ERCURANCE PROTOCOL

서왕규
wang1@hanyang.ac.kr

1. Introduction

금, 주식, 화폐, 블록체인 위의 암호화폐 등의 다양한 종류의 자산의 가치는 여러 요인에 종속되어 변동한다. 많은 사람들은 자산을 매입하거나, 자산의 가치에 종속되는 상품을 통해 변동성에서 발생하는 차익을 갖는다. 이러한 매입 행위는 반대로, 변동성으로 인한 손실의 위험에도 노출 되게 한다. 변동성이 큰 자산에 투자할수록 위험도가 높아지며 특히 주식이나 가상화폐와 같이 특정 주체나 프로젝트의 성패에 종속되어 가격이 결정되는 경우에는 가치가 없어질 위험이 언제나 존재한다. 문제는 정보의 불균형으로 인해, 많은 투자자는 이 위험도를 과소평가하는 경우가 많고, 투자가 매수자와 매도자의 제로섬 게임의 원리에 바탕하는 것을 감안하면 손실은 고스란히 개인들에게 돌아가는 경우가 대다수이다. 보험이란, 발생할 수 있는 위험에 대한 관리의 목적인 상품임에도, 보험회사만이 보험 상품을 운용할 수 있는 보험자가 될 수 있는 기존의 구조에선 투자의 손실에 대한 보험 상품이 존재할 수 없다. 또한, 기존의 방식은 개인이 보험계약관계자의 역할 중 보험자가 될 수 없기 때문에, 보험 상품의 다양성과 상품성을 낮추는 결과를 초래한다.

스마트컨트렉트를 통해 기금의 운영을 자동화하여 처리할 수 있고 변경 불가능한 계약을 체결할 수 있게 되었다. 이 제안서에서는 주식과 분산어플리케이션(Dapp)에서 발행한 토큰에 대하여 가치가 없어지거나, 일정 이상 낮아질 경우에 발생하는 손실에 대하여 분산된 보증인들이 보증을 해주는 보험을 스마트컨트렉트로 구축하는 방법을 제시하고자 한다.

블록체인과 스마트컨트렉트의 등장으로 현실의 여러 자산을 블록체인 상에 토큰화하거나, 토큰을 사용하는 분산어플리케이션(Dapp)을 구축할 수 있게 되었다. Dapp은 토큰의 판매를 통하여 투자를 받거나 수익을 창출한다. 성공적으로 진행되고 있는 소수의 Dapp을 제외하면, 대부분은 실패하거나 발전이 없는 실태 속에서 이들이 발행한 토큰은 판매할 수 있는 거래소조차 없이 0의 가치에 수렴하게 되고 있다. 문제는 개발자가 아닌 일반 투자자들은 이런 프로젝트를 선별하는 것이 매우 어렵기 때문에, 이로 인한 피해는 토큰의 구매를 통해 투자한 투자자들에게 돌아가게 되고, 일반 투자자들에게 피해가 누적될수록 블록체인 기술 자체의 평가는 악화되어 블록체인의 생태계에도 악영향을 끼친다. 이에 대한 해결책으로 ERC20 토큰에 대한 가치를 보증해주는 보험을 분산된 보증자들이 운용할 수 있도록 하는 스마트컨트렉트를 구축하는 방법에 대하여 서술한다.

또한, 이 제안서의 스마트 컨트렉트를 활용하여 기업의 가치에 의해 가격이 결정되는 블록체인 외부의 자산인 주식이나 그 외의 자산에 대한 분산된 보증 보험을 구축하는 방법을 서술한다.

이 제안서에서 제시하는 방법을 통하여 개인의 보험 상품 운용을 하는 보험자가 될 수 있도록 하는 새로운 형태의 금융 상품을 만든다. 아래의 내용은 Loop문을 포함한 Turing Complete한 스마트컨트렉트를 지원하는 블록체인을 대상으로 작성되며, 구체적으로 Solidity 언어를 통해 설명된다. 스마트컨트렉트의 구현의 서술에서 특정 부분은 코드 단계까지 설명한다.

