Security Audit Report for Calamari Network

Authors:
Shumo Chu (shumo@manta.network)

Internal Security Audit

Date: Oct-24-2021

Contents

1 Audit Overview

This report summarizes the internal security review by Manta team on released Calamari runtime.

1.1 Goals and non-goals

The overaching goal of this audit is the review the security design and potential security risks of Calamari Network runtime, concretely:

1. Review overall architecture of Calamari (§2).
2. Review Calamari runtime and token configurations (§3).
3. Review defensive security measures in calamari runtime development (§4).
4. Review potential security risks in Calamari Network (§5)

Non-goals:

1. Provide formal specifications of Calamari's security model.
2. Provide formal guarantee of Calamari's runtime implementation.

1.2 Scope of the audit

The scope of the audit includes Calamari's runtime, the major functionalities include:

1. Calamari's collator runtime, including collator consensus
2. `KMA` token configuration
3. `KMA` token distribution
4. `KMA` token transaction

(Note `KMA` is Calamari's native token)

We assume a user will use Polkadot's hosted wallet (https://polkadot.js.org/apps or Polkadot browser extension (https://polkadot.js.org/

extension/) for transacting KMA tokens. The concrete codes that this audit covers is:

https://github.com/Manta-Network/Manta/tree/manta-pc/runtime/calamari

The release information of this audit:

Relase version	`manta v3.0.8-calamari.kusama`
Git SHA	`5420061200d12fd46531f08905d70b846b8d336f`
Release Date	October 29, 2021

2 Calamari Network Architecture

2.1 Relay-Chain/Parachain architecture

Calamari Network is parachain that is connected to Kusama relay-chain network [3]:

> *A parachain is an application-specific data structure that is globally coherent and validatable by the validators of the Relay Chain. They take their name from the concept of parallelized chains that run parallel to the Relay Chain. Most commonly, a parachain will take the form of a blockchain, but there is no specific need for them to be actual blockchains.* [7]

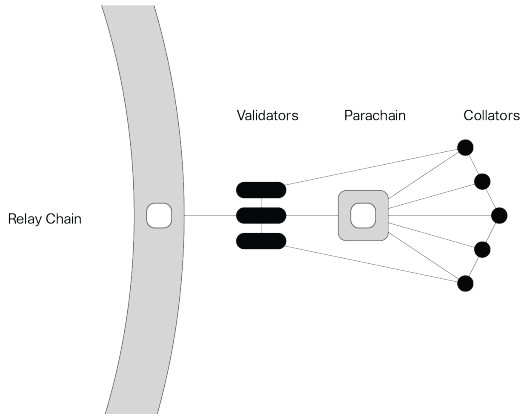

Figure 1: Relay-chain/Parachain Architecture, figure from https://wiki.polkadot.network/docs/learn-parachains

The parachain and relay chain architecture is shown in Figure 1. The relay-chain (a.k.a Kusama) consists of around 1,000 **validators** running a PBFT based Nominated Proof-of-Stake (NPoS) consensus algorithm [6]. The parachain (a.k.a Calamari) consists of 5 **collators** (as of Oct. 2021, will be expanded) that are in charge of producing parachain blocks. Since the parachain's validation logic is compiled to WebAssembly (WASM) and is getting executed in the relay chain for block finality, the parachain's collators only responsible for parachain's availability rather than security.

There are several key roles in this architecture:

1. **Kusama valiators**: responsible for confirming both Kusama relay-chain blocks and Calamari parachain blocks.

2. **Calamari colloators**: responsible for receiving extrinsic requests from the end user and producing Calamari parachain blocks.

3. **Calamari runtime**: Calamari's block validation logic, compiled to WASM and used both in Calamari collators for sanity check and Kusama validators for block validation for finality.

2.2 Calamari collators and runtime

Calamari collators and runtime is constructed using the Substrate blockchain framework [2]. Substrate is highly configurable thanks to its modular design. A collator client can consists of a set of components of the parachain developers' choices and sets parameters for these components. These components are called **pallets**. Substrate provides a few pallets that can be directly used by the parachain developers. At the same time, the parachain developers can develop and use their own pallets.

