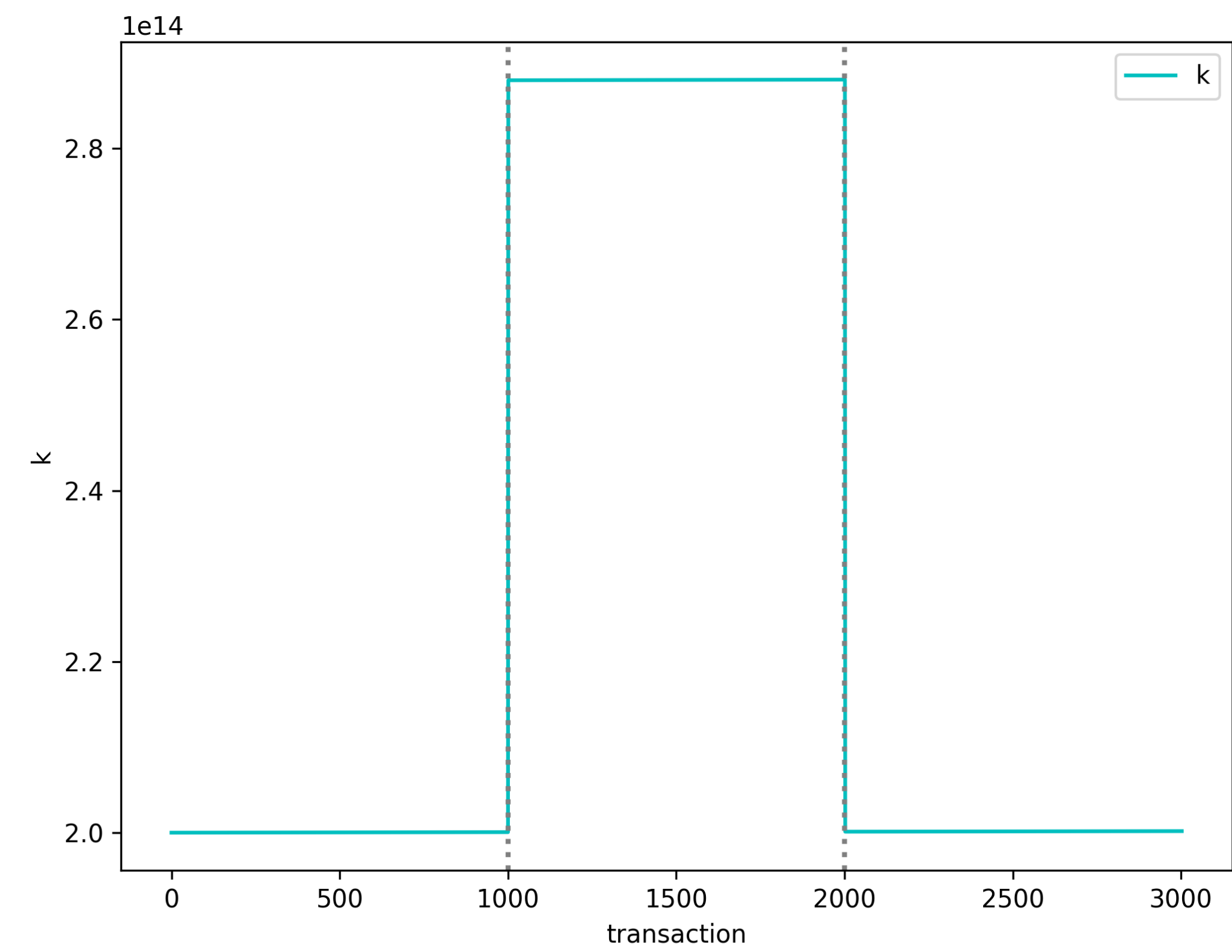
