智能合约安全审计报告

路印协议智能合约第二版

SECBIT

2018 年 12 月 15 日

1. 简介

路印协议智能合约第二版（下称 **LPSC**）是一组路印生态系统中的智能合约。它们被用来检查环路矿工（ring-miners）提交的订单环路（order-rings）是否符合规范，以用户利益为目的进行可信结算和转账，用交易费激励环路矿工和钱包，并产生相应的以太坊事件。

SECBIT 实验室从2018年10月15日至2018年12月15日对 LPSC 进行了安全审计。

1.1 LPSC 基本信息

- **项目网站**
  - https://loopring.org/
- **项目白皮书**
  - https://github.com/Loopring/whitepaper
- **审计合约代码**
  - https://github.com/Loopring/protocol2/tree/audit-1012
  - Commit `10100aa616223439516c48f2c76ef386e8f996ff`

根据路印白皮书，LPSC 可以被部署在多种区块链上，但本审计报告只针对 LPSC 的以太坊版本。

1.2 审计流程

本审计流程围绕 LPSC 的形式化模型进行，具体流程如下：

1. 根据 LPSC 的实现定义其形式化模型。该形式化模型是后续审计步骤的基础。
2. 人工检查 LPSC 形式化模型与 LPSC 实现以及路印白皮书之间的一致性，借此检查是否存在一些明显问题。
3. 基于形式化模型，形式化证明部分 LPSC 关心的性质，以展示 LPSC 具备良好的行为。如果无法证明某个性质，那么合约中可能存在相应的问题。

上述形式化模型和性质的形式化证明已在交互式形式化证明工具 **Coq** 中实现。Coq 实现代码在：https://github.com/sec-bit/loopring-protocol2-verification。

1.3 问题列表

本次审计过程中发现了 LPSC 存在以下问题。所有问题或者已经在最新版本的 LPSC 中被修复，或者通过 LPSC 之外的手段解决，或者实际中很难造成损失不需处理。

#	问题类型	问题描述	严重级别	当前状态	章节
1	实现错误	相邻订单购买/出售的代币类型可能不一致	高	已修复	4.1
2	实现错误	withdraw() 没有正确处理调用外部合约调用失败的情况	高	已修复	4.2
3	实现错误	未充分检查多个 all-or-none 订单存在于多个环路中的情况	中	已修复	4.3
4	潜在风险	订单所有者或中介可能对环路矿工进行 GAS 攻击	低	真实使用场景下不会出现	4.4
5	潜在风险	恶意代币合约会造成高 GAS 消耗	低	通过中继加以保护	4.5
6	潜在风险	舍入误差导致成交金额计算不准确	低	真实使用场景下损失可忽略	4.6

本报告后续章节组织如下：

- 第 2 节对 LPSC 及其形式化模型进行了分析
- 第 3 节列出通过形式化证明检查的 LPSC 的性质
- 第 4 节对审计中发现的问题进行了详细描述
- 第 5 节列出 LPSC 开发中的最佳实践
- 第 6 节总结 LPSC 的实现质量

2. 合约分析

2.1 LPSC 总览

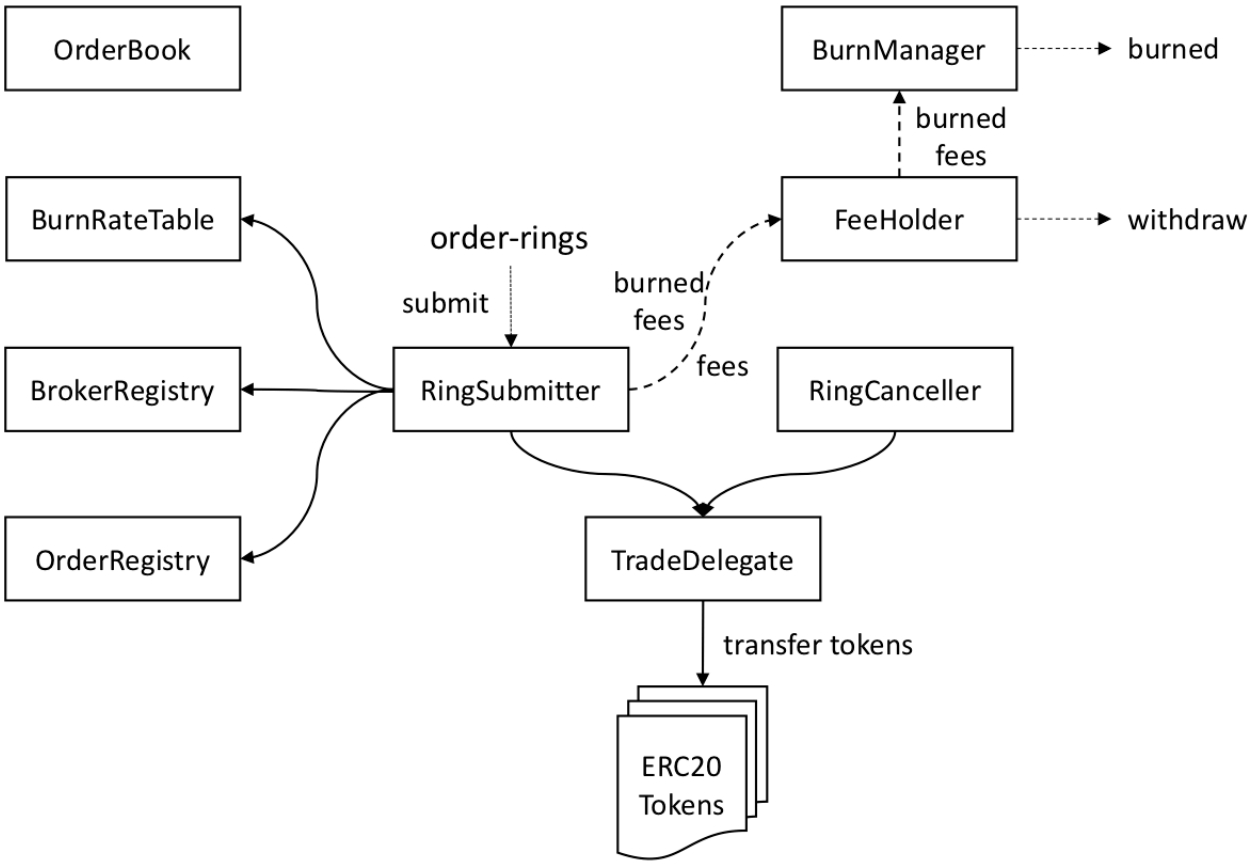
路印订单在 LPSC 中被组织成订单环路。订单环路中的每个订单向前一订单出售代币，并从后一订单购买代币。目前实现中，一个订单环路必须由 2 ~ 8 个订单构成。每个订单主要包含以下信息：

- 订单所有者
- 被出售代币的种类和数量
- 要购买代币的种类和数量
- 用于计算各种交易费的参数
- 订单的中介（broker）

用户可以将订单提交给 **中继网络**（relay-mesh）并由其中的环路矿工（ring-miner） 生成订单环路，也可以用户间直接进行订单匹配。后者被称为 **P2P订单**。

为了激励路印协议的参与者，部分用于交易的代币会被用作付给环路矿工和钱包的交易费。其中，矿工费的一部分会被燃烧掉，燃烧比例由交易费使用的代币种类和公开可见的**燃烧率**（burn rate）表决定。除此之外，LPSC 允许环路矿工免除指定订单的部分交易费，甚至可以把从其他订单得到的交易费作为返佣付给指定订单。