2. Process

아래는 스마트 컨트렉트가 작동하는 과정에 대한 추상화된 설명이다. "2. Process" 이후의 항목에서 이러한 추상화된 과정이 코드 단계에서 어떻게 구현되는지와 각 단계의 중요한 규칙에 대하여 서술된다.

블록체인 플랫폼의 재화(예를 들어, 클레이, 이더리움)를 A, Dapp의 토큰을 B라고 하자. 이 문서에서는 보험의 보험계약관계자에게 스마트컨트렉트에서의 역할에 따른 명칭을 부여하기 때문에, 보험계약관계자 중 보험자를 보증자, 피보험자를 보험가입자라 부른다. 보험수익자는 피보험자와 동일하게 설정한다. ① B에 대한 보험의 운용을 하길 원하는 유저들은 A를 보증금으로 지출한다. ② 해당 보험을 가입하고 싶은 이는 B를 스마트컨트렉트에 일정 기간마다 제출하는 것으로 가입한다. ③ 수거한 보험료를 보증금의 비율에 따라 나누어 가진다. ④ B의 가격이 스마트컨트렉트에서 설정된 금액보다 낮아진 상태로 일정기간 유지되면, 보증금을 가입자들에게 가입 기간에 비례하여 배분한다. ⑤ B의 프로젝트가 성공하여 B의 가격이 스마트컨트렉트에서 설정된 금액보다 높은 상태로 일정기간 유지되면 보증금은 보증자들에게 다시 회수된다.

위 과정을 통하여 보증자들은 Dapp의 토큰을 보증금에 비례하여 대가로 받고, 보험 가입자들은 토큰이 계약에 명시된 금액 아래로 가치가 떨어질 경우 보증금을 받을 수 있으므로, 투자의 위험을 줄일 수 있다.

추가적으로, 해당 상품의 운용에서 보증자의 보험에 대한 지분과 보험가입자의 보험 가입의 기간을 토큰화하거나, 다수의 보험 상품을 묶어 운용함으로 상품성을 강화시키고, 보증자의 위험도를 낮출 수 있을 것이다.

3. Oracle Problem

이 스마트컨트렉트를 구현함에 있어서 중요한 문제는 토큰의 가격에 대한 정보를 어떻게 블록체인 상에 위변조없이 가져올 수 있는가에 대한 것이다. 블록체인 외부의 정보가 블록체인에 잘못되게 기록되는 경우를 Oracle Problem 이라한다. 이는 블록체인이 중앙기관이 없기 때문에 발생하는 문제이다. Oracle Problem을 해결하기 위해, 중앙값의 사용, 미들웨어의 사용, 투표와 같은 해결책이 존재하는데, 이러한 방법들은 수수료가 과도하게 중첩될 가능성이 있거나, 악의적인 공격을 받을 수 있다. 이런 문제를 막기 위해, 블록체인 외부의 정보를 가져오는 것이 아닌, Uniswap과 같은 유동성 풀과 연결을 통해 Oracle Problem없이 토큰의 가치를 확인할 수 있다. 스마트컨트렉트를 유동성 풀과 연결하여, 해당 유동성 풀의 Reserve 비율을 토큰의 가치로 설정하는 것으로 블록체인 외부에서 정보를 가져오지 않고 분산된 보증 보험을 운용할 수 있다.

4. Malicious user

유동성 풀과 분산된 보증 보험을 연결했을 때, 보증금을 받기 위해 또는 보험의 자금을 강제로 회수하기 위해 유동성 풀에 단기적인 대량의 교환을 하는 공격이 이뤄질 수 있다. 이는 특히 유동성 공급자의 수가 적고, 유동성이 작은 풀인 경우 공격 받기 쉽다.

◎ 강제 보험 실행 공격
(1) 보험을 다수의 계정을 통해 가입한다. (2) 보험과 연결된 유동성 풀에 단기적으로 대량의 토큰을 판매하여 가치를 낮추어 보험을 실행시킨 뒤, 다시 토큰을 대량으로 구입한다.
◎ 강제 보증금 회수 공격
(1) 보증금을 넣고 보험료를 할당받아 수익을 창출한다. (2) 프로젝트의 성패가 정해지지 않았음에도 연결된 유동성 풀에 대량의 토큰을 구입하는 것으로 토큰의 가치를 올려 보증 참여자는 보증금을 회수한다.