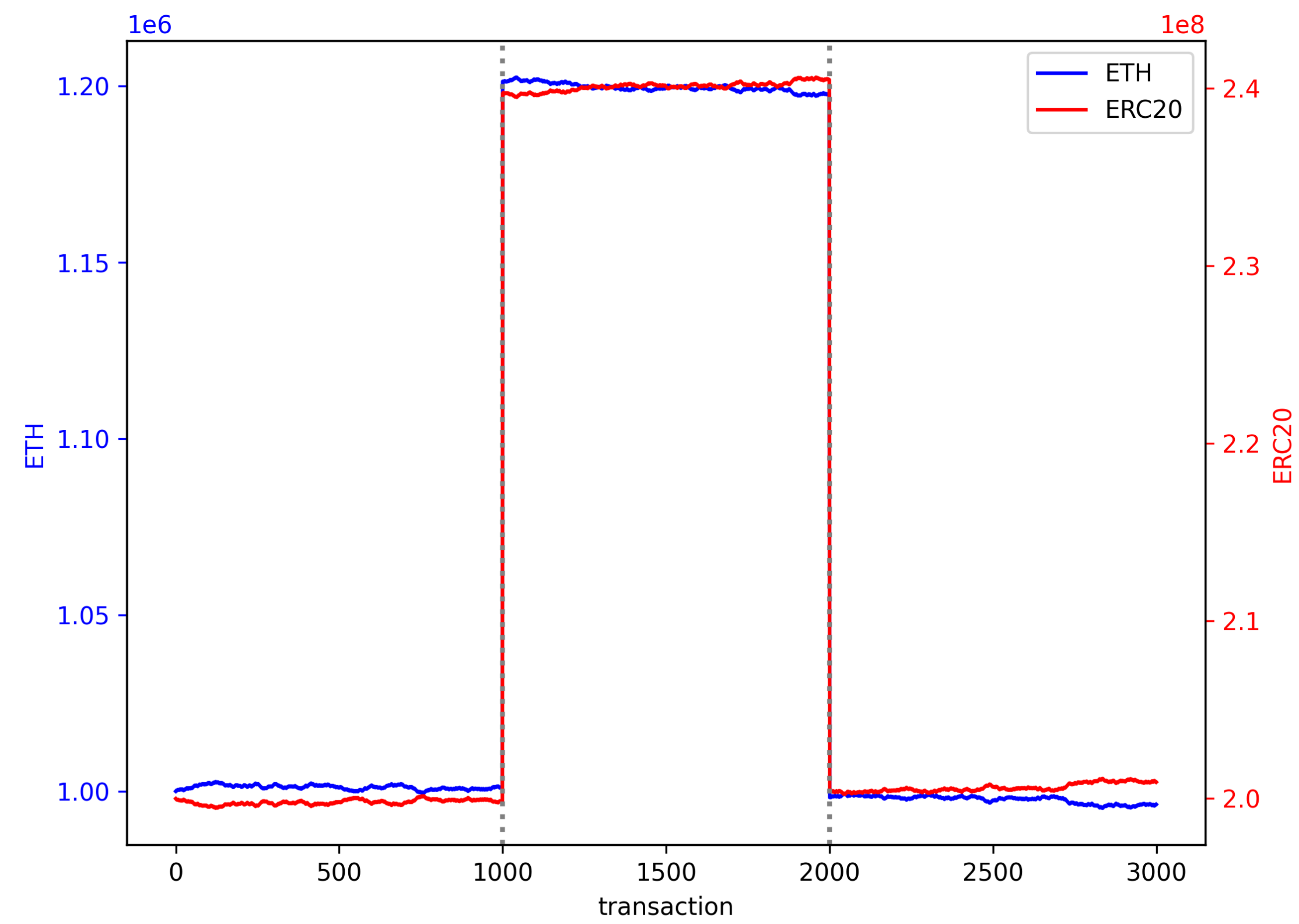
EVALUATION: UNISWAP PROTOCOL