LPSC 由多个独立的智能合约构成并分别部署，以此来分离相对独立的功能，如下图所示。



- `RingSubmitter` 接受订单环路、检查环路是否有效、计算交易费用和成交金额、最终进行代币转账。订单间的代币转账通过调用对应代币合约的 `transferFrom()` 函数实现。交易产生的费用被记录在 `FeeHolder` 合约中，费用接受者可以通过调用 `FeeHolder` 合约的对应函数取走交易费。
- `RingCanceller` 提供了一组函数用来取消订单。
- `TradeDelegate` 提供了多种功能：
    - 为 `RingSubmitter` 提供了安全调用代币合约进行交易的机制（例如调用 ERC20 合约的 `transferFrom()`）
    - 为 `RingCanceller` 实现了取消订单的接口
    - 为 `RingSubmitter` 提供了记录订单成交额度的接口
    - 查询订单成交额度和取消状态的接口
- `FeeHolder` 会被 `RingSubmitter` 调用并记录交易产生的费用。它还提供了接口让费用接受者可以取走交易费。
- `BurnManager` 提供了一个可以被任何人调用的接口，用于真正燃烧掉暂时被 `FeeHolder` 持有的燃烧交易费。

- `BurnRateTable` 为每种代币维护一个燃烧率。`RingSubmitter` 和其它人可以通过 `BurnRateTable` 提供查询函数得到某种代币当前的燃烧率。`BurnRateTable` 同时实现了一套调整燃烧率的机制。
- `BrokerRegistry` 提供了查询、注册、注销订单中介的方法。查询接口被 `RingSubmitter` 使用，来进行订单校验。
- `OrderBook` 提供了一个公开的订单账簿。订单所有者和中介可以将自己的订单记录在这个订单账簿中。其它人可以通过订单的 hash 在这个订单账簿中查询相应的订单信息。
- `OrderRegistry` 允许用户直接将订单注册在 LPSC 中，从而避免之后的交易过程中 `RingSubmitter` 可能会进行的订单签名校验。

2.2 形式化模型

本次审计主要围绕着 LPSC 的形式化模型进行。该形式化模型基于 LPSC 的实现构建，通过形式化语言抽象和描述 LPSC 的行为。后续的人工审阅和相关性质的形式化证明都基于该形式化模型进行。

我们在一个交互式定理证明工具 **Coq** 中实现了上述的形式化模型和证明。Coq 已经被广泛应用在学术和工业项目，如对操作系统、编译器、和密码学代码库的形式化验证中。Coq 中证明的可靠性由Coq 的底层类型系统和逻辑保证。

LPSC 形式化模型和相关性质证明的 Coq 的实现可以在这个链接找到 https://github.com/sec-bit/loopring-protocol2-verification。*本报告主要对形式化模型的主体框架和其中一些关键部分进行解释，并不对所有细节进行深入讲解。*

该形式化模型将 LPSC 定义为一个抽象状态机：

- 抽象状态机通过其**机器状态**（下称 **World State**）抽象表示 LPSC 和相关的代币合约 Storage 中存储数据。我们可以把 World State 想像为一个多元组，其中每个元素描述一个或多个 Storage 变量的语义。例如，类型为 `mapping(address => bool)` 的 Storage 变量可以用一个从自然数集合到布尔值集合的全映射表示。
- 抽象状态机通过**状态迁移**描述对 LPSC 和代币合约的公共函数的一次 Message Call 成功调用会如何修改 World State、产生什么事件和返回值。状态转移只包含不会产生 revert 的 Message Call 的执行。在某个条件下（例如，一组特殊的参数和 Storage 变量值的组合）发生 revert 的 Message Call 可以表示为，在该条件下，不存在该 Message Call 对应的状态转移。

在形式化模型的顶层定义中：

- World State 由 Coq 文件 `Models/States.v` 中的 `WorldState` 定义，它包含了 LPSC 的各个合约以及外部代币合约的**子状态**和一部分区块链状态（即 `Record WorldState` 中的各个 `wst_*` 域）。
- 所有可能的 Message Call 由 Coq 文件 `Models/Messages.v` 中的 `Message` 归纳定义，它包含了所有可供外部调用的 LPSC 函数和代币合约函数。
- 所有可能的事件由 Coq 文件 `Models/Events.v` 中的 `Event` 归纳定义，它包含了 LPSC 可能产生的所有事件，以及一些用于建模和证明的辅助事件。
- 状态迁移由 Coq 文件 `Models/TopModel.v` 中的 `lr_steps` 定义：
    - 在 `lr_steps` 的归纳定义中，基条款 `lr_steps_nil` 的含义是没有 Message Call 被执行时，状态不发生迁移，也不会产生事件或返回值。
    - 在 `lr_steps` 的归纳定义中，归纳条款 `lr_steps_cons` 的含义是一个或多个 Message Call 被成功执行时，其效果相当于每个 Message Call 按顺序依次执行。

    例如，

    ```
    lr_steps wst (msg0::msg1::nil) wst' v (evt0::evt1::evt2::nil)
    ```

    的含义是，以太坊成功执行了 Message Call `msg0` 和 `msg1`， World State 从 `wst` 变为 `wst'`，产生事件 `evt0`、 `evt1` 和 `evt2`，并返回 `v`。

在形式化模型定义的底层，状态迁移被细分为各个合约相关状态迁移。每个合约相关的状态迁移由该合约公共函数的形式化规范组成。每个函数的形式化规范从以下三个方面描述函数的行为 （由 Coq 文件 `Libs/LibModel.v` 中的 `FSpec` 定义）：

- `fspec_require` 描述为了成功执行该函数而不产生 revert 所需要满足的条件；
- `fspec_trans` 描述了函数成功执行后 World State 发生的变化以及产生的返回值；
- `fspec_events` 描述了函数成功执行会产生的事件序列。

本节以下部分结合相应的子状态和函数规范介绍 LPSC 的各个合约。

2.3 RingSubmitter

`RingSubmitter` 是 LPSC 的核心合约。它只有一个公共函数 `submitRings(bytes data)`，供环路矿工或用户提交订单环路。该函数会根据订单环路计算订单间的交易金额、交易费用，并最终完成代币和费用转账。

- 形式化模型涉及的合约文件
    - `contracts/impl/RingSubmitter.sol`
    - `contracts/impl/Data.sol`

    - contracts/helper/{Mining, Order, Participation, Ring}Helper.sol

- 相关的 Coq 文件
    - Models/{RingSubmitter, States, Events, Messages}.v

2.3.1 子状态

RingSubmitter 的子状态被表示为 WorldState 中 wst_ring_submitter_state，由 Coq文件 State.v 中的 RingSubmitterState 定义。该子状态的每个域代表 RingSubmitter 中相应的 Storage 变量（例如 submitter_lrcTokenAddress 域表示 storage 变量 lrcTokenAddress）。

因为 submitRings() 函数主要在处理存放在 Memory 中的订单和环路等数据，而不是 Storage 中的数据，RingSubmitter的形式化模型定义中又额外定义了一个内部状态 RingSubmitterRuntimeState（参见 Coq文件 RingSubmitter.v）表示它们。在该内部状态中，