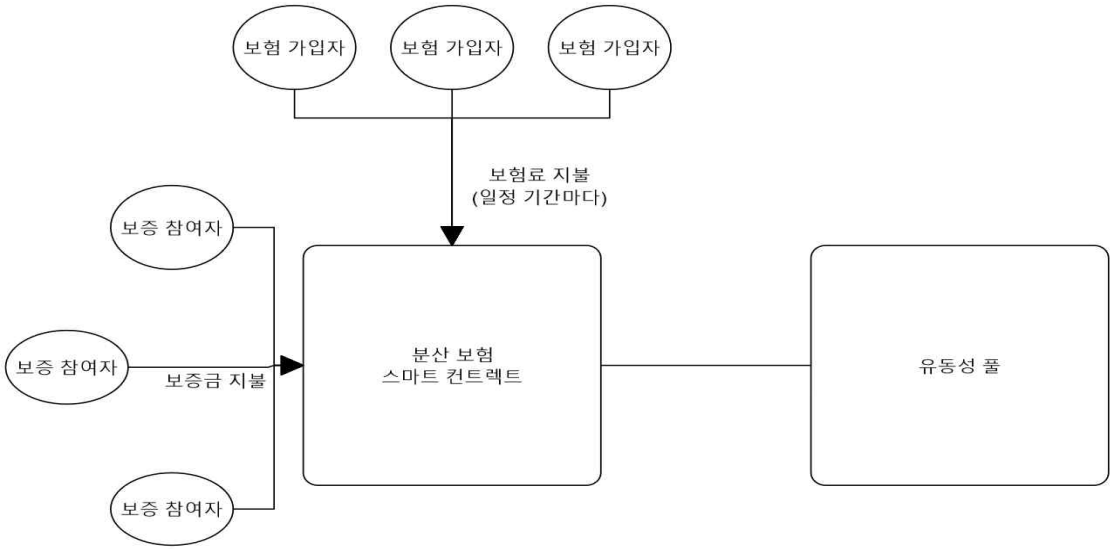
위의 두 공격은 유동성 풀에 유동성을 공급해두고 있는 상태에서 한 번에 유동성을 회수하고 토큰을 구매하거나 판매하는 행위를 통해 더욱 강화될 수 있다.

이를 방지하기 위해, 보증 참여자는 유동성이 높고, 유동성 공급자의 수가 많은 보험 상품에 참여하도록 주의를 기울여야한다. 보험 스마트컨트렉트 내에서는 이러한 공격을 방지하기 위해, 가치의 하락이나 상승에 대한 판단을 시간차이를 두고 유동성 풀을 여러 번 탐색하여 판단하도록 해야 한다. 아래는 제안서에 첨부된 코드에서 설정한 판단횟수를 통한 예시이다.

① [X분] 1회 차 판정 : 가치가 보험이 실행될 기준(Threshold)이상/이하로 판정 ② [(①+15)분, (①+30)분] 2회 차 판정 : 가치가 보험이 실행될 기준(Threshold)이상/이하로 판정 ③ [(이전 회 차)분 + 1시간, (이전 회 차)분 + 2시간] 3,4,5회 차 판정 : 가치가 보험이 실행될 기준(Threshold)이상/이하로 판정 ④ [③분 + 5시간, ③분 + 10시간] 6회 차 판정 : 가치가 보험이 실행될 기준(Threshold)이상/이하로 판정 ⑤ [④분 + 1일, ④분 + 2일] 7회 차 판정 : 가치가 보험이 실행될 기준(Threshold)이상/이하로 판정

위의 7번의 판정을 통과한 뒤 보험의 가입자들에게 보증금을 주도록 구현하여, Malicious user에 대한 공격을 방어할 수 있다. 또한, 유동성이 낮고, 공급자가 적은 풀은 일반적으로 그만큼 가입자와 보증자 수가 적어 수익을 기대하는 공격이 의미가 없을 것이다. 그러나 여전히 낮은 유동성 공급자의 수와 유동성을 지닌 풀은 공격을 받을 여지가 있으므로 가입과 보증에 주의할 필요가 있다.