▶ 교환 수량에 따른 상대의 가치



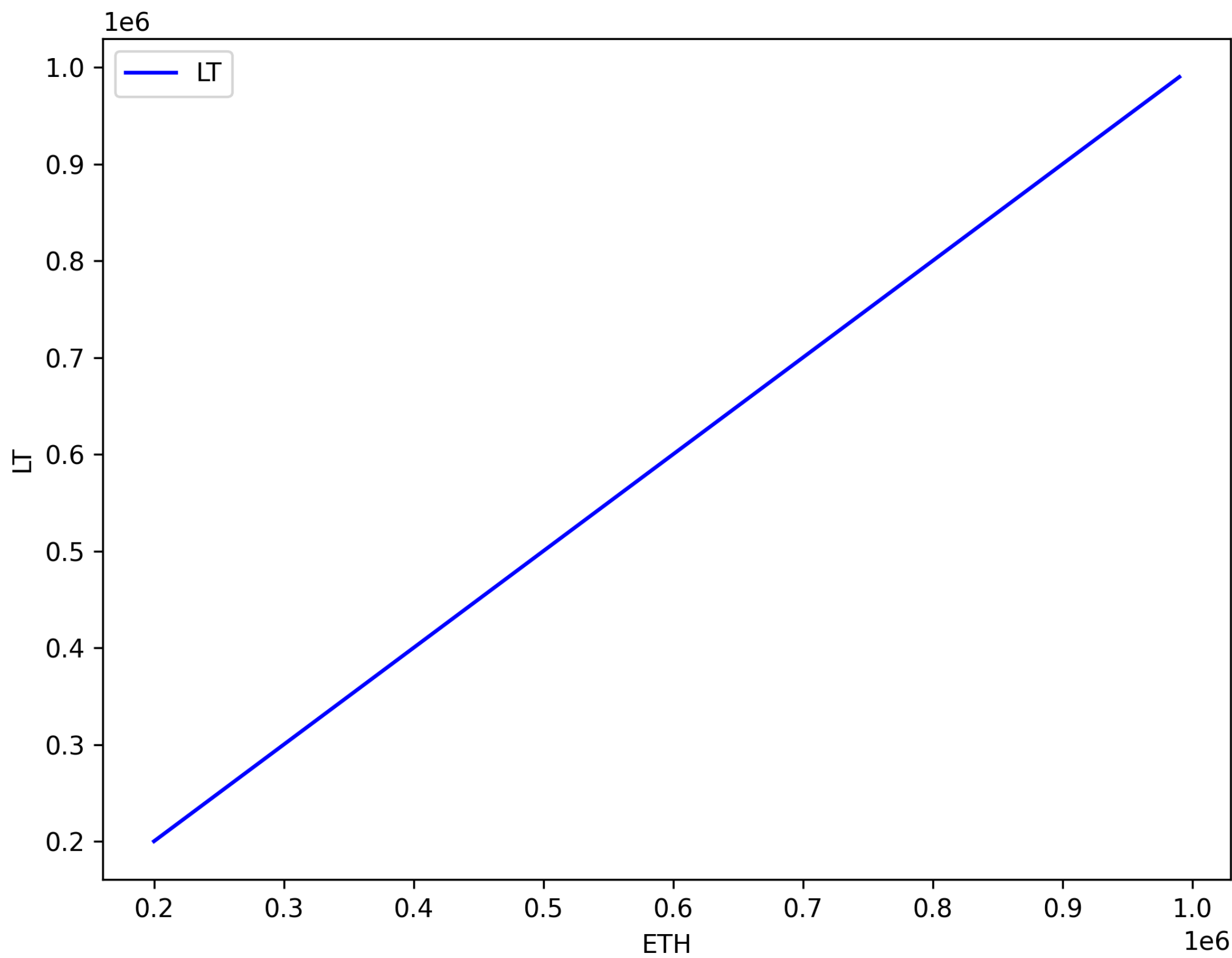