- submitter_rt_mining 由 MiningRuntimeState 定义，用于描述 Memory 中存放的环路挖矿相关的参数及其变化；
- submitter_rt_orders 由 list OrderRuntimeState 定义，用于描述 Memory 中存放的所有订单及其变化；
- submitter_rt_rings 由 list RingRuntimeState 定义，用于描述 Memory 中存放的所有环路及其变化；
- submitter_rt_token_spendables 是一个从代币所有者和代币合约地址到 TradeDelegate 可花费的（spendable）代币数量的映射；
- submitter_rt_broker_spendables，是一个从订单中介、订单所有者和代币地址到该订单中介可花费的代币数量的映射，它和上面的 submitter_rt_token_spendables 被用在 submitRings() 规范中描述各种可以使用代币数量的计算。

2.3.2 形式化规范

submitRings() 形式化规范由 Coq 文件 RingSubmitter.v 中的 submitRings_spec 定义。

对于 submitRings() 的 byte array 类型的参数，我们使用其反序列化之后的语义描述，即描述成一个订单列表、一个环路列表以及挖矿相关的参数的组合。

因为 submitRings 及其调用的各级子函数的实现较大而且比较复杂，所以它的形式化规范 submitRings_spec 被拆解成一系列子规范，每个子规范仅描述 submitRings 的一部分行为。

- update_orders_hashes_subspec, update_orders_brokers_and_interceptors_subspec, get_filled_and_check_cancelled_subspec和 check_orders_subspec 分别对各种更新和检查订单参数的过程建模，这些参数包括订单哈希、订单中介、已交易金额、订单是否已取消、剩余可用代币和交易费；
- update_rings_hash_subspec 对更新环路哈希的过程进行建模；
- update_mining_hash_subspec 和 update_miner_interceptor_subspec 分别对更新和检查订单中环路矿工相关的参数的过程进行建模；
- check_miner_signature_subspec 和 check_orders_dual_sig_subspec 分别对检查订单/环路/矿工哈希和签名的过程进行建模；
- calc_fills_and_fees_subspec 对成交金额交易费计算的过程进行建模；
- validate_AllOrNone_subspec 对校验 all-or-none 订单的过程进行建模；
- calc_and_make_payments_subspec 对最终交易代币和收取交易费进行建模。

对于这种复杂的实现，它的规范及子规范主要通过在 Coq 重写 submitRings() 及其子函数的关键部分的方式进行定义。比如 submitRings() 调用的子函数 RingHelper::checkOrdersValid() 的 Coq 形式化规范如下所示，对 Coq 代码的具体的解释参见 (* *) 中的注释。

（在 4.1 节中我们将提到 checkOrdersValid() 的实现存在问题，这里我们针对修复后的版本进行建模。）

```
(* Model the loop in checkOrdersValid() inductively.

   '_ring_orders_valid orders pps ps' means orders referred by
   participations 'ps' are valid.

   * 'pps' represents participations that have been iterated,
   * 'ps' represents the rest participations in the ring,
   * 'orders' represents all orders submitted to 'submitRings()'.

   If 'ps' represents all participations in a ring, then
   '_rings_orders_valid orders nil ps' represents the ring is valid.
*)
Inductive _ring_orders_valid
          (orders: list OrderRuntimeState)
          (pps: list Participation)
: list Participation -> Prop :=
| RingOrdersValid_nil:
  (* Base case: an empty ring (nil) is always valid *)
    _ring_orders_valid orders pps nil

| RingOrdersValid_cons:
  (* Inductive case:
```

```
      Suppose,
        * 'p' is the current participation being checked
        * 'p_ord' is the order referred by 'p'
        * 'pp' is the previous participation of 'p'
        * 'pp_ord' is the order referred by 'pp'
        * 'pps' are all checked participations
        * 'ps' are remaining participations (may be nil) except 'p'

      If
        * the valid field of the current participation 'p' is true,
        * the selling token of the current order 'p_ord' and the
          buying token of the previous order 'pp_ord' match, and
        * orders referred by remaining participations 'ps' are valid,
      then it can conclude orders referred by participations 'p::ps'
      are valid.
  *)
    forall p ps pp p_ord pp_ord,
      get_pp pps ps = Some pp ->
      nth_error orders (part_order_idx p) = Some p_ord ->
      nth_error orders (part_order_idx pp) = Some pp_ord ->
      (* the valid field of the current participation is true *)
      ord_rt_valid p_ord = true ->
      (* tokens match *)
      order_tokenS (ord_rt_order p_ord) =
      order_tokenB (ord_rt_order pp_ord) ->
      (* orders in the rest of participations are valid *)
      _ring_orders_valid orders (pps ++ p :: nil) ps ->
      _ring_orders_valid orders pps (p :: ps)
.

(* Orders in ring 'r' are valid. *)
Definition ring_orders_valid
           (r: RingRuntimeState)
           (orders: list OrderRuntimeState) : Prop :=
  (* r.valid == true *)
  ring_rt_valid r = true /\
  let ps := ring_rt_participations r in
  (* ... and r has at least 2 and at most 8 participations (ps) *)
  1 < length ps <= 8 /\
  (* ... and all orders referred by participations are valid *)
  _ring_orders_valid orders nil ps.
```

2.4 RingCanceller

RingCanceller 为订单中介提供了一组公共函数，用于在特定时刻取消该中介的订单。每个函数提供了一种独特的指定待取消的订单的方法，比如通过订单哈希、订单所有者、订单购买或出售代币的种类、或者它们的某种组合。submitRings() 不会成交包含已取消订单的回路。

- 形式化模型涉及的合约文件
    - contract/impl/RingCanceller.sol
- 相关的 Coq 文件
    - Models/{RingCanceller, States, Events, Messages}.v

2.4.1 子状态

RingCanceller 的子状态被表示为 WorldState 中的 wst_ring_canceller_state，由 Coq 文件 States.v 中的 RingCancellerState 定义。该子状态的每个域代表 RingCanceller 中相应的 Storage 变量。

2.4.2 形式化规范

RingCanceller 每个公共函数由 Coq 文件 RingCanceller.v 中名称类似的函数规范描述。例如 cancelOrders() 的形式化规范为 cancelOrders_spec。

RingCanceller 的各个公共函数主要通过调用 TradeDelegate 来实现订单取消。因此它们对应的函数规范主要通过 TradeDelegate 的形式化模型定义。

以函数 cancelAllOrdersForTradingPair() 为例。 该函数调用 TradeDelegate 中的 setTradingPairCutoffs() 实现功能。相对应的，其形式化规范 cancelAllOrdersForTradingPair_spec 也通过 TradeDelegate 中的形式化规范所描述的 setTradingPairCutoffs() 的 fspec_require、spec_trans 和 fspec_events，来描述自身成功执行所需要满足的前置条件、成功执行产生的状态变化、返回值以及事件。

2.5 TradeDelegate

RingSubmitter 和 RingCanceller 都通过调用 TradeDelegate 来实现相关的功能，包括

- 进行 ERC20 代币转账，
- 取消订单，
- 记录订单成交金额，
- 查询订单的已成交金额和取消状态。

上述功能，除了查询功能，都被限制为只能被**已授权用户**（authorized users）在 TradeDelegate 没有被**暂停**（suspended）或**终止**（killed）的情况下调用。相应的，TradeDelegate 也提供了一组公共接口实现