4. Smart Contract Detailed



(1) B에 대한 분산 보험 스마트컨트렉트를 발행한다. 스마트컨트렉트의 생성자에는 다음과 같은 요소가 포함되어야 한다.

```
constructor(IUniswapV2Pair _pair, uint256 _executeTrigger, uint256 _endTrigger, uint
_timeInterval, uint128 _feeRate) public {
        pair = _pair;
        executeTrigger = _executeTrigger;
        endTrigger = _endTrigger;
        interval = _timeInterval;
        feeRate = _feeRate;
        checkTime = now;
}
```

연결된 유동성 풀 주소
보증금이 보험 가입자에게 지급되는 트리거 금액
보증금이 보증 참여자에게 돌아오기 위한 종결 금액
가입자들의 보험금 납입 주기
보험료의 비율 (Fee Rate)

(2) B에 대한 보험의 운용을 하길 원하는 유저들은 A를 스마트 컨트렉트에 보증금으로 지출한다.

- 이 제안서의 코드에서는 스마트 컨트렉트에 지출하는 0.1 Ethereum당 보증 보험의 지분을 1 갖도록 한다. 이 과정에서 보증 보험의 지분을 거래할 수 있도록 mapping 자료구조의 기록이 아닌 ERC20 토큰을 상속받아 구현할 수도 있다. 이 제안서에서는 mapping 자료구조를 통한 기록에 중점을 두어 서술한다.

```
function guarantee() payable public onlyRunning onlyNotTakeDay {
        require(msg.value >= 0.1 ether, "INSURANCE : MORE THAN 0.1 ETHER");
        uint64 mySupport = uint64(msg.value / 0.1 ether);
        support[msg.sender] += mySupport;
        totalSupport += mySupport;
}
```

(3) 해당 보험을 가입하고 싶은 이는 토큰 B를 보험료로 스마트컨트렉트에 일정 기간마다 제출하는 것으로 가입한다. 보험료는 토큰의 가격에 반비례하게 설정함으로 위험부담이 큰 시점에선 높은 보험료를 받을 수 있도록 한다.

보험료 $= (B\,reserve / A\,reserve) * Fee\,Rate$

```
function payFee() public onlyRunning onlyNotTakeDay {
        require(
            subscriber[msg.sender].duration == 0 || subscriber[msg.sender].last - block.timestamp < 2 days || subscriber[msg.sender].last < block.timestamp,
            "INSURANCE : DENY"
        );
        uint256 fee = feeRate * (1/getpairRate());
        IERC20(pair.token1()).transferFrom(msg.sender, address(this), fee);
        feePool += fee;
        if(subscriber[msg.sender].duration == 0) {
            subscriber[msg.sender] = InsuredInfo({
                duration:1,
                last:block.timestamp + interval
            });
            totalDuration++;
        } else {
            subscriber[msg.sender].last = subscriber[msg.sender].last + interval;
            subscriber[msg.sender].duration++;
            totalDuration++;
        }
}
```

(4) 수거한 보험료를 보증금의 비율에 따라 나누어 가진다.

```
function giveFeeToGuarantor() public onlyRunning onlyTakeDay {
        require(check[checkTime][msg.sender] == false, "INSURANCE : ALREADY TAKE");
        if(reward[checkTime] == 0){
            reward[checkTime] = feePool / totalSupport;
        }
        uint256 mine = reward[checkTime] * support[msg.sender];
        check[checkTime][msg.sender] = true;
```

```
        IERC20(pair.token1()).transfer(msg.sender, mine);
        feePool = feePool - mine;
}
```

(5) B의 가격이 매우 낮아지고, 낮아진 가격이 일정 기간 유지될 경우 프로젝트가 실패한 것으로 판단하여, 보증금을 가입자들에게 가입 기간에 비례하여 지불한다. 이 때, 보험 가입자의 가입 기간이란 해당 보험에 불연속적으로 가입한 기간의 총합을 의미한다. 추가적으로, 이 가입기간을 ERC20토큰으로 만들어 거래가 가능하게 할 수 있다.

