Online accompaniement for "Collaborative and parametric insurance on the Ethereum blockchain"

Pierre-Olivier Goffard [*1] and Stéphane Loisel[†2]

[1] Université de Strasbourg, Institut de Recherche Mathématique Avancée, Strasbourg, France

[2] CNAM, Paris, France

December 2, 2024

We are going to review the solidity code of the smart contract `ParametricInsurance` discussed iinn the paper entitled "Collaborative and parametric insurance on the Ethereum blockchain". We start by indicating the version of the compiler and the name of the smart contract. Solidity is a high-level programming language

```
pragma solidity 0.8.19;

contract ParametricInsurance {
---
}
```

Code Fragment 1: Declaring a compiler and define a smart contract.

designed for smart contract development, where instructions are terminated with semicolons. The language is continuously evolving, with each stable release tied to a specific compiler version. This versioning ensures backward compatibility, allowing developers to compile code written in older Solidity versions and maintain stability for existing contracts. The following code fragments are intended to populate the space between the brackets following the contract declaration. In Code Fragment 2, we declare several variables to monitor the state of the system. When declaring a variable in Solidity, one must specify its type, visibility, and name. A

[*]Email: goffard@unistra.fr.

[†]Email: stephane.loisel@lecnam.net.

```
address public owner; // Ethereum address of the owner of the contract
uint256 public constant factor = 10 ** 4; // Numerical precision
uint256 public totalSupply; // Total supply of tokens
uint256 public rt = 10000; // Exchange rate of token against ETH
uint256 public sn2 = 0; // Variance of the liability
uint256 public mu = 0; // cumulative premium in excess of the pure premium
uint256 public SCR = 0; // Value of the Solvency Capital Requirement
uint256 public MCR = 0; // Value of the Minimum Capital Requirement
uint256 public Xt; // Reserve to calculate the exchange rate
uint16 public eta1 = 1000; // 10% loading (1000 basis points) to reward the token
    holders
uint16 public eta2 = 500; // 5% loading (500 basis points) to reward the
    policyholders
uint16 public qAlphaSCR = 25758; // standard normal quantile of order alpha = 0.995
uint16 public qAlphaMCR = 10364; // standard normal quantile of order alpha = 0.995
uint256 public Nt; // Counter for the number of insurance contracts
```

Code Fragment 2: Global variable declaration that play the role of the state variable of the system.

variable of type `address` stores a 20-byte value that represents the location of accounts (both externally owned accounts and smart contracts) on the Ethereum network. In Solidity, numbers can only be stored as integer values in variables of type `uint.x`, where *x* specifies the bit length of the number. Other basic types include strings, booleans, and fixed-size byte arrays[1]. In Code Fragment 3, we introduce a custom data type to store the relevant information for a parametric insurance contract. The `bytes32` type allows for storing a fixed-size 32-byte value instead of a string. To handle event descriptions, which are typically strings, we convert them into 32-byte arrays using the helper function `stringToBytes32`, provided in Code Fragment 19 in **??**. Storing strings can be problematic because they are variable-sized arrays, making storage costs unpredictable. In contrast, `bytes32` provides a more efficient and predictable alternative for fixed-size data. Code Fragment 5 defines an array to store the Ethereum addresses of all token holders.

This array is essential for redistributing funds to token holders in the event of a smart contract liquidation. The implementation of this functionality is provided in Code Fragment 18, which defines the `redistributeToInvestors` function. Code Fragment 5 declares several mappings used to store and manage key data structures

[1] See https://docs.soliditylang.org/en/latest/types.html

```
struct InsuranceContract {
        address customer; // Address of the policyholder
        bytes32 EventDescription; // Description of the insured event
        uint256 l; // Payout amount
        uint256 p; // Probability of the event
        uint16 eta; // loading of the premium at the underwritting time
        uint256 refund; // Refund amount in case of bankruptcy
        uint8 status; // Status of the contract 0 = open, 1 = closed and settled, 2 =
            closed without compensation, 3 = refunded
    }
```

Code Fragment 3: The `struct` type to store the information relative to an insurance policy.

```
    address[] public investorAddresses; // Array to store investor addresses
```

Code Fragment 4: Array declaration to store the addresses of the token holders

within the smart contract.

```
    mapping(uint256 => InsuranceContract) public insuranceContracts; // Mapping of
        insurance contracts
    mapping(address => uint256) public Yt; // Balance of investors in tokens
```

