Polymesh Baseline Security Assurance

Threat model and hacking assessment report

V2.0, March 26, 2021

Stephan Zeisberg stephan@srlabs.de

Mostafa Sattari mostafa@srlabs.de

Vincent Ulitzsch vincent@srlabs.de

Abstract. This study describes the results of a thorough, independent baseline security assurance audit of the Polymesh blockchain platform performed by Security Research Labs. In the course of this study, Polymath provided full access to relevant documentation and supported the research team effectively.

The protection of Polymesh was independently verified to assure that existing hacking risks are understood and minimized.

The research team identified several issues ranging from low to critical risk were identified by the examiners and Polymath addressed them quickly. Most of the issues are no longer present in the latest development version of Polymesh.

To further improve the security of the Polymesh network, we recommend best practices around handling numeric bounds and dependency patching as well as leveraging continuous fuzz-testing during development. In addition, we recommend to only use custom cryptography schemes when absolutely necessary.

- Confidential, 14 pages -

Content

1 Motivation and scope

This review assesses the Polymesh blockchain system's existing protections against a variety of **likely hacking scenarios** and **points out the most relevant weaknesses**, all with the goal of improving the protection capabilities of the blockchain system.

Data stored on future Polymesh nodes poses an attractive theft target. Threats that could compromise systems using Polymesh go far beyond the theft of value tokens. Notable hacking scenarios include the potential to undermine trust in the blockchain system by 'short-selling', 'double-spending', or artificially driving up the value of supply by 'locking up' tokens.

This report details the baseline security assurance results with the aim of creating transparency in three steps:

Threat Model. The threat model is considered in terms of *hacking incentives*, i.e. the motivations to achieve the goals of breaching the integrity, confidentiality, or availability of nodes in future Polymesh systems. For each hacking incentive, we postulate *hacking scenarios,* by which these goals could be reached. The threat model provides guidance for the design, implementation, and security testing of Polymesh.

Security design coverage check. Next, the Polymesh design was reviewed for coverage against relevant hacking scenarios. For each scenario, the following two aspects were investigated:

a. **Coverage**. Is each potential security vulnerability sufficiently covered?

b. **Underlying assumptions**. Which assumptions must hold true for the design to effectively reach the desired security goal?

Implementation baseline check. As a third step, the current Polymesh implementation was tested for openings whereby any of the defined hacking scenarios could be executed.

Polymesh is built upon Substrate, a blockchain development framework. Both Polymesh and Substrate are written in Rust, a memory safe programming language. Mainly, Substrate works with three technologies: a WebAssembly (WASM) based runtime, decentralized communication via libp2p, GRANDPA finality gadget and the BABE block production engine.

The Polymesh runtime consists of multiple modules compiled into a WASM Binary Large Object (blob) that is stored on-chain. Nodes execute the runtime code either natively or will execute the on-chain WASM blob. These runtime modules (e.g. asset, identity, balances) are implemented in the Polymesh source[1] as well as in the Substrate framework.

Polymath shared an overview containing the current state of the runtime modules used by Polymesh and its audit priority. The priority and the in-scope components are reflected in Table 1.

[1] https://github.com/PolymathNetwork/Polymesh

Repository	Priority	Component(s)
Polymesh	High	Bridge, Asset, Identity, Multisig, Polymesh Improvement Proposals, Settlement, Runtime, Cryptography[2]
	Medium	Balances, Committee, Common, Compliance Manager, Corporate Actions, Group, Permissions, Portfolio, Staking, Security Token Offering, Transaction Payment, Treasury, Utility, Weights
	Low	Smart Contract, Confidential, Primitives, Protocol Fees, Statistics, Im-Online, RPC, Node RPC, Primitives Derive

Table 1. In-scope Polymesh components with audit priority

2 Methodology

To be able to effectively review the Polymesh codebase, a threat-model driven code review strategy was employed. For each identified threat, hypothetical attacks that can be used to realize the threat were developed and mapped to their respective threat category as outlined in chapter 3.

Prioritizing by risk, the codebase was assessed for present protections against the respective threats and attacks as well as the vulnerabilities that make these attacks possible. For each threat, the auditor:

1. Identified the relevant parts of the codebase, for example, the relevant pallets.
2. Identified viable strategies for the code review. Manual code audits, fuzz-testing, and manual tests were performed where appropriate.
3. Ensured the code did not contain any vulnerabilities that could be used to execute the respective attacks, otherwise, ensure sufficient protection measures against specific attacks were present.
4. Immediately reported any vulnerability that was discovered to the development team along with suggestions around mitigations.