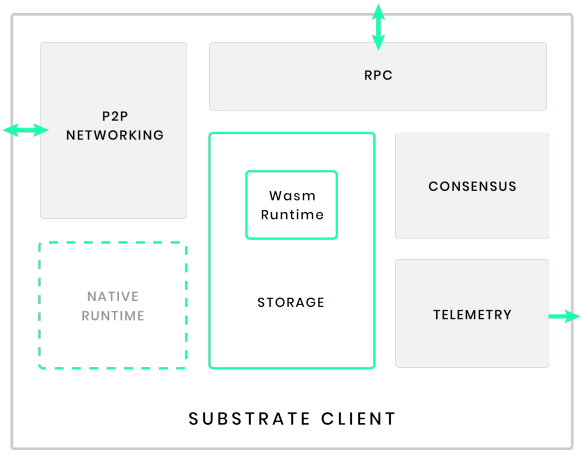

Figure 2: Substrate client architecture, figure from https://docs.substrate.io/v3/getting-started/architecture/

We demonstrate a typical substrate client construction in Figure 2. It consists of (but not limited to) the following components:

1. *Storage*: used to persist the evolving state of a Substrate blockchain. The blockchain network allows participants to reach trustless consensus about the state of storage.

2. *Runtime*: the logic that defines how blocks are processed, including state transition logic. In Substrate, runtime code is compiled to Wasm and becomes part of the blockchain's storage state.

3. *Peer-to-peer network*: the capabilities that allow the client to communicate with other network participants. Substrate uses the Rust implementation of the libp2p network stack to achieve this.

4. *Consensus*: the logic that allows network participants to agree on the state of the blockchain.

5. *RPC (remote procedure call)*: the capabilities that allow blockchain users to interact with the network. Substrate provides HTTP and WebSocket RPC servers.

6. *Telemetry*: client metrics that are exposed by the embedded Prometheus server.

3 Runtime and Token Configuration of Calamari

In this section, we first review the important facts about the runtime configurations of Calamari (§3.1), and then review the configuration and distribution of Calamari's native token `KMA` (§3.2).

3.1 Calamari Runtime Configurations

Since Calarmari runtime largely reuses Substrate[2] code base, one of the crucial aspect of the security is to configure the runtime properly. In this section, we first review the block configurations for Calamari in §3.1.1 and then review the pallets configurations of the runtime §3.1.2

3.1.1 Block Configurations

We list the important block configurations in Table 1.

Property	Value	Remark
Block time	12 sec	default value for parachain
Block size	5 MB	default value
Max weight per single extrisic	10% block weight	higher than 5% default value
Normal dispatch ratio	70%	lower than 75% default value
Block compute budget	0.5 sec computation	default value

Table 1: Important facts about blocks

We review their security implications as follow:

1. **Block time** and **Block size**: we use the default settings here. This is decided by the distribution of internet bandwidth and computational power of collators. We use the deployment proven default settings here.

2. **Max weight per single extrisinc**: there is a roughly 5% of the block weight is consumed by `on_initialize` handlers. We set it to a more conservative 10% here.

3. **Normal dispatch ratio**: this number defines how much of the block can be filled by "normal" (non-system) extrinsics. The default value is 75%, we set it to a more conservative 70% here.

4. **Block compute budget**: this value is by setting the `MAXIMUM_BLOCK_WEIGHT`. We set `MAXIMUM_BLOCK_WEIGHT` to `WEIGHT_PER_SECOND/2`, thus the it allows 0.5 sec compute budget for all the extrinsics if their weight is measured properly.