- 授予/取消某个用户的授权；
- 暂停/恢复/终止 TradeDelegate。

由于 RingSubmitter 和 RingCanceller 对 TradeDelegate 的依赖，**暂停**或**终止** TradeDelegate 实际上会**暂停**或**终止**整个 LPSC 的交易订单和取消订单的功能。

- 形式化模型涉及的合约文件
    - contract/impl/TradeDelegate.sol
- 相关的 Coq 文件
    - Models/{TradeDelegate, States, Events, Messages}.v

2.5.1 子状态

TradeDelegate 的子状态被表示为 WorldState 中的 wst_trade_delegate_state，由 Coq文件 States.v 中的 TradeDelegateState定义。该子状态的每个域表示 TradeDelegate 的相应 Storage 变量。例如：

- delegate_owner 表示 TradeDelegate 的 owner 变量，即该合约所有者；
- delegate_suspended 表示 TradeDelegate 的suspended 变量，即该合约是否被暂停；
- delegate_authorizedAddresses 是一个以用户地址到布尔值的映射，它表示 TradeDelegate 合约的 authorizedAddresses 变量，用于记录用户是否已被授权。

2.5.2 形式化规范

TradeDelegate 的每个公共函数由 Coq文件 TradeDelegate.v 中名称类似的形式化规范描述。例如batchTransfer() 的形式化规范为 batchTransfer_spec。

在这些函数规范中，fspec_require 中描述了对函数调用者的限制：

- 对于要求调用者已被授权的函数，其规范中的 fspec_require 包含了对 sender（对应于合约中的 msg.sender）的约束，要求函数执行前的子状态中 delegate_authorizedAddresses 将 sender 映射为 true；
- 对于要求调用者只能是合约所有者的函数，其规范中的 fspec_require 包含了对 sender 的约束，要求函数执行前的子状态中 delegate_owner 与 sender 相等。

合约中与暂停相关的要求也被定义在 fspec_require 中：

- 对于要求合约未被暂停时才能被调用的函数，其规范中的 `fspec_require` 要求函数执行前的子状态中 `delegate_suspended` 为 `false`；
- 对于要求合约被暂停时才能被调用的函数，其规范中的 `fspec_require` 包含了与前者相反的约束。

2.6 FeeHolder

在 LPSC 中，包括钱包费、矿工费、燃烧费和分配给订单的费用在内的交易费都会被先转移和记录到 `FeeHolder` 合约中。交易费接受者可以通过调用 `FeeHolder` 合约自行取出暂存在 `FeeHolder` 中的交易费。取出交易费的功能通过调用对应代币合约的 `transfer()` 函数完成。

- 形式化模型涉及的合约文件
  - `contracts/impl/FeeHolder.sol`
- 相关的 Coq 文件
  - `Models/{FeeHolder, States, Events, Messages}.v`

2.6.1 子状态

`FeeHolder` 的子状态被表示为 `WorldState` 中的 `wst_feeholder_state`，由 Coq文件 `States.v` 中的 `FeeHolderState` 定义。该子状态的每个域表示 `FeeHolder` 的相应的 Storage 变量。例如：

- `feeholder_feeBalances` 被定义为 Coq 中从代币合约地址和费用接受者的二元组到暂存交易费余额的映射，用于表示 `FeeHolder` 合约中的 `feeBalances` 变量。

2.6.2 形式化规范

`TradeDelegate` 的每个公共函数由 Coq 文件 `FeeHolder.v` 中名称类似的形式化规范描述。例如 `withdrawBurned()` 的形式化规范为 `withdrawBurned_spec`。

其中，`batchAddFeeBalances(byte32[] batch)` 被 `RingSubmitter` 调用来计算各项交易费余额。在其形式化规范 `batchAddFeeBalances_spec` 中：

- `batchAddFeeBalances` 的 byte array 参数被抽象成了反序列化后的语义，即一个由代币地址、费用接受者、费用金额组成的多元组的列表（参见 Coq 文件 `Messages.v` 中的 `FeeBalanceParam`）。
- 该规范的 `fspec_trans` 要求该函数成功执行后，上述参数中对应的增加的交易费金额被累加到子状态中的 `feeholder_feeBalances` 相应项中。
- 该规范的 `fspec_require` 包含了对 `sender` 的约束 `is_authorized wst sender`，即要求 `batchAddFeeBalances()` 只能被已授权的用户调用。

`withdrawBurned(address token, uint value)` 被 `BurnManager` 调用，将要燃烧费从 `FeeHolder` 转移给 `BurnManager`，其参数`token` 表示要被燃烧的代币种类，`value` 表示要被燃烧的代币数量。其规范 `withdrawBurned_spec` 中：

- `fspec_require` 包含与 `batchAddFeeBalances_spec` 相同的约束，要求调用者已被授权。
- `fspec_require` 同时包含了对 `value` 的约束，要求调用该函数前，子状态中的 `feeholder_feeBalances token` 的值不少于 `value`，即在 `FeeHolder` 中有足够的燃烧费余额用于转账。
- `fpsec_trans` 要求该函数的成功执行必须产生以下所有效果
    - 函数返回值为 `true`；
    - 函数成功调用了 `token` 的 `transfer()`，并已将 `value` 数额的代币从 `FeeHolder` 转移给了调用者。

`withdrawToken(address token, uint value)` 可以被任何人调用，来取走暂存在 `FeeHolder` 中的 `value` 数额的 `token` 代币。其规范 `withdrawToken_spec` 中：

- `fspec_require` 不对调用者授权进行约束；
- `fspec_require` 包含对`value`的约束，与`withdrawBurned_spec`相同，要求调用者在 `FeeHolder` 中有足够的费用余额用于转账；
- `fspec_trans` 的定义与 `withdrawBurned_spec` 类似，要求
    - 函数返回值为 `true`；
    - 函数成功调用了 `token` 的 `transfer()`，并已将 `value` 数额的代币从 `FeeHolder` 转移给了调用者。

2.7 BurnManager

`BurnManager` 提供了公共的燃烧接口，让任何人可以过调用该接口来燃烧暂存在 `FeeHolder` 中暂存的待燃烧交易费用。

注意在本次审计的版本中，LPSC只允许燃烧 LRC 代币。

- 形式化模型涉及的合约文件
    - `contracts/impl/BurnManager.sol`
- 相关的 Coq 文件
    - `Models/BurnManager, States, Events, Messages}.v`

2.7.1 子状态

`BurnManager` 的子状态被表示为 `WorldState` 中的 `wst_burn_manager_state`，由 Coq 文件 `States.v` 中的 `BurnManagerState` 定义。该子状态的每个域表示 `BurnManager` 相应的 Storage 变量。

2.7.2 形式化规范

该合约只有一个公共函数 `burn(address token)` 提取 `FeeHolder` 中以 LRC 存储燃烧费用，并通过 LRC 代币合约的 `burn()` 函数真正燃烧掉这些费用。其规范 `burn_spec` 中：

- `spec_require` 包含对参数 `token` 的约束，要求其必须是 LRC 代币合约的地址；
- `spec_trans` 要求该函数成功的执行包含以下效果：
  - 函数返回值是`true`；
  - 成功执行了 `FeeHolder::withdrawBurned()`，将所有剩余待燃烧的 `token` 代币转移到 `BurnManager` 中；
  - 成功执行了 `token` 合约的 `burn()` 函数，燃烧掉前面转移到 `BurnManager` 中的燃烧费用。