During the audit, we carried out a hybrid strategy utilizing a combination of code review and dynamic tests (e.g. fuzz-testing) to assess the security of the Polymesh codebase.

While fuzz-testing and dynamic tests establish a baseline assurance, the main focus of this audit was a manual code review of the Polymesh codebase to identify logic bugs, design flaws, and best practice deviations. We used the v1_mainnet[3] branch of the Polymesh repository as the basis for the review. The approach of the review was to trace the intended functionality of the runtime modules in scope and to assess whether an attacker can bypass/misuse/abuse these components or trigger

[2] The cryptography resides in a separate repository: (https://github.com/PolymathNetwork/cryptography)
[3] Commit: d7bfeecbebfe927725ebeb2a531610d59de19b96

unexpected behavior on the blockchain due to logic bugs or missing checks. Since the Polymesh codebase is entirely open source, it is realistic that a malicious actor would analyze the source code while preparing an attack.

Fuzz-testing is a technique to identify issues in code that handles untrusted input, which in Polymesh's case is mostly the functions implementing the extrinsics. (Note that the network part is handled by Substrate, which was not in scope for this review, but is built with a strong emphasis on security and where fuzz-testing is also used). Fuzz-testing works by taking some valid input for a method under test, applying a semi-random mutation to it, and then invoking the method under test again with this semi-valid input. Through repeating this process, fuzz-testing can unearth inputs that would cause a crash or other undefined behavior (e.g., integer overflows) in the method under test. The fuzz-testing methods written for this assessment utilized the test runtime Genesis configuration as well as mocked externalities to execute the fuzz-test effectively against the extrinsics in scope.

3 Threat modeling and attacks

The goal of the threat model framework is to be able to determine specific areas of risk in Polymesh's blockchain system. Familiarity with these risk areas can provide guidance for the design of the implementation stack, the actual implementation of the stack, as well as the security testing. This section introduces how risk is defined and provides an overview of the identified threat scenarios. The *Hacking Value*, categorized into *low*, *medium*, and *high*, takes into account the incentive of an attacker, as well as the effort required by an attacker to successfully execute the attack. The hacking value is calculated as:

$$Hacking\ Value = \frac{Incentive}{Effort}$$

While *incentive* describes what an attacker might gain from preforming an attack successfully, *effort* estimates the complexity of this same attack. The degrees of incentive and effort are defined as follows:

Incentive:

- Low: Attacks offer the hacker little to no gain from executing the threat.
- Medium: Attacks offer the hacker considerable gains from executing the threat.
- High: Attacks offer the hacker high gains by executing this threat.

Effort:

- Low: Attacks are easy to execute. They require neither elaborate technical knowledge nor considerable amounts of resources.
- Medium: Attacks are somewhat difficult to execute. They might require bypassing countermeasures, the use of expensive resources or a considerable amount of technical knowledge.

- High: Attacks are difficult to execute. The attacks might require in-depth technical knowledge, vast amounts of expensive resources, bypassing countermeasures, or any combination of these factors.

After applying the framework to the Polymesh system, different threat scenarios were identified. Table 2 provides a high-level overview of the threat model with identified example threat scenarios and attacks, as well as their respective hacking value and effort.

Security promise	Hacking value	Example threat scenarios	Hacking effort	Example attack ideas
Confiden-tiality	High	- Compromise ownership privacy (asset linkage) - Linking confidential identities to real entities	High	- Exploit a bug in the cryptography implementation/design
Integrity	High	- Governance Capture (e.g. interfering with approving and executing any PIP) - Front running - Abuse the bridge to mint tokens - Invest in assets with different identities without linking them	Medium	- Storing malicious runtime code on-chain - Exploit logic bug in bridge implementation to mint tokens - Exploit a bug in the cryptography implementation/design
Availability	High	- Validate malicious blocks to double spend tokens via adding malicious validators to the validator pool - Locking account to freeze access	Medium	- DoS validator nodes - Halt block production by spamming computationally expensive/wrongly weighted transactions - Transaction spamming - Exploit logic bug to crash nodes

Table 2. Threat scenario overview. The threats for Polymesh's blockchain were classified using the CIA security triad model, mapping threats to the areas: (1) Confidentiality, (2) Integrity, and (3) Availability.