EVALUATION: UNISWAP PROTOCOL

▶ 유동성 공급 및 제거



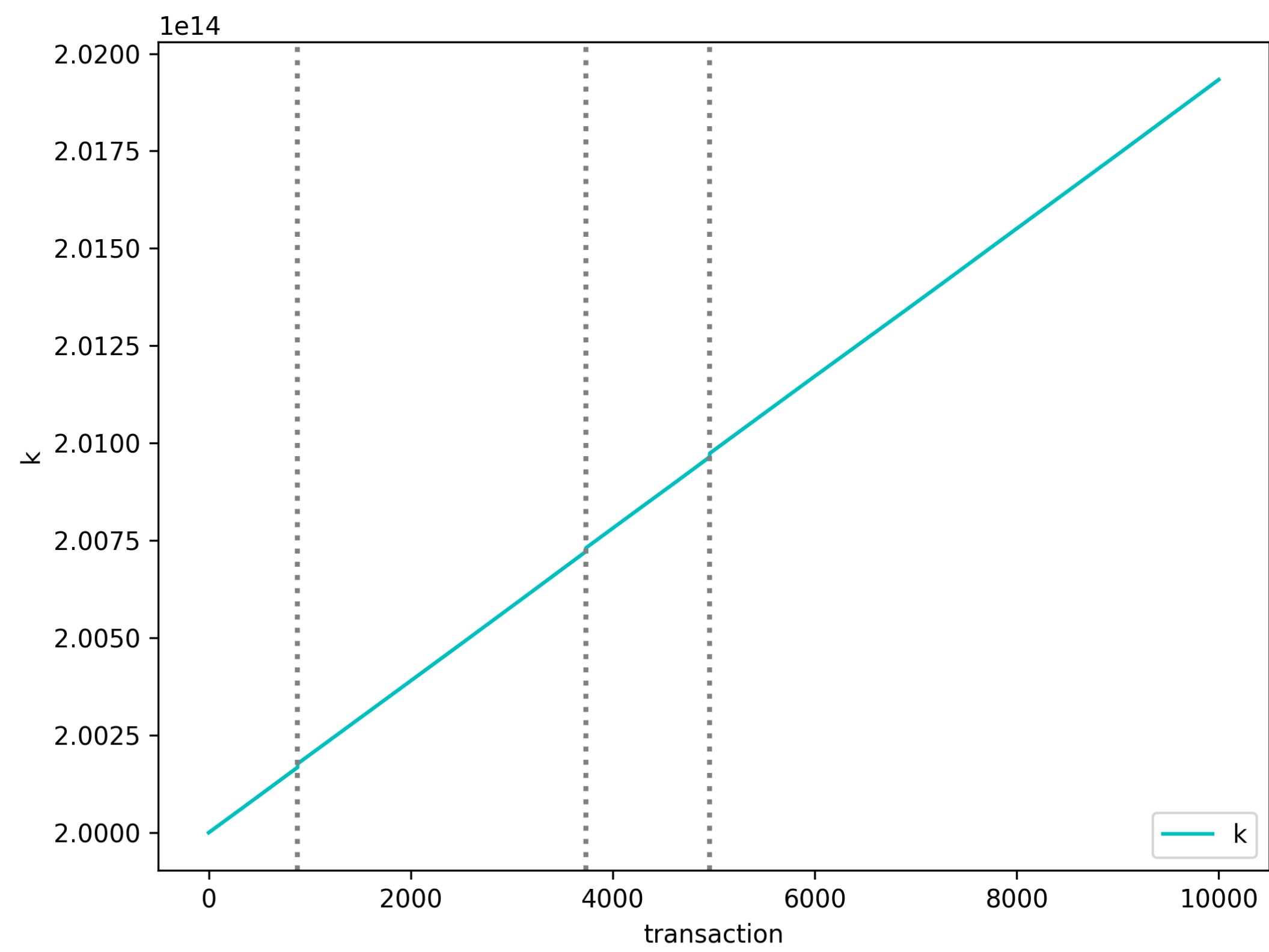