The corresponding Calamari source code for these block configurations:

```
/// We assume that ~5% of the block weight is consumed by `on_initialize` handlers. This is
/// used to limit the maximal weight of a single extrinsic.
pub const AVERAGE_ON_INITIALIZE_RATIO: Perbill = Perbill::from_percent(10);
/// We allow `Normal` extrinsics to fill up the block up to 75%, the rest can be used by
/// Operational  extrinsics.
pub const NORMAL_DISPATCH_RATIO: Perbill = Perbill::from_percent(70);

/// We allow for 0.5 seconds of compute with a 6 second average block time.
pub const MAXIMUM_BLOCK_WEIGHT: Weight = WEIGHT_PER_SECOND / 2;

pub RuntimeBlockLength: BlockLength =
        BlockLength::max_with_normal_ratio(5 * 1024 * 1024, NORMAL_DISPATCH_RATIO);
```

3.1.2 Pallets Configurations

Currently, all the pallets that Calamari contains is from the substrate code base (imported as source dependencies). The Calamari parachain runtime consists of the follows pallets (we review their functionalites below as well):

1. **frame_system**: `frame_system` pallet defines the core data types used in a Substrate runtime. It also provides several utility functions for other FRAME pallets. In addition, it manages the storage items for extrinsics data, indexes, event records, and digest items, among other things that support the execution of the current block.It also handles low-level tasks like depositing logs, basic set up and take down of temporary storage entries, and access to previous block hashes.

2. **cumulus_pallet_parachain_system**: `cumulus_pallet_parachain_system` is a base pallet for cumulus-based parachains. It handles low-level details of being a parachain: such as ingestion of the parachain validation data, ingestion of incoming downward and lateral messages and dispatching them, coordinating upgrades with the relay-chain and communication of parachain outputs.

3. **pallet_timestamp**: The Timestamp pallet allows the validators to set and validate a timestamp with each block.

4. **parachain_info**: Minimal pallet that injects a ParachainId into Runtime storage from.

5. **pallet_balances**: The pallet that handles accounts and balances of native token `KMA`. More details will be explained in §3.2.

6. **pallet_transaction_payment**: This pallet provides the basic logic needed to pay the absolute minimum amount needed for a transaction to be included. This includes base fee, weight fee, length fee, and tip.

7. **pallet_democracy**: The Democracy pallet handles the administration of general stakeholder voting.

8. **pallet_collective**: Collective system: Members of a set of account IDs can make their collective feelings known through dispatched calls from one of two specialized origins. We instantiate two instances of `pallet_collective`, one for Calamari council and one for Calamari Tech. Committee.

9. **pallet_membership**: Allows control of membership of a set of 'AccountId's, useful for managing membership of of a collective. A prime member may be set. We instantiate two instances of `pallet_membership`, one for Calamari council and one for Calamari Tech. Committee.

10. **pallet_authorship**: Authorship tracking for FRAME runtimes: this pallet tracks the current author of the block and recent uncles.

11. **pallet_collator_selection**: The Collator Selection pallet manages the collators of a parachain. Note that collation is **not** a secure activity

and this pallet does not implement any game-theoretic mechanisms to meet BFT safety assumptions of the chosen set.

12. **pallet_session**: The Session pallet allows collators to manage their session keys, provides a function for changing the session length, and handles session rotation.

13. **pallet_aura**: The Aura pallet extends Aura consensus by managing offline reporting.

14. **cumulus_pallet_aura_ext**: Cumulus extension pallet for Aura. This pallet extends the Substrate Aura pallet to make it compatible with parachains.

15. **pallet_scheduler**: A Pallet for scheduling dispatches.

16. **cumulus_pallet_xcmp_queue**: A pallet which uses the XCMP transport layer to handle both incoming and outgoing XCM message sending and dispatch, queuing, signalling and back-pressure.

17. **pallet_xcm**: Pallet to handle XCM messages.

18. **cumulus_pallet_xcm**: Pallet for stuff specific to parachains' usage of XCM.

19. **cumulus_pallet_dmp_queue**: Pallet implementing a message queue for downward messages from the relay-chain.

20. **pallet_utility**: A stateless pallet with helpers for dispatch management which does no re-authentication, e.g. sending batched transactions.

21. **pallet_multisig**: A pallet for doing multisig dispatch.