4 Findings summary

We identified 12 issues - summarized in Table 3 - during our analysis of the runtime modules in scope in the Polymesh codebase that enable the attacks outlined above. In addition, we also reported some minor bugs and best practice deviations. In summary, one critical severity, 1 high severity, 8 medium severity issues and 2 low severity issues were found. Most of the vulnerabilities were already mitigated by the Polymath team, as detailed in this document. Polymath decided to accept the risk posed by the FIFO transaction scheme as explained in detail in chapter 4.1.7.

Issue description	Severity	Refere nce	Remediation
Exponential complexity of weight calculation functions in *vote_or_propose* call allows for DoS attack, potentially stalling the blockchain	Critical	[1]	Polymesh#861

Investor can generate multiple scope claims via incorrect computation of *SHA(SCOPE_DID, INVESTOR_UNIQUE_DID)*	High	[2]	Polymesh#962 cryptography#121
Integer underflow when validating CUSIP identifiers	Medium	[3]	Polymesh#867
Integer underflow when validating a LEI identifier	Medium	[4]	Polymesh#867
Integer overflow when validating a ISIN identifier	Medium	[5]	Polymesh#867
Missing tipping mechanism and basing the transaction priority solely on the fee enable an attacker to delay time-critical transactions	Medium	[6]	Polymesh#893 Polymesh#918
Integer overflow in weight calculation for *fn instantiate()*	Medium	[7]	Polymesh#890
Integer overflow when creating a fundraiser	Medium	[8]	Polymesh#889
Deposit for an extrinsic in *propose* extrinsic in *pallets/pips/src/lib.rs* does not take into account the length of the *url* parameter that is stored on-chain	Medium	[9]	Polymesh#959
Integer underflow via *remove_multisig_signers_via_creator*	Low	[10]	Polymesh#887
Proofs may be replayed for different claim types	Low	[11]	Not fixed[4]
The FIFO transaction schemes enables an attacker to delay "normal" extrinsics and introduces the risk of missing updates because of forking Substrate	none	[12]	Risk accepted

Table 3. Overview of identified issues

4.1 Detailed findings

4.1.1 Exponential complexity of weight calculation allows for DoS attack

The weight calculation function of the *vote_or_propose* extrinsic in *pallets/committee/src/lib.rs* has exponential complexity because they perform two calls to the *get_dispatch_info()* function [1]. An attacker can abuse the exponential complexity of *O(2^n)* to craft a nested extrinsic (that is still below the *MAX_EXTRINSIC_DEPTH* of 256), for which the weight computation is not feasible in limited time and thus cause a validator to miss its slot and fail at block production, potentially halting block production.

The same problem existed in other extrinsics in Substrate as well. To mitigate exponential complexity in other extrinsics, we suggest updating the Substrate version Polymesh is using to a version that includes PR 7849 [13], which fixes the respective problems in Substrate. Polymesh mitigated this issue in Polymesh#861 by forking the sudo pallet of Substrate repository, accepting the risks that forking Substrate will impose on their code base (refer to [12]).

[4] The current version of Polymesh is not vulnerable (more info at https://github.com/PolymathNetwork/polymesh-audit/issues/12#issuecomment-799464721).

4.1.2 Investor can generate multiple scope claims using an erroneous hash

The *add_investor_uniqueness_claim* extrinsic in *pallets/identity/src/lib.rs* accepts a SCOPE_ID via the *InvestorUniqueness* claim, which is calculated by the investor with the following formula:

SCOPE_ID = p_hash(SCOPE_DID, INVESTOR_UNIQUE_ID, SHA(SCOPE_DID, INVESTOR_UNIQUE_ID)).

The assumption is that this will always lead to the same *SCOPE_ID* for the same investor (*INVESTOR_UNIQUE_ID*) even if the investor is using different identities. However, an investor could just generate and submit two scope claims for the same Ticker by replacing *SHA(SCOPE_DID, INVESTOR_UNIQUE_DID)* with a random value. Nobody without access to *INVESTOR_UNIQUE_DID* can detect that the hash has not been calculated correctly. The two claims will have different *SCOPE_IDs* and would therefore be accepted by the chain. Nobody without access to *INVESTOR_UNIQUE_DID* can see that the hash has been replaced with some random data.

A malicious investor could invest into an asset with two DIDs without linking them, and, as a result, the asset issuer would assume incorrect information about the number of investors in their asset [2]. In order to mitigate this issue Polymath introduced a new approach in the confidential identity implementation (called *PIUS v2*) which is planned to replace the previous version before main-net launch.