아래의 코드는 설명을 위해 풀어쓴 코드로, 수수료의 최적화에 대한 처리가가 되어있지 않다. 수수료를 최적화하기 위해서, storage data를 memory data로 복사하여 사용해야하고, 나열된 조건문의 순서를 빈도수가 높은 순서로 바꾸고 수정할 필요가 있다.

```
function reportBankrupt() public onlyRunning {
        uint time = block.timestamp;
        require(getpairRate() < executeTrigger, "INSURANCE : DENY");
        require(bankruptReport + timeInterval[bankruptReportCount] < time, "INSURANCE
: NOT NOW");
        if (bankruptReportCount == 0) {
            bankruptReport = time;
            bankruptReportCount++;
        } else if (bankruptReport + timeInterval[bankruptReportCount] < time &&
bankruptReport + timeInterval[bankruptReportCount]*2 > time) {
            bankruptReport = time;
            bankruptReportCount++;
        } else if (bankruptReport + timeInterval[bankruptReportCount]*2 < time) {
            bankruptReport = time;
            bankruptReportCount = 0;
        }

        if (bankruptReportCount >= 7) {
            state = ContractState.EXECUTE;
        }
}
```

(6) B의 프로젝트가 성공하여, 더 이상 보험의 신청자가 없거나 매우 적을 경우, 보증금은 보증자들에게 다시 회수된다.

```
function reportSuccess() public onlyRunning {
        require(getpairRate() > endTrigger, "INSURANCE : DENY");
        uint time = block.timestamp;
        require(successReport + timeInterval[successReportCount] < time, "INSURANCE :
NOT NOW");
        if (successReportCount == 0) {
            successReport = time;
            successReportCount++;
        } else if (successReport + timeInterval[successReportCount] < time &&
```

```
successReport + timeInterval[successReportCount]*2 > time) {
            successReport = time;
            successReportCount++;
        } else if (successReport + timeInterval[successReportCount]*2 < time) {
            successReport = time;
            successReportCount = 0;
        }

        if (successReportCount >= 7) {
            state = ContractState.SUCCESS;
        }
    }
```

5. Stock

이 제안서의 스마트컨트렉트를 활용하여 주식이나 자산의 가치에 대한 분산된 보증 보험을 제작할 수 있다.

이는 특정 A기관이 1 Token을 1주로 보장해주는 주식 기준의 Stable Token을 발행하는 것으로 실현이 가능하다. 이러한 Stable Token의 또 다른 장점은 기존의 1주 단위라는 주식의 최소 단위를 없애 더 작은 단위로 거래할 수 있다는 장점을 가지고 있다. 이 과정에서 A기관이 1주를 보증하는 양식은 다음 두 방법으로 나뉜다.

(1) 1 Token을 주식 1주와 교환을 담보한다.

(2) 1 Token을 주식 1주의 가격과 동일하게 담보한다.

분산된 보증 보험의 구현을 설명하자면,

① A기관은 테슬라의 1000주를 보유하고 B라는 토큰을 발행하여, 1토큰 당 1주의 가치를 담보한다.

② B토큰에 대한 유동성 풀을 생성한다.

③ 해당 유동성 풀과 연결된 분산 보증 보험을 발행하여 가치에 대한 보증을 한다.

이 일련의 과정을 통해 주식의 가격이 보험에서 설정한 기준 밑으로 떨어져 유지되는 경우에 발생하는 손실에 대한 보증을 하는 보험을 분산된 보증자들을 통해 운용할 수 있다.