2.8 BurnRateTable

`BurnRateTable` 负责管理代币燃烧率。

- 形式化模型涉及的合约文件
  - `contracts/impl/BurnRateTable.sol`
- 相关的 Coq 文件
  - `Models/{BurnRateTable, States, Events, Messages}.v`

在路印协议第二版中，特定百分比（燃烧率）的费用会被从付给环路矿工的交易费中扣除。这部分被扣除的费用最终会被燃烧掉，而不会被付给矿工。

所有代币被分成四个等级（Tier），每个等级都和其它等级不同的燃烧率。

Tier	P2P订单的燃烧率	非P2P订单的燃烧率
1	0.5%	5.0%
2	2.0%	20.0%
3	3.0%	40.0%
4	6.0%	60.0%

- LRC 被固定为 Tier 1.
- WETH 被固定为 Tier 3.
- 其他订单被初始化为 Tier 4，但可以通过调用 `upgradeTokenTier()` 来提升等级。
- 如果某种代币等级在 Tier 1 之下，任意用户都可以通过调用 `upgradeTokenTier()` 将其等级提升一级。但是每次调用该函数，调用者需要支付当前 LRC 总量的 0.5% 的费用，这些费用随后会被燃烧掉。
- 除了 LRC 和 WETH，任意不在 Tier 4 的代币都会在当前等级维持为 63,113,904 秒（730天，即大约2年）。如果在这期间没有人继续调用 `upgradeTokenTier()` 提升其等级，该代币将会被降级到 Tier 4。

2.8.1 子状态

`BurnRateTable` 的子状态被表示为 `WorldState` 中的 `wst_burn_rate_table_state`，由 Coq 文件 `States.v` 中的 `BurnManagerState`定义。

- `burnratetable_tokens` 被定义为 Coq 中的从代币合约地址到代币Tier及其有效期限的映射，用于表示 `BurnRateTable` 的 `tokens` 变量。

2.8.2 形式化规范

`BurnRateTable` 的每个公共函数都被 Coq 文件 `BurnRateTable.v` 名称类似的形式化规范描述。

其中，`getBurnRate(address token)` 可以被任何人调用，用来查询`token` 代币的 P2P 和非 P2P 燃烧率，这两个燃烧率被编码成一个`uint32`值。其规范 `getBurnRate_spec` 中：

- `spec_trans` 根据其子状态的 `burnratetable_tokens` 映射的 `token` 的级别按照上表计算对应的 P2P 和非 P2P 燃烧率，并编码在返回值中。

`getTokenTier(address token)` 可以被任何人调用，用来查询`token`的Tier。其规范 `getTokenTier_spec` 中：

- `spec_trans` 通过其子状态的 `burnratetable_tokens` 获得其映射的 `token` 的级别。并编码在返回值中。

`upgradeTokenTier(address token)` 可以被任何有足够 LRC 的人调用，通过消耗 LRC 来提升 `token` 的等级。这里 `token` 不可以是 LRC 或 WETH。在其规范 `upgradeTokenTier_spec` 中：

- `spec_requires` 要求：

    - 参数 `token` 不等于 `0`、LRC 合约地址以及 WETH 合约地址，
    - `token` 的当前等级不为 1，
    - 调用者拥有足够的 LRC（当前 LRC 总量的 0.5%）用于燃烧。
- `spec_trans` 要求函数的成功执行能够产生如下效果：
    - 成功调用 LRC 合约的 `totalSupply()` 获取当前的 LRC 总量，
    - 成功调用 LRC 合约的 `burnFrom()` 函数，并从调用者账户燃烧掉当前 LRC 总量的 0.5%，
    - 返回 `true`。
- `spec_events` 要求成功执行函数后产生的事件序列须包含 `TokenTierUpgraded`，指明 `token` 被提升到的等级。

2.9 BrokerRegistry

`BrokerRegistry` 提供了一组公共函数供订单所有者向 LPSC 注册自己的订单中介。`RingSubmitter` 在校验收到的订单时会调用 `BrokerRegistry` 来查询订单的中介信息。

- 形式化模型涉及的合约文件
    - `contracts/impl/BrokerRegistry.sol`
- 相关的 Coq 文件
    - `Models/{BrokerRegistry, States, Events, Messages}.v`

2.9.1 子状态

`BrokerRegistry` 的子状态被表示为 `WorldState` 中的 `wst_broker_registry_state`，由 Coq 文件 `States.v` 中的 `BrokerRegistryState`定义。

- `broker_registry_brokersMap` 定义为订单所有者地址到中介地址到中介信息 `Broker` 的映射。它抽象表示了该合约中的 `brokersMap` 和 `positionMap` 变量。

2.9.2 形式化规范

`registerBroker(address broker, address interceptor)` 由订单所有者调用，用于向 LPSC 注册订单的中介 `broker`，以及中介的 Interceptor 为 `interceptor`（如果其不为 0）。其规范 `registerBroker_spec` 中：

- `fspec_require` 要求
    - 参数 `broker` 不为 `0`；
    - `broker` 必须还没有被子状态中 `broker_registry_brokersMap` 映射，即

其还没有被注册过。
- `fspec_trans` 要求函数成功执行后 `broker_registry_brokersMap` 将调用者和 `broker` 映射到该中介的信息的，其中包含订单所有者的地址，中介的地址，中介 Interceptor 的地址。
- `fspec_events` 要求函数成功执行后会产生 `BrokerRegistered` 事件，指明调用者向 LPSC 注册了 `broker` 为其中介，以及 `interceptor` 为其 Interceptor。

`unregisterBroker(address addr)` 由订单所有者调用，用来向 LPSC 注销之前注册过的地址为 `addr` 的中介。其规范 `unregisterBroker_spec` 中：

- `fspec_require` 要求
  - 参数 `addr` 不为 `0`；
  - `addr` 必须已经在子状态的 `broker_registry_brokersMap` 中被映射，即调用者的中介 `addr` 已经被注册过。
- `fpsec_trans` 要求函数成功执行后的子状态中 `broker_registry_brokersMap` 不会将调用者和 `addr` 映射到任何中介数据上。
- `fspec_events` 要求函数成功执行后产生 `BrokerUnregistered` 事件，指明调用者向 LPSC 注销了地址为 `addr` 的中介。

`unregisterAllBrokers()` 由订单所有者调用，用来向 LPSC 注销该调用者的所有中介。其规范 `unregisterAllBrokers_spec` 中：

- `fspec_trans` 要求函数成功执行后 `broker_registry_brokersMap` 不会将调用者和任何地址映射到某中介信息。
- `fspec_events` 要求函数成功执行后产生 `AllBrokersUnregistered` 事件，指明所有该调用者的中介都已从 LPSC 注销。

`getBroker(address owner, address addr)` 检查订单所有者 `owner` 的地址为 `addr` 的中介是否已被注册。如果已注册，则同时返回该中介的 Interceptor。其规范 `getBroker_spec` 中：

- `fspec_trans` 查询子状态中的 `broker_registry_brokersMap` 中关于 `owner` 和 `addr` 映射：
  - 如果该中介未被映射，函数会返回 `RetBrokerInterceptor None`，其对应于合约中的返回值 `(false, 0)`；
  - 如果该中介已被映射，函数会返回 `RetBrokerInterceptor (broker_interceptor broker_info)`，其对应于合约中的返回值 `(true, interceptor)`。