4.1.3 Multiple arithmetic over/underflows

We identified multiple arithmetic overflows and underflows during the course of our audit which could lead to various medium-severity vulnerabilities such as unexpected behaviors or the crash of any node compiled in debug mode or with overflow checks enabled. [3]

Three of these over/underflows are inside *asset_identifier.rs* and can lead to undefined behavior [3], [4], [5]. For example, on nodes that are compiled without overflow checks, this could lead to an invalid CUSIP id being marked as valid, which does not pose a security issue in itself. However, integer over/underflow can be considered undefined behavior. As such, the behavior in the case of an integer overflow might be non-deterministic (and depend on the compiler version in use, for example), and, in the worst case, this difference in behavior could lead to a chain split. Right now, it seems like all Rust compiler versions implement a wrap-around for integer over/underflows, but this might change in the future.

Another integer overflow is inside the *instantiate* extrinsic in *contracts/src/lib.rs* and may lead to the weight calculation's wrapping around and thus an underestimate of the weight of the extrinsic [7]. In the worst-case, a validator will still include an extrinsic with very high computation time in a block (because the overflow causes an underestimate of the weight) and the block will timeout. This could lead to a **DoS** of the whole chain with considerably low-cost requirements for the attacker.

An overflow inside *create_fundraiser* in *sto/src/lib.rs* causes the variable *offering_ammount* to wrap around and thus underestimate the amount to be locked for a fundraiser creator's account [8]. Subsequently, all the instructions created for this fundraiser (using invest extrinsic) may fail to execute due to insufficient funds.

This overflow also allows an attacker to create unlimited number of fundraisers. Furthermore, since their tokens are not properly locked, they may use this vulnerability to unlock their balances that have been locked for other purposes using a combination of this extrinsic and the *invest* extrinsic which will unlock the tokens from their account.

We also identified an integer underflow inside in *pallets/multisig/src/lib.rs* via *remove_multisig_signers_via_creator* which could lead to (partial) DoS of the multisig account in the following ways [10]:

- Proposals could become non-rejectable, because a guarding check in *unsafe_reject* will always fail.

- After triggering the underflow, if *remove_multisign_signer* is called, signers could be removed in a way that in the end there are fewer signers in the multisig than the minimum amount needed for a consensus to execute a proposal, making it impossible to approve/reject proposals.

- By calling *change_sigs_required*, one could set the number of signatures required for a consensus much higher than the number of actual members in the multisig. This way, there will not ever be a consensus reached to approve/reject a proposal.

We propose using **saturating** or **checked** arithmetic functions to mitigate these type of arithmetic over/underflows. Polymath fixed all these overflows in series of PRs (refer to Table 3).

4.1.4 No tipping mechanism may be used to delay time-critical transactions

The current transaction-payment implementation in Polymesh makes the following design-decisions:

- It disables the tipping mechanism. Thus, there is no possibility for users to increase the priority of their transaction with a tip.

- It bases the transaction priority solely on the transaction fee, as opposed to some ratio of the weight and length.

These design decisions open up the possibility for a **DoS Attack** [6]. The missing tipping mechanism and the current transaction priority mechanism allow an attacker to block important calls abusing *sudo_unchecked_weight* calls (or other as-much-weight-as-you-want extrinsics).

We suggest addressing the issues in the following way:

1. Change the transaction priority to the following formula:

$$priority = \frac{fee}{\max\left(\frac{weight}{MaximumBlockWeight}, \frac{len}{MaximumBlockLength}\right)}$$

This will de-prioritize extrinsics that take a large chunk of the *MaximumBlockWeight* or *MaximumBlockLength* and will give higher priority to legitimate transaction that are likely to not take huge chunks of either the *MaximumBlockWeight* or the *MaximumBlockLength*. Note that Substrate has also moved to this formula, see the function *fn get_priority* in *Substrate/frame/transaction-payment/src/lib.rs*.

2. Enable the tipping mechanism.

   By enabling the tipping mechanism, a potential attack that is, blocking important calls by filling up the transaction queue, can be remediated via outbidding an attacker.

   This issue was mitigated by Polymath via a series of PRs Polymesh (893, 918) and Polymath's Substrate fork. These changes include re-enabling the tipping mechanism for operational extrinsics and marking governing council and CDD provider callable extrinsics as *Operational*. Additionally, all normal transactions now have the same priority and are processed according to their *insertion_id*.