6. Conclusion

기존의 보험은 규모와 신뢰도를 고려하였을 때, 중앙화되어 운영될 수 밖에 없었다. 이 제안서에서는 스마트컨트렉트를 통해 보험기금의 운용이 투명하게 공개될 수 있고, 분산된 다수가 운용하는 보험상품을 만든다. 이 새로운 형태의 금융상품을 통하여, 분산된 보증자들은 Dapp이나 자산을 간접적으로 투자할 수 있고, 보험가입자는 투자에 대한 위험도를 낮출 수 있을 것이다.

<별첨 : Smart Contract 예시코드>

```
pragma solidity ^0.6.0;

import '@uniswap/v2-periphery/contracts/libraries/UniswapV2Library.sol';
import '@uniswap/v2-core/contracts/interfaces/IUniswapV2Pair.sol';
import '@openzeppelin/contracts/token/ERC20/IERC20.sol';



/** @author Wanggyu, suh
    @dev
    이 코드는 예시 코드로 직접적으로 사용하지 않아야 합니다.
    설명과 가독성을 위해 풀어서 작성된 코드이며, Overflow의 처리와 같은 예외처리를 배제한 코
    드입니다.
    실제 구현에서는 Uniswap이 가지고 있는 코드 구조와 같이 Factory와 Router로 구분하여 작
    성해야 합니다.
*/

contract Insurance {

    IUniswapV2Pair pair;
    enum ContractState {RUN, EXECUTE, SUCCESS}
    struct InsuredInfo {
        uint64 duration;
        uint last;
    }

    ContractState state;
    uint[7] timeInterval = [0, 15 minutes, 1 hours, 1 hours, 1 hours, 5 hours, 1 days];
    mapping(address => uint64) support;
    mapping(address => InsuredInfo) subscriber;
    mapping(uint => mapping(address => bool)) check;
    mapping(uint => uint256) reward;
    mapping(address => bool) checkExecution;
    uint256 public rewardOfExecution;
    mapping(address => bool) checkSuccess;
    uint256 public totalSupport;
    uint256 public totalDuration;
    uint256 public executeTrigger;
    uint256 public endTrigger;
    uint256 public feePool;
    uint checkTime;
    uint interval;
    uint128 feeRate;
    uint bankruptReport;
    uint64 bankruptReportCount;
```

```
    uint successReport;
    uint64 successReportCount;

    modifier onlyRunning() {
        require(state == ContractState.RUN, "INSURANCE : FINISHED INSURANCE");
        _;
    }

    modifier onlyTakeDay() {
        /* Need to chang safeMath */
        require(now - checkTime >= interval && now - checkTime < interval + 3 days,
"INSURANCE : Can't take now");
        _;
    }

    modifier onlyNotTakeDay() {
        require(now - checkTime < interval, "INSURANCE : Can't pay Now");
        _;
    }

    constructor(IUniswapV2Pair _pair, uint256 _executeTrigger, uint256 _endTrigger, uint
_timeInterval, uint128 _feeRate) public {
        pair = _pair;
        executeTrigger = _executeTrigger;
        endTrigger = _endTrigger;
        interval = _timeInterval;
        feeRate = _feeRate;
        checkTime = now;
    }

    /** @dev Erase when you want to deploy this contract. This function is only for dev
*/
    function set(IUniswapV2Pair _pair, uint256 _executeTrigger, uint256 _endTrigger, uint
_timeInterval, uint128 _feeRate) public {
        pair = _pair;
        executeTrigger = _executeTrigger;
        endTrigger = _endTrigger;
        interval = _timeInterval;
        feeRate = _feeRate;
        checkTime = now;
    }

    function setTime() public {
        while(now - checkTime > interval + 3 days){
            checkTime = checkTime + interval;
```

```
        }
    }

    function giveFeeToGuarantor() public onlyRunning onlyTakeDay {
        require(check[checkTime][msg.sender] == false, "INSURANCE : ALREADY TAKE");
        if(reward[checkTime] == 0){
            reward[checkTime] = feePool / totalSupport;
        }
        uint256 mine = reward[checkTime] * support[msg.sender];
        check[checkTime][msg.sender] = true;
        IERC20(pair.token1()).transfer(msg.sender, mine);
        feePool = feePool - mine;
    }

    function executeInsurance() public{
        require(state == ContractState.EXECUTE, "INSURANCE : No");
        require(checkExecution[msg.sender] == false, "INSURANCE : Already take");
        if(rewardOfExecution == 0){
            rewardOfExecution = address(this).balance / totalDuration;