`getAllBrokers(address owner, uint start, uint count)` 可以查询调用者的多个中介地址和对应的 Interceptor 地址。该函数尚未被形式化建模，由人工审计完成。

2.10 OrderBook

`OrderBook` 提供了一个公开的订单账簿，订单所有者和中介可以向该合约提交订单，所有人可以通过订单哈希查询 `OrderBook` 中记录的订单信息。

- 形式化模型涉及的合约文件
  - `contracts/impl/OrderBook.sol`
- 相关的 Coq 文件
  - `Models/{OrderBook, States, Events, Messages}.v`

2.10.1 子状态

`OrderBook` 的子状态被表示为 `WorldState` 中的 `wst_order_book_state`，由 Coq 文件 `States.v` 中的 `OrderBookState` 定义。

- `ob_orders` 定义为从订单哈希到订单信息 `OrderElem`（参见 Coq 文件 `States.v`）的映射，用于存储所有已记录的订单信息，对应于 `OrderBook` 的 `orders` 变量。

2.10.2 形式化规范

`submitOrder(bytes32[] dataArray)` 由订单所有者或订单中介调用，用于提交订单到订单账簿。其规范 `submitOrder_spec` 中：

- byte array 参数通过其反序列化后的语义描述。
- `fspec_require` 要求函数执行前
  - 调用者必须是订单的所有者或者中介，
  - `ob_orders` 中不包含该订单信息，即该订单未在账簿中记录。
- `fspec_trans` 要求函数成功执行后
  - `ob_orders` 中将该订单哈希映射到该订单信息，
  - 返回该订单的哈希。
- `fspec_events` 要求函数成功执行后须产生 `OrderSubmitted`事件，指明该订单的哈希。

`getOrderData(address hash)` 通过订单哈希查询订单信息。在其规范 `getOrderData_spec` 中：

- `fspec_trans` 查询 `ob_orders` 中是否包含 `hash` 的映射：

    - 如果该哈希被映射到对应的订单信息，则返回 `RetOrder (Some ord)`，对应于合约中返回包含订单信息的非空 byte array 的情况；
    - 如果该哈希没有被映射，则返回 `RetOrder None`，对应于合约中返回空 byte array 的情况。

2.11 OrderRegistry

`OrderRegistry` 提供接口，允许订单中介向 LPSC 注册订单的哈希。如果订单的哈希已经被注册，则 `RingSubmitter` 可以省略校验该订单签名的步骤。

- 形式化模型涉及的合约文件
    - `contract/impl/OrderRegistry.sol`
- 相关的 Coq 文件
    - `Models/{OrderRegistry, States, Events, Messages}.v`

2.11.1 子状态

`OrderRegistry` 的子状态被表示为 `WorldState` 中的 `wst_order_registry_state`，由 Coq 文件 `States.v` 中的 `OrderRegistryState` 定义。

- `order_registry_hashMap` 定义为从中介地址和订单地址到布尔值的映射，表示某订单是否已被某中介注册，用于表示合约中的 `hashMap` 变量。

2.11.2 形式化规范

`isOrderHashRegistered(address owner, bytes32 hash)` 可以被任何人调用，检查中介 `owner` 的订单 `hash` 是否已被注册。其规范 `isOrderHashRegistered_spec` 中：

- `fspec_trans` 要求函数成功执行后，返回值是 `order_registry_hashMap` 将 `owner` 和 `hash` 映射到的布尔值。

`registerOrderHash(bytes32 orderHash)` 被订单中介调用，用于注册其订单。其规范 `registerOrderHash_spec` 中：

- `fspec_trans` 要求函数成功执行后 `order_registry_hashMap` 将调用者和订单哈希 `hash` 映射到 `true`；
- `fspec_events` 要求函数成功执行后产生 `OrderRegistered` 事件，指明调用者及其注册的订单的哈希。

3. 性质

通过证明形式化模型满足相应的性质是检查合约实现能否具备某些良好的行为的一种手段。只要形式化模型和合约实现保持一致，对某个性质证明的失败往往意味着合约实现中可能存在问题。

本节余下部分介绍审计过程中证明的性质。

3.1 与 suspend 和 kill 相关的性质

`TradeDelegate` 提供公共函数 `suspend()` 和 `kill()` 供授权用户暂停和终止 LPSC 成交和取消订单。Coq 文件 `Props/ControlProps.v` 中通过以下定理证明了它们的一些相关性质。

- 定理 `no_further_LPSC_transaction_once_suspended`

  `suspend()` 成功执行后，如果 `resume()` 没有被成功调用过，那么对以下可能转移 token 和取消订单的 LPSC 函数的调用都会失败：

  - `RingSubmitter::submitRings()`
  - all `cancel*()` functions in `RingCanceller`
  - `TradeDelegate::batchTransfer()`
  - `TradeDelegate::batchUpdateFilled()`
  - `TradeDelegate::setCancelled()` and all `set*Cutoffs*()` functions in `TradeDelegate`

  即 LPSC 成交订单和取消订单的功能无法进行。

- 定理 `no_further_LPSC_transaction_once_killed`

  `kill()` 成功执行后，无论 `resume()` 是否被调用过，对以下所有可能转移 token 和取消订单的 LPSC 函数的调用都会失败：

  - `RingSubmitter::submitRings()`
  - all `cancel*()` functions in `RingCanceller`
  - `TradeDelegate::batchTransfer()`
  - `TradeDelegate::batchUpdateFilled()`
  - `TradeDelegate::setCancelled()` and all `set*Cutoffs*()` functions in `TradeDelegate`

  即 LPSC 成交订单和取消订单的功能无法进行。

- 定理 `LPSC_cannot_be_resumed_once_killed`

`kill()` 被成功执行后，对 `resume()` 调用不会成功，即被终止的 `TradeDelegate` 合约无法被恢复。

3.2 与特权用户相关的性质

LPSC 中的一些关键操作只能由特权用户进行操作。Coq 文件 `Props/ControlProps.v` 中通过以下定理证明了相关性质。

- 定理 `only_owner_is_able_to_control_LPSC`

  只有 `TradeDelegate` 合约的所有者才能成功调用 `suspend()`、`resume()` 和 `kill()`。结合 3.1 节中的定理，我们可以总结出，只有 `TradeDelegate` 的合约可以暂停、恢复和终止 LPSC 成交和取消订单。

- 定理 `only_authorized_contracts_are_able_to_fill_or_cancel_orders`

  `TradeDelegate` 中与转移 token 和取消订单相关的函数只能被授权用户调用成功。

3.3 与合法环路相关的性质

LPSC 对能够成功交易的环路的合法性有一定的限制。Coq 文件 `Props/RingSubmitterProps.v` 和 `Props/FilledRingProps.v` 中通过以下定理证明了相关性质。

- 定理 `no_sub_rings`

  如果通过 `submitRings()` 提交的环路中包含子环路 (即环路中的多个订单在卖同一种 token)， `submitRings()` 不会成交这些环路。