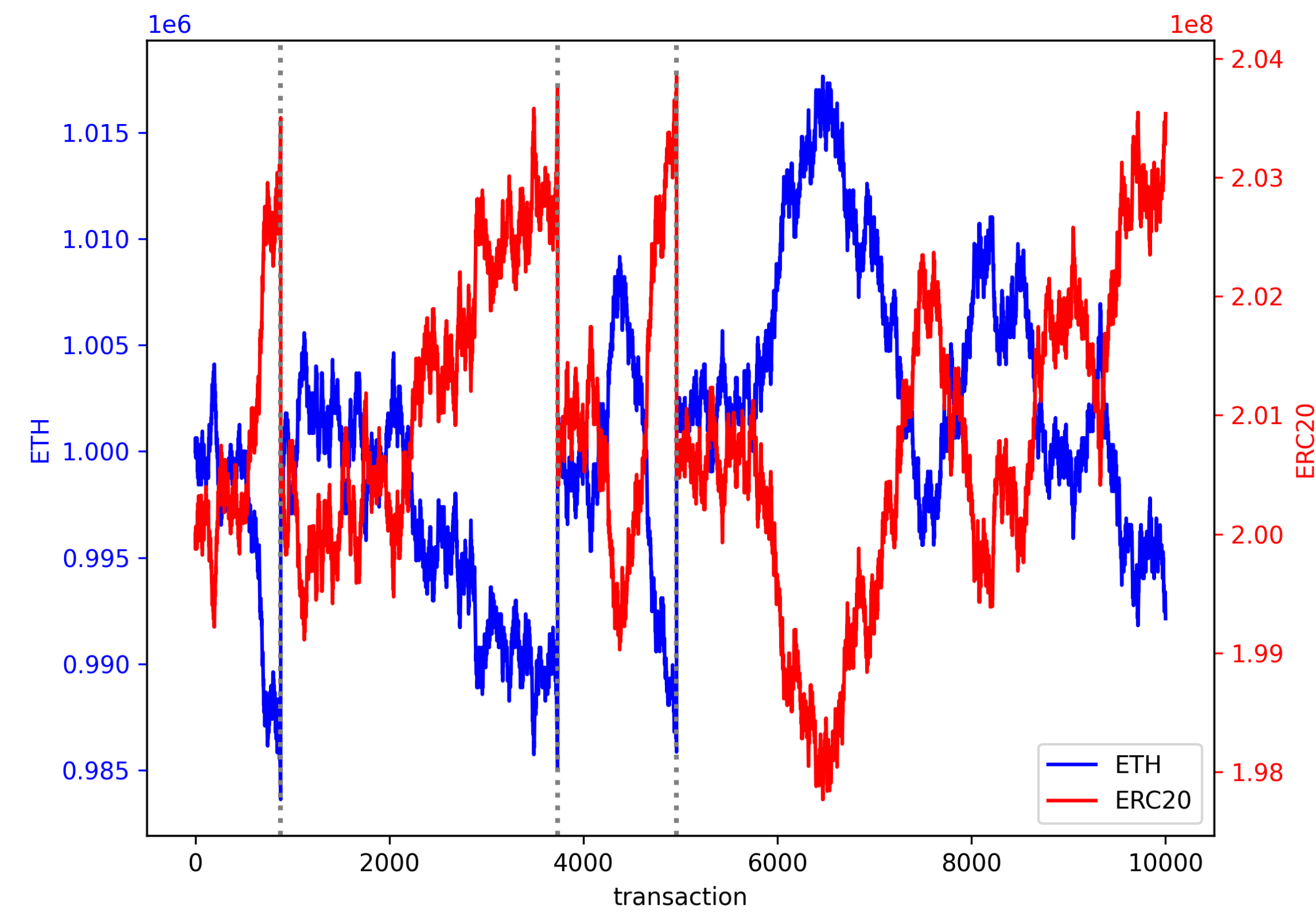
EVALUATION: UNISWAP PROTOCOL