22. **pallet_sudo**: The Sudo pallet allows for a single account (called the "sudo key") to execute dispatchable functions that require a 'Root' call or designate a new account to replace them as the sudo key. Only one account can be the sudo key at a time.

The corresponding code in Calamari for parachain configurations:

```
// System support stuff.
System: frame_system::{Pallet, Call, Config, Storage, Event<T>} = 0,
ParachainSystem: cumulus_pallet_parachain_system::{
        Pallet, Call, Config, Storage, Inherent, Event<T>,
    ValidateUnsigned,} = 1,
Timestamp: pallet_timestamp::{Pallet, Call, Storage, Inherent} = 2,
ParachainInfo: parachain_info::{Pallet, Storage, Config} = 3,

// Monetary stuff.
Balances: pallet_balances::{Pallet, Call, Storage, Config<T>, Event<T>}
     = 10,
TransactionPayment: pallet_transaction_payment::{Pallet, Storage} = 11,

// Governance stuff.
Democracy: pallet_democracy::{Pallet, Call, Storage, Config<T>, Event<T
    >} = 14,
Council: pallet_collective::<Instance1>::{Pallet, Call, Storage, Origin
    <T>, Event<T>, Config<T>} = 15,
CouncilMembership: pallet_membership::<Instance1>::{Pallet, Call,
    Storage, Event<T>, Config<T>} = 16,
TechnicalCommittee: pallet_collective::<Instance2>::{Pallet, Call,
    Storage, Origin<T>, Event<T>, Config<T>} = 17,
TechnicalMembership: pallet_membership::<Instance2>::{Pallet, Call,
    Storage, Event<T>, Config<T>} = 18,

// Collator support. the order of these 4 are important and shall not
    change.
Authorship: pallet_authorship::{Pallet, Call, Storage} = 20,
CollatorSelection: pallet_collator_selection::{Pallet, Call, Storage,
    Event<T>, Config<T>} = 21,
Session: pallet_session::{Pallet, Call, Storage, Event, Config<T>} =
    22,
Aura: pallet_aura::{Pallet, Storage, Config<T>} = 23,
AuraExt: cumulus_pallet_aura_ext::{Pallet, Storage, Config} = 24,

// System scheduler.
Scheduler: pallet_scheduler::{Pallet, Call, Storage, Event<T>} = 29,

// XCM helpers.
XcmpQueue: cumulus_pallet_xcmp_queue::{Pallet, Call, Storage, Event<T>}
     = 30,
PolkadotXcm: pallet_xcm::{Pallet, Call, Event<T>, Origin} = 31,
CumulusXcm: cumulus_pallet_xcm::{Pallet, Event<T>, Origin} = 32,
DmpQueue: cumulus_pallet_dmp_queue::{Pallet, Call, Storage, Event<T>} =
     33,
```

```
34
35 // Handy utilities.
36 Utility: pallet_utility::{Pallet, Call, Event} = 40,
37 Multisig: pallet_multisig::{Pallet, Call, Storage, Event<T>} = 41,
38 Sudo: pallet_sudo::{Pallet, Call, Config<T>, Storage, Event<T>} = 42,
```

3.1.3 Filters

For safety, Calamari currently only allows a subset of extrinsic calls being made. The current set of allowed extrinsic calls are from:

1. **ParachinSystem, Authorship**: Essential parachain system calls.
2. **Sudo**: Sudo access.
3. **Multisig**: Multisig is required since sudo account is configured as an multisig.
4. **Democracy, Council, CouncilMembership, TechnicalCommittee, TechincalCommitteeMembership**: Essentical calls for governance.

3.2 `KMA` Token Configuration and Distribution

`KMA` is Calamari's native currency, `KMA` has following major utilities:

1. **Existential Deposit**. As Calamari's native currency, each single account need to keep a minimal amount of `KMA` as existential deposit (more details can be found at §3.2.1.
2. **Extrinsic Fee Payment**. `KMA` is used to pay the fee for every extrinsic, and as a mechanism to prevent DDoS attack to Calamari.
3. **Commission Fee**. Commission fee for private payment/private swap.
4. **On-Chain Governance**. `KMA` holders can use the token to join the on-chain governance, voting for the governance proposal.