- 定理 `no_cancelled_orders`

  如果通过 `submitRings()` 提交的环路中包含已被取消的订单，`submitRings()` 不会成交这些环路。

- 定理 `no_token_mismatch_orders`

  如果通过 `submitRings()` 提交的环路中存在相邻的订单之间买卖的 token 不一致的情况，`submitRings()` 不会成交这些环路。

- Coq 文件 `Props/FilledRingProps.v` 中的定理 `soundness` 和 `completeness`

  在一个简化的没有考虑交易费的计算和舍入误差的 `submitRings()` 算法中，当且仅当环路中所有订单卖出量和买入量比值的乘积大于等于 1 时，`submitRing()` 能计算出成交时环路中订单的买入和卖出金额，且每个订单成交时的价格不低于原本的价格。

3.4 与订单取消相关的性质

RingCanceller 提供了一系列取消订单的公共函数。Coq 文件 Props/RingCanceller.v 中通过以下定理证明所有这些函数的调用都不会影响其之前取消的订单的取消状态。

- 定理 cancelOrders_no_side_effect

  所有被 cancelOrders() 取消的订单，都不会被后续的 cancelOrders() 调用恢复。

- 定理 cancelAllOrdersForTradingPair_no_side_effect

  所有被 cancelAllOrdersForTradingPair() 取消的订单，都不会被后续的 cancelAllOrdersForTradingPair() 调用恢复。

- 定理 cancelAllOrders_no_side_effect

  所有被 cancelAllOrders() 取消的订单，都不会被后续的 cancelAllOrders() 调用恢复。

- 定理 cancelAllOrdersOfOwner_no_side_effect

  所有被 cancelAllOrdersOfOwner() 取消的订单，都不会被后续的 cancelAllOrdersOfOwner() 调用恢复。

- 定理 cancelAllOrdersForTradingPairOfOwner_no_side_effect

  所有被 cancelAllOrdersForTradingPairOfOwner() 取消的订单，都不会被后续的 cancelAllOrdersForTradingPairOfOwner() 调用恢复。

3.5 与 withdraw 相关的性质

FeeHolder 提供了用于转移燃烧费和交易费的公共函数 withdrawBurned() 和 withdrawToken()。Coq 文件 Props/FeeHolderProps.v 中通过以下定理证明这些函数具备的部分性质。

- 定理 withdrawBurned_noauth

  如果调用者未被 TradeDelegate 授权，那么它无法成功调用 withdrawBurned()。

  注意 该性质并不表示燃烧费用只能由授权用户燃烧。未授权用户仍然可以通过 BurnManager 的 burn() 燃烧 FeeHolder 中以 LRC 存储的燃烧费。BurnManager 是一个授权的用户。

- 定理 withdrawBurned_auth

  如果被授权的调用者成功的调用了 withdrawBurned(token, amount)，那么以下三个结果必然同时出现：

  - FeeHolder 中记录的调用者的以代币 token 计算的燃烧费余额被减掉相应的数量 amount

    - 返回值为 `true`
    - 产生一个 `TokenWithdrawn` 事件，标明调用者、被转移的燃烧费的 `token` 以及被转走的数量 `amount`
- 定理 `withdrawToken_noauth`

  如果被授权的调用者成功的调用了 `withdrawToken(token, amount)`，那么以下三个结果必然同时出现：

    - `FeeHolder` 中记录的调用者的以代币 `token` 计算的交易费余额被减掉相应的数量 `amount`
    - 返回值为 `true`
    - 产生一个 `TokenWithdrawn` 事件，标明调用者、被转移的交易费的 `token` 以及被转走的数量 `amount`

3.6 与 burn 相关的性质

`BurnManager` 提供可以被任何人调用的，用于燃烧 `FeeHolder` 中以 LRC 存储的燃烧费的公共函数 `burn()`。Coq 文件 `Props/BurnManagerProps.v` 中通过以下定理证明它的一些性质。

- Theorem `BurnManager_decrease_balance_of_fee_holder`

  `burn()` 只会减少 `FeeHolder` 的燃烧费的 ERC20 代币余额，而不会影响其他用户的 ERC20 代币余额和 Allowance。

4. 问题详述

我们在审计过程中发现了如下问题。所有这些问题或者已经在最新版本中被修复，或者在 LPSC 之外被处理，或者在实际中几乎没有影响。

4.1 问题1 相邻订单购买/出售的代币类型可能不一致

- **问题描述**: 路印协议中，每个订单中包含要卖的代币 (以 `tokenS` 表示) 和要买的代币 (以 `tokenB` 表示)。每个订单中的 `tokenS` 和 `tokenB` 不一定一样。当 `RingSubmitter::submitRings()` 将当前的订单的 `tokenS` 转移给前一个订单时，需要确认前一个订单的 `tokenB` 和当前订单的 `tokenS` 是同一种 token，否则可能会转移错误的代币。审计代码中的 `submitRings()` 遗漏了这个检查。
- **可能解决方法**: 在 `submitRings()` 中添加相应检查
- **目前状态**: 已修复
- **严重级别**: 高
- **其它** 该问题是在比对白皮书检查 `submitRings()` 的形式化规范 (参见 2.3 节) 时发现的。基于被审计代码构建的 `submitRings()` 的形式化规范的各个子规范，特别是包含环路和订单合法性检查的子规范中，未对相邻订单的买卖 token 的类型做限制，这一点与白皮书上的示例不一致。

4.2 问题2 withdraw() 没有正确处理调用外部合约调用失败的情况

- **问题描述**: `FeeHolder` 的公开接口函数 `withdrawBurned()` 和 `withdrawToken()` 调用该合约内部函数 `withdraw()` 完成实际的工作。审计代码中的 `withdraw()` 进行如下步骤的操作。

  1. 根据要提取的代币的类型和数量更新 `FeeHolder` 内记录的费用余额。
  2. 调用外部代币合约进行实际的费用转移。
  3. 如果上述外部合约调用成功，返回 `true`; 否则，返回 `false`。

  第 3 步在外部合约调用返回 `false` 的情况下未回滚第 1 步中对本合约状态的修改。在这种情况下，`withdrawBurned()` 和 `withdrawToken()` 的调用者将无法获得这部分转移失败的 代币，因为 `FeeHolder` 合约根据其状态变量认为这部分费用已经被取走。
- **可能解决方法**: 修改 `withdraw()` 使其在外部合约调用失败时 revert
- **目前状态**: 已修复
- **严重级别**: 高

- **其它**: 该问题是在试图证明定理 `withdrawToken_noauth` (参见 3.5 节) 时发现的。

  审计代码中的 `withdrawToken()` 对应的形式化规范的 `fspec_trans` 中包含一个分支，在该分支下 `withdrawToken()` 返回 `false`，并且 `FeeHolder` 的结束状态对应于上述第 1 步之后的状态。

  基于审计代码中的 `withdrawToken()` 定义的定理 `withdrawToken_noauth` 中要求证明结论之一是：在 `withdrawToken()` 返回 `false` 的情况下，`FeeHolder` 的状态与调用 `withdrawToken()` 之前的状态一样。由上述的 `withdrawToken()` 的形式化规范明显会推导出该结论的一个反例。

4.3 问题3 未充分检查多个 all-or-none 订单存在于多个环路中的情况

- **问题描述**: 如果一个 all-or-none 订单无法被完成成交，LPSC 应该跳过这个订单。`RingSubmitter::submitRings()` 的确在计算完所有订单的成交量和费用后，会遍历所有的订单，检查 all-or-none 订单是否被完全成交；包含未被完全成交的 all-or-none 订单的环路都不会被成交。