▶ 유동성 토큰 발행량



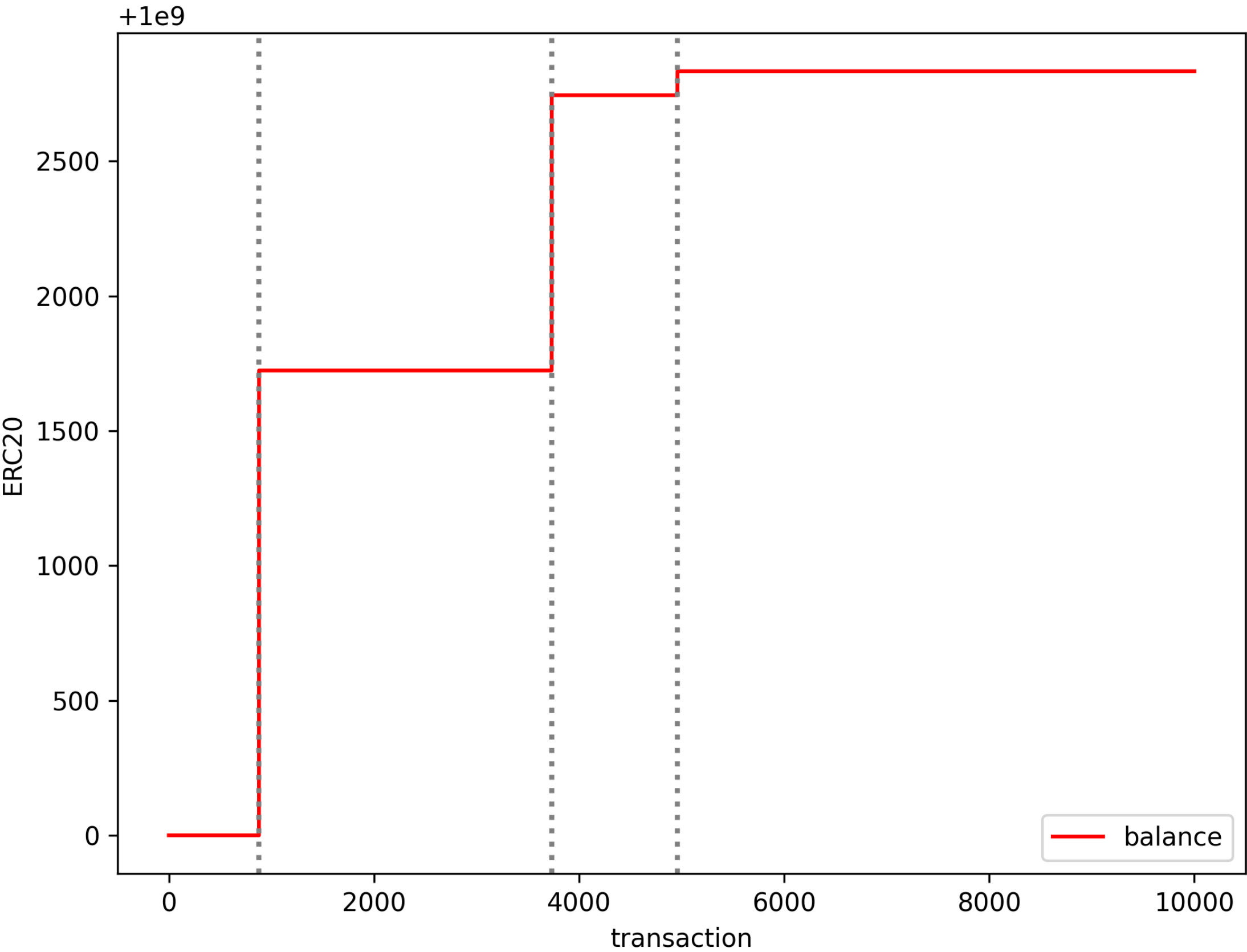
EVALUATION: ARBITRAGE

▶ 차익 거래 발생



EVALUATION: ARBITRAGE

▶ 차익 거래자의 자산



CONCLUSION

OVERVIEW

- ▶ 유니스왑 v1 프로토콜
 - ▶ 기존의 P2P 대출과는 달리 판매자가 필요 없음
 - ▶ 충분한 유동성(풀의 수량)만 공급되어 있다면
 - ▶ 어떤 시점에서든 알고리즘적으로 결정되는 가격에 교환을 수행할 수 있음
- ▶ 충분한 잔액을 갖춘 차익거래자의 존재를 가정하면,
 - ▶ 풀의 균형이 안정적으로 유지됨을 보장할 수 있음

UNISWAP

WITH PYTHON-IMPLEMENTED SIMULATOR