Code Fragment 5: Mapping declaration to iterate over the insurance policies and to keep track of the token holdings of the investors

Mappings in Solidity function as data dictionaries. The first mapping stores all the contracts that have been underwritten, while the second keeps track of the token balance for each investor. To monitor changes in the system, events are emitted whenever a significant state change occurs. Code Fragment 6 defines the events associated with these state transitions. The events are closely tied to the possible actions of the smart contract, as described in Section 3 of our paper. Each time an event is emitted, a corresponding record is added to the blockchain, enabling users to monitor the system's state. However, it is crucial to keep events as concise as possible, as storing data on the blockchain incurs significant costs. Developers must design events carefully so that the state of the system can be reconstructed through post-processing of the event logs.

```
event ParametersUpdated(uint16 newEta1, uint16 newEta2, uint16 newQAlphaSCR, uint16
    newQAlphaMCR);
event Fund(address indexed from, uint256 x, uint256 y);
event Burn(address indexed from, uint256 x, uint256 y);
event InsuranceUnderwritten(uint256 indexed contractId, address indexed customer,
bytes32 indexed EventDescription, uint256 payoutAmount, uint8 status);
event ClaimSettled(uint256 indexed contractId, address indexed customer, bool
    payoutTransferred, uint256 Xt);
```

Code Fragment 6: Event declaration

For instance, the `ClaimSettled` event includes a `bool` parameter, which indicates whether a compensation has been paid. This allows users to infer the outcome of a claim efficiently. To create an instance of `ParametricInsurance`, a `constructor` function is required, as shown in Code Fragment 7.

```
constructor() {
    owner = msg.sender;
}
```

Code Fragment 7: Consructor declaration

Since our smart contract receives ETh from investors and policyholders, then our code must include a `receive()` function as in Code fragment 8.

```
receive() external payable {}
```

Code Fragment 8: `receive` function

The owner of the contract—the blockchain user who deployed it—has the authority to modify the premium loading parameters and risk levels. This functionality is implemented through the function defined in Code Fragment 9.

The `updateParameters` function is public, meaning it can be called externally to interact with the smart contract. The function takes as input the new values for the safety loadings η_1 and η_2, as well as the new quantile values of the standard normal distribution used to calculate the SCR and MCR. A `require` statement

```
function updateParameters(uint16 newEta1, uint16 newEta2, uint16 newQAlphaSCR, uint16
    newQAlphaMCR) public {
 require(msg.sender == owner, "Only the owner can update parameters");
 eta1 = newEta1;
 eta2 = newEta2;
 qAlphaSCR = newQAlphaSCR;
 qAlphaMCR = newQAlphaMCR;
 SCR = (sqrt(sn2) * qAlphaSCR) / factor    - mu;
 MCR = (sqrt(sn2) * qAlphaMCR) / factor    - mu;
 if (Xt < MCR) {
         refundPremiumToPolicyholders();
         if (Xt > 0){
             redistributeToInvestors();
         }
         SCR=0;
         MCR=0;
         sn2=0;
         mu=0;
         rt=10000;
         Xt=0;
     }

 emit ParametersUpdated(newEta1, newEta2, newQAlphaSCR, newQAlphaMCR);
 }
```

Code Fragment 9: The `updateParameters` function

ensures that only the contract owner is authorized to execute this function. Additionally, the `emit` statement emits an event to notify all users of the parameter changes.

Modifying the MCR risk level may cause the surplus X_t to fall below the MCR threshold. In such cases, the smart contract must be liquidated. Active policy premiums are refunded to policyholders using the `refundPremiumToPolicyholders` function, as shown in Code Fragment 17. Any remaining funds are redistributed to token holders through the `redistributeToInvestors` function, provided in Code Fragment 18.

Next, we examine the `fundContract` function in Code Fragment 10.

```
function fundContract() public payable {
     require(msg.value > 0, "Funding amount must be greater than 0");
     uint256 tokensToMint = (msg.value * (factor)) / rt;
     totalSupply += tokensToMint;
     Xt += msg.value;
     if (Yt[msg.sender] == 0) {
         investorAddresses.push(msg.sender);
         }
     Yt[msg.sender] += tokensToMint;
     emit Fund(msg.sender, msg.value, tokensToMint);
 }
```

Code Fragment 10: The `fundContract` function

The `fundContract` function does not take any parameters. It is a public and payable function, meaning that when this function is called, the user can include an amount of ETH to be transferred to the smart contract. When deployed on the Ethereum blockchain, a contract is associated with an Ethereum address, enabling it to receive ETH. A `require` statement ensures that the caller includes ETH in the transaction, which is accessed via the `msg` object[2].