  但是，该检查无法发现如下的部分成交的情况。

  - 一个 all-or-none 订单 `Order0` 首先被检查，并且其可以被多个环路完全成交
  - 另一个包含 `Order0` 的环路还包含一个无法完全成交的 all-or-none 订单 `Order1`
  - `Order1` 在之后的检查中会被发现无法完全成交

  因为包含 `Order1` 的环路会被跳过，所以 `Order0` 也会变成无法完全成交。但是前述检查没有考虑环路，所以无法发现这种情况，最后会部分成交 all-or-none 订单 `Order0`。
- **可能解决方法**: 在遍历检查所有的订单时，如果发现被部分成交的订单时，取消同一环路中的所有订单在该环路中的成交量，然后重新遍历检查所有的订单。
- **目前状态**: 已修复
- **严重级别**: 中
- **其它**: 该问题是在检查由 `submitRings` 的实现抽象出的不考虑交易费的简化算法时发现的。

4.4 问题4 订单所有者或中介可能对环路矿工进行 GAS 攻击

- **问题描述**: 一个恶意的订单所有者或者中介可能会在环路矿工检查完它的代币余额后和向 LPSC 提交订单前，将代币账户中余额转走，从而可能导致环路矿工提交的所有包含该订单的环路全部交易失败，浪费环路矿工的 GAS。
- **目前状态**: Loopring 开发团队回复，通常环路矿工会在检查完余额后迅速提交订单，所以可以被利用的攻击窗口非常短。因此，该类 GAS 攻击在现实中很难实现。

SECBIT 团队认为这个解释是合理的。

- **严重级别**: 低

4.5 问题5 恶意代币合约会造成高 GAS 消耗

- **问题描述**: LPSC 在调用外部代币合约时 (例如调用 ERC20 合约的 `transfer`、`transferFrom`、`allowance` 和 `balanceOf` 等函数) 未设置 GAS limit。恶意或者被攻击的代币合约可能会包含高 GAS 消耗的代码。如果提交给 LPSC 的订单中包含这些代币，LPSC 的调用者 (例如调用 `submitRings()` 的 ringminer，调用 `withdrawBurned()` 的 `BurnManager` 合约等) 可能会消耗掉大量的 GAS。
- **目前状态**: Loopring 开发团队回复，这类不正常的代币由 Loopring 生态中的 Relay 负责检查和过滤。
- **严重级别**: 低

4.6 问题6 舍入误差导致成交金额计算不准确

- **问题描述**: Solidity 中的整型数字的除法存在舍入误差，可能会导致除法结果偏小。对于如下的通过 `submitRings()` 提交的包含两个订单的环路:
  - `Order0`: 卖 1 token A，买 10 token B，该订单能够花费的 token A 大于 1
  - `Order1`: 卖 10 token B，买 1 token A，该订单能够花费的 token B 只有 5

  在调用 `setMaxFillAmounts()` 处理上述环路后，这两个订单的成交量变成:
  - `Order0`: 卖出 0 token A, 买入 5 token B
  - `Order1`: 卖出 5 token B，买入 0 token A

  这样，如果没有任何费率，在最终成交后，`Order1` 会在卖出 5 token B 的情况下，没有买到任何 token A。
- **目前状态**: Loopring 开发团队回复，上述例子中所有数字的单位实际是 `wei`。考虑到实际发行的代币的 decimal 通常都远大于 1 (通常 1 token $\geq 10^{18}$ wei)，上述情况造成的损失实际中可以忽略。
- **严重级别**: 低

5. 最佳实践

LPSC 的开发质量很高，其中如下最佳实践尤为重要。

5.1 GAS 优化

LPSC 中的订单和环路的数据结构 `Order` 和 `Ring` 中包含较多的内容，但是 LPSC 使用了多种优化以降低 GAS 消耗。

- 如果一个 LPSC 的公开函数需要传入订单或者环路的信息，那么它的参数中只包含的确需要的 `Order` 或者 `Ring` 中的域，并且把这些域打包成 byte 数组，以减小需要的 calldata 的大小，从而节省用户的 GAS。
- 在合约内部，总是在 memory 上存储大数据结构，并且尽可能的使用它们的引用。
- 用汇编实现一些 GAS 消耗比较大的操作，避免编译器生成冗余和低效的字节码。

5.2 使用 SafeMath

整数运算的上溢和下溢是以太坊智能合约安全漏洞的一个主要来源。LPSC 大量使用 SafeMath 包装合约中的整数算术运算，对于缓解潜在的未发现的溢出漏洞尤为有用。

5.3 安全的外部合约调用

LPSC 需要调用外部的代币合约和订单中介的中继合约（Broker Interceptor），但是这些合约不属于 LPSC，也不能被 LPSC 信任。LPSC 使用底层的 `call` 调用 ERC20 合约的 `transfer` 和 `transferFrom` 以及中介中继合约的 `getAllowance()` 和 `onTokenSpent()`，并检查 `call` 的返回结果。这样，LPSC 才能有机会妥善处理失败的外部调用，而不是简单的 revert 所有的修改。这些安全 Wrapper 在合约文件 `contracts/lib/ERC20SafeTransfer.sol` 和 `contracts/impl/BrokerInterceptorProxy.sol` 中实现。

此外，在调用中介中继合约时，LPSC 还会设置 GAS limit，从而防止异常的中介中继合约消耗过多 GAS。

5.4 重入检查

`RingSubmitter::submitRings()` 和它的子调用中可能会调用外部合约。如果这些外部合约再调用 `submitRings()`，`submitRings()` 的完整性可能会遭到破坏。`submitRings()` 部署了相应的重入检查。它利用 `RingSubmitter` 的 Storage 变量 `ringIndex` 的最高位作为其是否正在执行的标记。如果在 `submitRings()` 入口处的检查中发现这一位已经被置位，即发生了重入，那么会发生 revert 回滚所有的操作。否则，该

位会被置位，以检查未来的重入。在 `submitRings()` 返回前，该位会被清零。

5.5 测试

Loopring 开发团队利用 Typescript 语言重写了 LPSC，并在其上做了大量的测试，并通过其检查合约是否按照预期工作。

5.6 完整的白皮书

Loopring 协议的白皮书详尽的解释了其生态、经济模型和大量技术细节，对于审计者、用户和投资者理解 Loopring 协议和合约帮助很大。

6. 结语

本次审计中发现了 3 处实现错误和 3 处可能风险，但是这些问题或者已经被修复，或者在 LPSC 之外已经解决，或者在实际中很难造成损失。此外，LPSC 遵循了大量的最佳实践。

总体而言，LPSC 具备很高的质量。

免责声明

SECBIT 智能合约安全审计从合约代码质量、合约逻辑设计和合约发行风险等方面对合约的正确性、安全性、可执行性进行审计，但不做任何和代码的适用性、商业模式和管理制度的适用性及其他与合约适用性相关的承诺。本报告为技术信息文件，不作为投资指导，也不为代币交易背书。

附录

漏洞风险级别介绍

风险级别	风险描述
高	可以严重损害合约完整性的缺陷，能够允许攻击者盗取以太币及Token，或者把以太币锁死在合约里等缺陷。
中	在一定限制条件下能够损害合约安全的缺陷，造成某些参与方利益损失的缺陷。
低	并未对合约安全造成实质损害的缺陷。

安比（SECBIT）实验室致力于参与共建共识、可信、有序的区块链经济体。

https://secbit.io
info@secbit.io
@secbit_io