4.1.5 No deposit for on-chain stored parameters can lead to storage clutter

The *propose* extrinsic in *pallets/pips/src/lib.rs* takes parameters *url* and *description*, which are byte Vectors of arbitrary length. The deposit that is charged by the user for a proposal does not take into account the length of these byte vectors, which is stored on chain. This would allow an attacker to fill up storage very cheaply and clutter the blockchain storage [9].

A similar issue also exists in the contracts pallet. The extrinsic *put_code* accepts a parameter of type *TemplateMetadata* which contains a *url* and a *description* both of which are essentially a *vec<u8>* and no limits are enforced on the length of these parameters.

As a mitigation we suggested to charge a deposit that scales with the length of the *url* and *description* vectors.

Polymath mitigated this issue by limiting the strings, vectors, BTrees, etc to a fixed length which is currently configured as 2048.

4.1.6 Proofs may be replayed for different claim types

The function *evaluate_claim* in *primitives/src/valid_proof_of_investor.rs* will generate a message based on the claim. However, the message does not contain any indication of the claim type being used [11]. Therefore, the same proof may be used to verify different claims, e.g. *Scope::Ticker(x)* and *Scope::Identity(x)* with the same parameter (byte array) *x*.

Our suggestion is that different claim types have a different role in the protocol and should be signed/verified with a different *SigningContext* to avoid any ambiguity on the meaning of the signature. However, since the current version of Polymesh does not use claim types other than *InvestorUniqueness*, it is not vulnerable. Polymesh will add a *SigningContext* for different claims types when they will be introduced in the future.

4.1.7 Risks through FIFO transaction processing scheme

We also have the following concerns regarding the proposed scheme of processing normal transactions in a FIFO order introduced in PR [15]:

Forking Substrate introduces a security risk. Forking the Substrate codebase comes with two inherent risks:

1. Since there are no security advisories for Substrate and security vulnerabilities are fixed via "silent patches", Polymath would need to port all changes from Substrate to Polymath's fork of Substrate should Polymath choose to fork Substrate.

2. Moreover, modifying parts of the Substrate codebase itself could introduce new, unforeseen, security vulnerabilities. The different modules in Substrate have complex interactions/dependencies with each other, with lots of intricacies. Modifying code in one module could have subtle side-effects in other modules, which could easily result in a security vulnerability.

FIFO provides only partial protection against front running. If transactions are processed by the FIFO principle, an attacker can just broadcast more transactions at a high frequency, which makes it very likely that the attacker will block the transaction slots for other people. Also note that the order of transactions is in no way finalized before the transactions make it to the blockchain. An attacker who is well connected in the gossip network could monitor pending transactions and quickly broadcast front running transactions for these transactions in an aggressive way (broadcasting to many validators in parallel). In such cases, a significant portion of legitimate validators would see the front running transaction before seeing the original transaction. It will also give participants in the network that have a short latency when communicating to the validator an advantage, comparable to "High Frequency Trading" situations in the traditional financial sector.

Only enabling tipping for operational extrinsics could an allow an attacker to potentially stall the chain for normal extrinsics. If the transaction priority is set to a fixed value for all normal extrinsics, but tipping remains enabled for operational extrinsics, this introduces the possibility for an attacker to stall the chain for an extended period of time, abusing the higher priority of operational extrinsics. Note that, as of now, operational extrinsics do have an inherently higher priority than normal extrinsics, which could aggravate this problem, depending on how exactly the FIFO scheme is implemented.

5 Evolution suggestion

Polymesh understands that security is an integral part of the Polymesh development process, ensuring the product and its users are well protected. Parts of that process towards a security-mature product are conducting thorough, regular reviews of Polymesh's critical codebase that help to harden the codebase and train developers to cultivate a security mindset.

Moving forward, we suggest taking the following measures to harden Polymesh's codebase against potential security vulnerabilities introduced in future development.

Reconsider the FIFO transaction priority scheme. During this audit, Polymath decided to opt for a FIFO scheme to prioritize transactions to minimize the possibility of front running. As laid out earlier in this report, this scheme introduces several security risks, including the ability for an attacker to delay "normal" extrinsics' being included on-chain and the need to fork Substrate. To minimize risk, we recommend moving away from this scheme towards a transaction priority system that introduces tipping for “normal” extrinsics and does not require a Substrate fork.