By providing an amount x of ETH to the contract, the investor receives y tokens, referred to as `tokensToMint`. Recall that `factor` is used to provide decimal precision, as Solidity does not support floating-point arithmetic. The function updates the global variables `Xt` (the surplus), `totalSupply`, and the mapping `Yt` (representing token balances). Finally, it emits an event to document the funding of the smart contract.

We now turn our attention to the `withdraw` function, described in Code Fragment 11.

The `withdraw` function takes the number of tokens to be burned as its parameter. The first `require` statement ensures that the investor holds a sufficient number of tokens to burn. The amount of ETH to be transferred in exchange for the tokens is calculated and stored in the local variable `etherAmount`. A second `require` statement verifies that this transfer does not cause the reserves to fall below the solvency capital requirement. The instruction `payable(msg.sender).transfer(etherAmount);` executes the transfer of ETH from the smart

[2]Other attributes of the `msg` object can be found at https://docs.soliditylang.org/en/latest/units-and-global-variables.html

```
function withdraw(uint256 _value) public returns (bool success) {
        require(Yt[msg.sender] >= _value, "Insufficient balance");
        uint256 etherAmount = _value * rt / (factor);
        require(Xt - SCR >= etherAmount, "Insufficient contract balance");

        Yt[msg.sender] -= _value;
        totalSupply -= _value;
        Xt -= etherAmount;
        payable(msg.sender).transfer(etherAmount);

        emit Burn(msg.sender, etherAmount, _value);

        return true;
    }
```

Code Fragment 11: The `withdraw` function

contract to the caller of the `withdraw` function.

Additionally, a Burn event is emitted to document the transaction. By tracking both Fund and Burn events, we can determine the identity of investors and their respective token holdings. The reserves provided by investors are intended to support customers in underwriting insurance policies. Customers can first inquire about the premium by calling the `getQuote` function.

```
    function getQuote(string memory EventDescription, uint256 l) public view returns (
        uint256) {
        uint256 p = getRandomNumber(EventDescription);
        uint256 pp = (l * p) / factor;
        uint256 cp = (pp * (factor + eta1 + eta2)) / factor;
        return cp;
    }
```

Code Fragment 12: The `getQuote` function

The `getQuote` function is of the `view` type, meaning it does not send a transaction to the blockchain and therefore does not incur any gas fees. This function returns the commercial premium associated with the

event specified as a string parameter, `EventDescription`. The quote is determined by generating a random probability p using the helper function `getRandomNumber`, provided in Code Fragment 20.

In practice, this random generation should be replaced with a more accurate estimation of the probability of the insured event occurring. Such an estimation could be derived using a predictive model and historical data retrieved from an external API. If the customer finds the quote acceptable, they can proceed to underwrite the policy and become a policyholder by calling the `underwritePolicy` function, described in Code Fragment 13.

To underwrite a policy, the customer must include exactly the premium amount returned by the `getQuote` function. Additional verifications ensure that the SCR resulting from adding a new policy does not exceed the available surplus. A new `InsuranceContract` instance, referred to as `policy`, is created and added to the mapping `insuranceContracts`. An `InsuranceUnderwritten` event is emitted to document the action.

The normal approximation requires updating the variance of the liability (`sn`) and the cumulative premium that has not yet been acquired (`mu`). Updating the SCR involves using the square root function, which is not a built-in feature in Solidity. To address this limitation, we implemented a custom `sqrt` function, provided in Code Fragment 21.

A parametric insurance contract can be settled by calling the `settle` function, provided in Code Fragment 14.

The `settle` function takes the contract identifier and a description of the observed event as parameters. The parameter `EventObserved` is a string passed to the `getRandomNumber` function to generate a random number between 0 and 1. If this number is smaller than the random number p generated by applying `getRandomNumber` to `EventDescription`, a compensation must be paid. Consequently, the probability of paying compensation is exactly p. The payout is transferred using the `sendPayout` function, provided in Code Fragment 15.

The internal function `updateReserveAfterSettlement`, shown in Code Fragment 16, updates the surplus based on whether compensation was paid. If a compensation is issued, the surplus may fall below the MCR level, triggering the termination of the contract. In such cases, the premiums of active policies are refunded to policyholders via the `refundPremiumToPolicyholders` function, detailed in Code Fragment 17. Any remaining surplus is redistributed to token holders using the `redistributeToInvestors` function, as provided in Code Fragment 18.