        }
        checkExecution[msg.sender] == true;
        (msg.sender).transfer( subscriber[msg.sender].duration * rewardOfExecution);
    }

    function successInsurance() public {
        require(state == ContractState.SUCCESS, "INSURANCE : No");
        require(checkSuccess[msg.sender] == false, "INSURANCE : Already take");
        checkSuccess[msg.sender] == true;
        (msg.sender).transfer(support[msg.sender] * 0.1 ether);
    }


    function guarantee() payable public onlyRunning onlyNotTakeDay {
        /* @dev Need to control overflow */
        require(msg.value >= 0.1 ether, "INSURANCE : MORE THAN 0.1 ETHER");
        uint64 mySupport = uint64(msg.value / 0.1 ether);
        support[msg.sender] += mySupport;
        totalSupport += mySupport;
    }

    function payFee() public onlyRunning onlyNotTakeDay {
        require(
            subscriber[msg.sender].duration == 0 || subscriber[msg.sender].last -
block.timestamp < 2 days || subscriber[msg.sender].last < block.timestamp,
            "INSURANCE : DENY"
```

```
        );
        uint256 fee = feeRate * (1/getpairRate());
        IERC20(pair.token1()).transferFrom(msg.sender, address(this), fee);
        feePool += fee;
        if(subscriber[msg.sender].duration == 0) {
            subscriber[msg.sender] = InsuredInfo({
                duration:1,
                last:block.timestamp + interval
            });
            totalDuration++;
        } else {
            subscriber[msg.sender].last = subscriber[msg.sender].last + interval;
            subscriber[msg.sender].duration++;
            totalDuration++;
        }
    }


    function reportBankrupt() public onlyRunning {
        uint time = block.timestamp;
        require(getpairRate() < executeTrigger, "INSURANCE : DENY");
        require(bankruptReport + timeInterval[bankruptReportCount] < time, "INSURANCE
: NOT NOW");
        if (bankruptReportCount == 0) {
            bankruptReport = time;
            bankruptReportCount++;
        } else if (bankruptReport + timeInterval[bankruptReportCount] < time &&
bankruptReport + timeInterval[bankruptReportCount]*2 > time) {
            bankruptReport = time;
            bankruptReportCount++;
        } else if (bankruptReport + timeInterval[bankruptReportCount]*2 < time) {
            bankruptReport = time;
            bankruptReportCount = 0;
        }

        if (bankruptReportCount >= 7) {
            state = ContractState.EXECUTE;
        }
    }

    function reportSuccess() public onlyRunning {
        require(getpairRate() > endTrigger, "INSURANCE : DENY");
        uint time = block.timestamp;
        require(successReport + timeInterval[successReportCount] < time, "INSURANCE :
NOT NOW");
```

```
        if (successReportCount == 0) {
            successReport = time;
            successReportCount++;
        } else if (successReport + timeInterval[successReportCount] < time &&
successReport + timeInterval[successReportCount]*2 > time) {
            successReport = time;
            successReportCount++;
        } else if (successReport + timeInterval[successReportCount]*2 < time) {
            successReport = time;
            successReportCount = 0;
        }

        if (successReportCount >= 7) {
            state = ContractState.SUCCESS;
        }
    }

    /** @dev
        It is a pseudo function. In uniswap factory, token is sorted. so make this
function to check WETH address and divided WETH reserve / Token reserve
해당 함수의 구현은 연결된 유동성 풀의 API에 따라 다르며, 실제 구현에서는 Uniswap v2의 누적
가격을 이용하여, 악의적인 공격을 방지할 수 있다.
     */
    function getpairRate() internal returns (uint) {
        (uint reserves0, uint reserves1,) = pair.getReserves();
        return reserves0 / reserves1;
    }
}
```