Fork as little Substrate code as possible. Polymesh requires various design/implementation changes to the way Substrate is implemented, concerning various Substrate-modules and – in the case of the transaction priority scheme – even the core Substrate codebase. To implement the behavior desired by Polymath, Polymesh includes multiple modules forked from Substrate. Polymath plans to fork Substrate as a whole to introduce further changes. Forking modules and Substrate itself introduces considerable security risk, as patches for security fixes need to be ported to the forked codebase. Porting patches is a complex, intricate process, which could introduce new vulnerabilities into Polymesh. This issue is aggravated by the fact that Parity does not publish security advisories, so Polymesh would need to port every change made to Substrate in order to not miss any security fixes.

To remove the need for forking Substrate, we recommend creating pull requests to the Substrate code base that would make the Substrate codebase itself more configurable so it can be used by Polymesh as-is.

Have any custom cryptography solutions peer-reviewed: For some of its functionality, Polymesh relies on cryptography that is based on established primitives (like the Discrete Logarithm assumption) but adds some custom logic on top of it. It is generally considered hard to design secure cryptography solutions. For that reason, we recommend to only resort to “custom” cryptography solutions when absolutely necessary and always conduct a peer-review of the custom design before including it in production code.

Enforce safe math functions by default and employ fuzz-testing to detect arithmetic bugs early in the development process. In the spirit of defensive programming, we recommend enforcing the use of safe-math functions, such as *saturating_add* and *checked_add* throughout the whole codebase, even if there is no immediate indication that a math operation could overflow. To further alleviate the issue of integer overflows, we recommend employing fuzz-testing to identify early on the in the development process any arithmetic or other bugs (such as extrinsics that trigger a panic in the code). Ideally, Polymath would continuously fuzz their code on each commit made to the codebase.

Remove or feature-guard debug code. The Polymesh codebase still has some debug code included in the codebase. While the debug code is explicitly marked as such via comments, it is not feature-guarded. To prevent that any debug code is still present in production, we recommend to either completely remove the debug code or feature guard it with a consistent build flag that is disabled by default.

6 References

[1] "Polymesh audit repository issues, Issue #5," [Online]. Available: https://github.com/PolymathNetwork/polymesh-audit/issues/5.

[2] "Polymesh audit repository issues, Issue #11," [Online]. Available: https://github.com/PolymathNetwork/polymesh-audit/issues/11.

[3] "Polymesh audit repository issues, Issue #1," [Online]. Available: https://github.com/PolymathNetwork/polymesh-audit/issues/1.

[4] "Polymesh audit repository issues, Issue #2," [Online]. Available: https://github.com/PolymathNetwork/polymesh-audit/issues/2.

[5] "Polymesh audit repository issues, Issue #3," [Online]. Available: https://github.com/PolymathNetwork/polymesh-audit/issues/3.

[6] "Polymesh audit repository issues, Issue #4," [Online]. Available: https://github.com/PolymathNetwork/polymesh-audit/issues/4.

[7] "Polymesh audit repository issues, Issue #6," [Online]. Available: https://github.com/PolymathNetwork/polymesh-audit/issues/6.

[8] "Polymesh audit repository issues, Issue #7," [Online]. Available: https://github.com/PolymathNetwork/polymesh-audit/issues/7.

[9] "Polymesh audit repository issues, Issue #10," [Online]. Available: https://github.com/PolymathNetwork/polymesh-audit/issues/10.

[10] "Polymesh audit repository issues, Issue #8," [Online]. Available: https://github.com/PolymathNetwork/polymesh-audit/issues/8.

[11] "Polymesh audit repository issues, Issue #12," [Online]. Available: https://github.com/PolymathNetwork/polymesh-audit/issues/12.

[12] "Polymesh audit repository issues, Issue #9," [Online]. Available: https://github.com/PolymathNetwork/polymesh-audit/issues/9.

[13] "Substrate PR #7849: Store dispatch info of calls locally in weight calculation," [Online]. Available: https://github.com/paritytech/substrate/pull/7849.

[14] "Polymesh PR #893: MESH-1518/Upgrade to Substrate's Polymath fork," [Online]. Available: https://github.com/PolymathNetwork/Polymesh/pull/893.

[15] "Polymesh#861," [Online]. Available: https://github.com/PolymathNetwork/Polymesh/pull/861.

[16] "Polymesh PR #918: Allow tipping for Operational TX from CDD/GC member," [Online]. Available: https://github.com/PolymathNetwork/Polymesh/pull/918.