This function verifies whether the balance, accessed via `address(this).balance`, is sufficient to cover the agreed compensation. The `updateReserveAfterSettlement` function, which is part of the `settle` function, is detailed in Code Fragment 16.

```
function underwritePolicy(string memory EventDescription, uint256 l) public payable
    {
    uint16 eta = eta1 + eta2;
    uint256 p = getRandomNumber(EventDescription);
    uint256 pp = (l * p) / factor;
    uint256 cp = (pp * (factor + eta)) / factor;
    uint256 cp1 = (pp * (factor + eta1)) / factor;
    sn2 += (l * l) * p * (factor-p) / factor / factor;
    mu += p * l * eta / factor / factor;
    SCR = (sqrt(sn2) * qAlphaSCR) / factor   - mu;
    MCR = (sqrt(sn2) * qAlphaMCR) / factor   - mu;
    uint256 tokensToMint = (eta2 * pp) / rt;
    totalSupply += tokensToMint;
    if (Yt[msg.sender] == 0) {
        investorAddresses.push(msg.sender);}
    Yt[msg.sender] += tokensToMint;
    rt = totalSupply > 0 ? (Xt * (factor)) / totalSupply : 1;
    require(msg.value == cp, "Incorrect premium amount sent");
    require(SCR <= Xt, "Payout amount exceeds reserve");
    (bool success, ) = payable(address(this)).call{value: msg.value}("");
    require(success, "Failed to transfer premium to reserve");
    Nt++;
    InsuranceContract memory policy = InsuranceContract({
        customer: msg.sender,EventDescription: stringToBytes32(EventDescription),l:
            l,p:p,
        eta: eta,refund:cp1,status: 0
    });
    insuranceContracts[Nt] = policy;
    emit InsuranceUnderwritten(Nt, msg.sender, stringToBytes32(EventDescription), l,
         0);
}
```

Code Fragment 13: The underwritePolicy function

```
    function settle(uint256 contractId, string memory EventObserved) public {
        InsuranceContract storage policy = insuranceContracts[contractId];
        uint256 pObserved = getRandomNumber(EventObserved);
        require(policy.customer != address(0), "Invalid contract ID");
        require(msg.sender == owner || msg.sender == policy.customer, "Not authorized");
        require(policy.status == 0, "Contract is not active");
        if (pObserved < policy.p) {
            sendPayout(payable(policy.customer), policy.l);
            policy.status = 1; // Settled with compensation
            updateReserveAfterSettlement(policy.l, policy.p, policy.eta, true);
            emit ClaimSettled(contractId, policy.customer, true, Xt);
        } else {
            updateReserveAfterSettlement(policy.l, policy.p, policy.eta, false);
            policy.status = 2; // Settled without compensation
            emit ClaimSettled(contractId, policy.customer, false, Xt);
        }
        if (Xt < MCR) {
            refundPremiumToPolicyholders();
            if (Xt > 0){
                redistributeToInvestors();
            }
            SCR=0; MCR=0; sn2=0; mu=0; rt=10000; Xt=0;
        }
    }
```

Code Fragment 14: The settle function

```
function sendPayout(address payable customer, uint256 l) internal {
        require(address(this).balance >= l, "Insufficient reserve for payout");
        (bool success, ) = customer.call{value: l}("");
        require(success, "Failed to send payout to policyholder");
    }
```

Code Fragment 15: The sendPayout function

```
function updateReserveAfterSettlement(uint256 l, uint256 p, uint256 cp, bool settlement)
     internal {
        if (settlement) {
            Xt += cp;
            Xt -= l;
        } else {
            Xt += cp;
        }
        sn2 -= (l * l) * p * (factor-p) / factor / factor ;
        mu -= p * l / factor;
        SCR = (sqrt(sn2) * qAlpha) / factor   - mu * eta / factor;
        rt = totalSupply > 0 ? (Xt * (factor)) / totalSupply : 1;
    }
```

Code Fragment 16: The `updateReserveAfterSettlement` function

The `updateReserveAfterSettlement` function updates the reserve and adjusts the level of risk capital accordingly. The `settlement` parameter of `updateReserveAfterSettlement` is a boolean that indicates whether a compensation has been paid. Regardless of the outcome, the premium associated with the contract is earned, and the SCR is updated to reflect the closure of the contract. Additionally, the exchange rate is recalculated, ensuring no division by zero occurs during the process.