Property	Value	Remark
Token Name	Calamari	
Token Symbol	`KMA`	
Decimal	12	1 KMA = 10^{12} units
Existential Deposit	0.1 `KMA`	
Total Initial Supply	10,000,000,000 `KMA`	

Table 2: Important facts about `KMA`

3.2.1 Supply, Existential Deposit, and Extrinsic Fee

We show the important facts about `KMA` in Table 2. `KMA` has a total supply of 10,000,000,000 and decimal 12. Figure 3 shows the token allocation of `KMA`.

The **Existential Deposit** is the minimal amount that a user need to hold in order to make the account active. For example, if Bob creates an account with zero balance, by default, the account would not appear in the ledger state. Now, if Alice sends Bob no less than 0.1 `KMA`, then Bob balance will be added to `pallet_balances` as part of ledger state. However, if Alice sends Bob less than 0.1 `KMA`, the transaction would be rejected since if ledger accepts the transaction, Bob will have a `KMA` balance less than 0.1, which will violate the *existential deposit* requirement. Similarly, Alice cannot transfer out her balance such that she has less than 0.1 `KMA` left. She can either choose leave more than 0.1 `KMA` or sending all the remaining balances and remove the account from the ledger state.

The extrinsic fee is defined by the **weight** of each extrinsic and is paid in `KMA`. For calamari, each block has a 0.5 second block execution budget, and this budget is divided into $500,000,000,000$ weights (10^12 weights per second). In each calamari release, the extrinsic's runtime on the collator is measured on a standard AWS instance (`r5ad.2xlarge`).

3.2.2 Governance, Super users and Treasury Accounts

At the time when Calamari is launched, a sudo account is configured to perform runtime upgrade. This sudo account is a 3/5 multisig account to ensure token safety. In the planed runtime upgrade on November 6th, with the launch of governance, sudo account will be removed.

Before the distribution of the crowdloan token rewards, all the `KMA`tokens

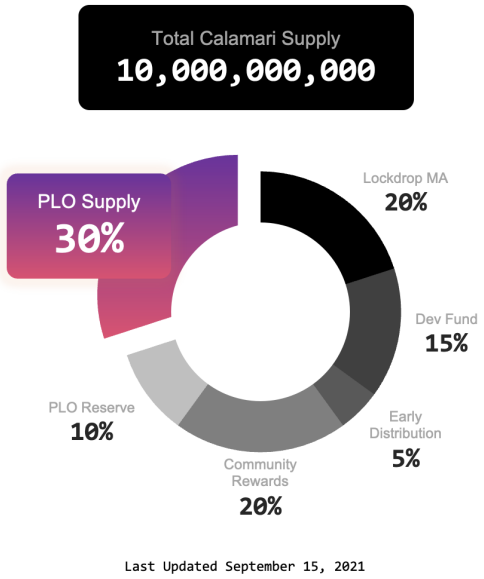

Figure 3: KMA token allocation

are hold in 3 treasury account (each of them holds 1/3 of the total supply). Each treasury account is a 3/5 multisig account as well.

4 Security Design and Defensive Measures

4.1 Shared Security with Kusama

Shared security, sometimes referred in documentation as pooled security, is one of the unique value propositions for chains considering to become a parachain and join the Kusama network. On a high level, shared security means that all parachains that are connected to the Kusama Relay Chain by leasing a parachain slot will benefit from the economic security provided by the Relay Chain validators.

The notion of shared security is different from interchain protocols that build on an architecture of bridges. For bridge protocols, each chain is considered sovereign and must maintain its own validator set and economic security. One concern in these protocols is on the point of scalability of security. For example, one suggestion to scale blockchains is that of scale by altcoins, which suggests that transaction volumes will filter down to lower market cap altcoins as the bigger ones fill their blocks. A major flaw in this idea is that the lower market cap coins will have less economic security attached, and be easier to attack. A real life example of a 51% attack occurred recently (Ethereum Classic attack on January 10), in which an unknown attacker double spent 219,500 ETC (1.1 million USD). This was followed by two more 51% attacks on ETC.