Both `sendPayout` and `updateReserveAfterSettlement` are marked as `internal`, meaning they cannot be called from outside the smart contract, ensuring they are used only within the contract's internal logic. The function `refundPremiumToPolicyholders` in Code Fragment 17 is also internal.

The function iterates through all active policies to refund premiums to the policyholders. If the remaining surplus is insufficient to fully refund all premiums, the policyholders are weighted based on the premiums they have paid and are refunded proportionally to their contributions.

If any surplus remains after reimbursing the policyholders, it is redistributed among the token holders using the `redistributeToInvestors` function, as detailed in Code Fragment 18.

The function iterates over all token holders, distributing the remaining surplus proportionally to their token holdings based on the exchange rate updated after the reimbursement of premiums.

```
function refundPremiumToPolicyholders() internal {
    Xt = address(this).balance;
    require(Xt > 0, "Nothing to refund");
    uint256 TotalRefund = 0;
    uint256 ActivePolicyCount = 0;
    // Calculate total refund amount for active policies
    for (uint256 i = 1; i <= Nt; i++) {
        if (insuranceContracts[i].status == 0) { // Active contract
            TotalRefund += insuranceContracts[i].refund;
            ActivePolicyCount++;
        }
    }
    require(TotalRefund > 0, "No active policies to refund");
    // Check if the reserve is sufficient for full refunds
    if (Xt >= TotalRefund) {
        // Fully refund all policyholders
        for (uint256 i = 1; i <= Nt; i++) {
            if (insuranceContracts[i].status == 0) { // Active contract
                uint256 refundAmount = insuranceContracts[i].refund;
                insuranceContracts[i].status = 3; // Mark as refunded
                sendPayout(payable(insuranceContracts[i].customer), refundAmount);
                Xt -= refundAmount;}}
    } else {
        // Prorated refunds
        for (uint256 i = 1; i <= Nt; i++) {
            if (insuranceContracts[i].status == 0) { // Active contract
                uint256 proratedRefund = (insuranceContracts[i].refund * Xt) /
                    TotalRefund;
                insuranceContracts[i].status = 3; // Mark as refunded
                sendPayout(payable(insuranceContracts[i].customer), proratedRefund);}}
        Xt = 0; // All reserve used}
}
```

Code Fragment 17: The `refundPremiumToPolicyholders` function

```
function redistributeToInvestors() internal {
        Xt = address(this).balance;
        require(Xt > 0, "Nothing to distribute");
        for (uint256 i = 0; i < investorAddresses.length; i++) {
            address investor = investorAddresses[i];
            uint256 investorShare = (Yt[investor] * Xt) / totalSupply;
            Yt[investor] = 0;
            if (investorShare > 0) {
                sendPayout(payable(investor), investorShare);
            }
        }
        Xt = 0; // All reserve distributed
        totalSupply = 0;
        delete investorAddresses;
    }
```

Code Fragment 18: The `redistributeToInvestors` function

Additionally, we have implemented three helper functions in Solidity. The first function converts a string value into a 32-byte number, as shown in Code Fragment 19.

```
    function stringToBytes32(string memory source) private pure returns (bytes32 result)
         {
        bytes memory tempEmptyStringTest = bytes(source);
        if (tempEmptyStringTest.length == 0) {
            return 0x0;
        }

        assembly {
            result := mload(add(source, 32))
        }
    }
```

Code Fragment 19: Function to convert a character string into a 32 bytes long array.

The second function computes an integer-valued approximation of the square root of an integer, as shown in Code Fragment 21.

The third and final function takes a string value as input and returns a random number between 0 and 10,000. This number represents the probability of occurrence of the event described by the string.

```
function getRandomNumber(string memory EventDescription) public pure returns (uint)
    {
    // Generate a hash based only on flightNumber and departureDate
    uint randomHash = uint(keccak256(abi.encodePacked(EventDescription)));

    // Limit the random number to the range [0, 10000]
    return randomHash % 10001;
}
```

Code Fragment 20: Function to generate a random probability from a string value.

```
function sqrt(uint256 y) public pure returns (uint256 z) {
    if (y > 3) {
        z = y;
        uint256 x = y / 2 + 1;
        while (x < z) {
            z = x;
            x = (y / x + x) / 2;
        }
    } else if (y != 0) {
        z = 1;
    }
}
```

Code Fragment 21: Function to calculate a root square of an integer-valued number.