Polkadot/Kusama overcomes security scalability concerns since it gravitates all the economic incentives to the Relay Chain and allows the parachains to tap into stronger guarantees at genesis. Sovereign chains must expend much more effort to grow the value of their coin so that it is sufficiently secure against well-funded attackers.

Example: Let's compare the standard sovereign security model that exists on current proof-of-work (PoW) chains to that of the shared security of Polkadot. Chains that are secured by their own security model like Bitcoin, Zcash, Ethereum, and their derivatives all must bootstrap their own independent network of miners and maintain a competitive portion of honest hashing power. Since mining is becoming a larger industry that increasingly centralizes on key players, it is becoming more real that a single actor may control enough hash power to attack a chain.

This means that Calamari doesn't have to maintain a secure amount of hash power on their networks could potentially be attacked by a large mining cartel at the simple whim of redirecting its hash power away from Bitcoin

and toward a new and less secure chain. 51% attacks are viable today with attacks having been reported on Ethereum Classic (see above), Verge, Bitcoin Gold, and other cryptocurrencies.

On Polkadot/Kusama, this disparity between chain security will not be present. When a Calamari connects to Kusama, the Relay Chain validator set become the securers of Calamari's state transitions. Calamari will only have the overhead of needing to run a few collator nodes to keep the validators informed with the latest state transitions and proofs/witness. Validators will then check these for the parachains to which they are assigned. In this way, new parachains instantly benefit from the overall security of Polkadot even if they have just been launched.

4.2 Defensive Measures in Calamari

Using mature pallets from substrate code base. For the most important building blocks of the runtime (a.k.a pallets), Calamari are not reinventing the wheel. Instead, Calamari uses the mature code base from substrate where Polkadot/Kusama also uses. This means these pallets' code is third party audited and has been battle tested.

Using mature wallets infrastructure. Calamari's users are using mature web wallet and browser extension wallets provided by polkadot.js (`https://polkadot.js.org/`). Polkadot.js wallets provide the following guarantees:

1. All the account secrets is maintained and encrypted in the local storage. It provide the `keyring` interface for the Dapp to sign all the extrinsics without directly using the account secrets.

2. DApp's call to `keyring` interface is password protected. A user need to type to local password in order to sign a extrinsics. With the exception that a user could set 15 mins grace period after the password is entered.

5 Potential Risks and Solutions

In this section, we outline the potential security risks of Calamari and propose practical solution for each of them.

5.1 Mis-configuration of Pallets

Despite most pallets in Calamari runtime directly reuses source code from substrate, a industry leading and battle proven blockchain framework. Misconfiguri g these pallets could still need to invulnerabilities. For example, in a recent hack, Acala loses 10,000 KSM due to a misconfiguration of XCM pallet [8].

This happened due to the compounding of three factors on the Karura chain:

1. The misconfiguration of the XCM pallet.
2. The usage of older, flawed, code.
3. The utilization of Reserve asset transfers.

The combination of these factors led to the possibility of the attacker being able to execute a message which could make a withdrawal from Karura's reserve account on the Relay-chain.

Due to the complexity of substrate framework, the configuration error is a potential security risk for Calamari network.

solution:

When introducing unaudited code, Always Verify Correctness. This is really just common sense in an industry or movement that prides itself on reducing the need for trust and authorities. No code author or company is good enough to be blindly trusted. Linus's law, "given enough eyeballs, all bugs are shallow", doesn't work if nobody studies the code except the original authors themselves.

Introduce more stringent review requirements for highly sensitive code. A basic code review process is clearly not enough to avoid code changes that remove guard rails erroneously. Parity will introduce the idea of locked files and locked lines which are small pieces of highly sensitive code which require enhanced review by multiple experts in order to be changed in our repository. The guardrail which was incorrectly removed would be just such

a locked line. We will of course be making this CI software available for all teams to use.

Introduce contingency mechanisms. One of the reasons that Karura was quickly able to resolve this issue was due to it containing a special pallet called "Transaction Pause". This enabled the Karura governance body to halt most types of user transactions including the misconfigured one which enabled the attack. A similar "safe-mode" feature in should be implemented in Calamari as well, making such a contingency mechanism available for all parachains to use. In the case of a possible attack, this will be the first line of defence and will buy time for the governance process to determine the nature of the exploit (if any) and possible remedial effects including the repatriation of funds and upgrading.

5.2 Risk involved in Zero-Knowledge-Proof Cryptography

Calamari's security depends on the soundness, completeness and zero-knowledge property of the underlying zero-knowledge proof systems. Currently, calamari is using Arkwork(https://github.com/arkworks-rs)'s implmentation of Groth16 [4] scheme.

Almost all efficient zero-knowledge proof systems require pairing as the primitive (Groth16 included). Pairings are special maps defined using elliptic curves and it can be applied to construct several cryptographic protocols such as identity-based encryption, attribute-based encryption, and so on. At CRYPTO 2016, Kim and Barbulescu proposed an efficient number field sieve algorithm named exTNFS for the discrete logarithm problem in a finite field [5]. Several types of pairing-friendly curves such as Barreto-Naehrig curves are affected by the attack. In particular, a Barreto-Naehrig curve with a 254-bit characteristic was adopted by a lot of cryptographic libraries as a parameter of 128-bit security, however, it ensures no more than the 100-bit security level due to the effect of the attack.

Solution: Unlike Ethereum, Calamari chooses a pair-friendly curve (BLS12-381, https://electriccoin.co/blog/new-snark-curve/) that doesn't affect by the exTNFS attack. We will closely follow the on-going IETF draft (https://www.ietf.org/id/draft-irtf-cfrg-pairing-friendly-curves-10.html) to update the construction of the underlying zero-knowledge proof system.

5.3 MariPay's Concrete Privacy

Calamari's shielded payment's privacy is formally specified and with reduction based security proof [1]. However, in the concrete implementation, there might still be privacy leakage that is caused by implementation due to the complexity of the protocol. More specifically:

1. Runtime implmentation: the MariPay's runtime implemenation need to be faithful to the protocol without introducing any further leackage
2. Wallet/Client implementation: MariPay's wallet protocol (which is not covered by the paper) cannot leak user's information.

Solution:

1. Perform an internal (CFL styled) penetration test to find security and privacy vulerabilities.
2. Before MariPay goes online, conduct a security audit from a 3rd party audit to verify the security and privacy claims.
3. Design and implemenent fuzzers to perform randomized fuzzing test for the protocol.

References

[1] Shumo Chu, Yu Xia, and Zhenfei Zhang. Manta: a plug and play private defi stack. Cryptology ePrint Archive, Report 2021/743, 2021. https://ia.cr/2021/743.

[2] Substrate Developers. Substrade developer home. https://docs.substrate.io/, 2021.

[3] Web3 Foundation. Kusama, polkadot's canary network. https://kusama.network/, 2021.

[4] Jens Groth. On the size of pairing-based non-interactive arguments. *IACR Cryptol. ePrint Arch.*, page 260, 2016.

[5] Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In *CRYPTO (1)*, volume 9814 of *Lecture Notes in Computer Science*, pages 543–571. Springer, 2016.

[6] Kusama Network. Kusama guide: : Learn consensus. https://guide.kusama.network/docs/learn-consensus, 2021.

[7] Polkadot Network. Polkadot wiki: Learn parachains. https://wiki.polkadot.network/docs/learn-parachains, 2021.

[8] Parity Technology. Kusama's governance thwarts would-be attacker! https://medium.com/kusama-network/kusamas-governance-thwarts-would-be-attacker-9023180f6fb, 2021.