[Critical] Unmitigatable exposure to theft of funds, XSS injection, and malware distribution due to the lack of idempotency in core Ethereum Execution JSON-RPC API logic in a compromised provider scenario.

Joseph Habel

December 5, 2022

1 Contents

2 Overview

The following report documents a variety of attack vectors that exist between an interface that acts as an RPC consumer, and an independent RPC node provider whose layer of operational trust has been compromised. While these vectors might all initially appear out of scope, the underlying issue being raised is not the existence of these vectors, but rather the lack of ability to either identify this behavior from the log stream of an end consumer, or to create redundancy systems that allow for these kinds of compromises to be identified and mitigated in real time. This report proposes a solution of adding a handful of additional RPC methods that would enable the creation of systems that can provide both mitigation and detection mechanisms, while not impacting the backwards compatibility of the Ethereum Execution API.

There are two primary application patterns that are exposed to a significant risk if the provider trust is compromised, either:

1. Relying on an address returned from an on-chain lookup for some form of transaction construction, leading to exposure of **Theft of Funds**
2. Relying on a URI returned from an on-chain lookup to resolve to some form of media, leading to exposure of **XSS or Malware Distribution** depending on the kind of resolved resource

While the following attack scenarios outlined don't occur directly from the implementation of the Ethereum RPC Execution API, the lack of idempotency on `eth_call`, when called with a `'latest'` block parameter, makes it impossible to build automated solutions that can either detect or mitigate compromised trusted providers, including the following examples:

- detecting a compromised node in a fleet deployment of RPC nodes.
- allowing for a consumer to wait for the response of multiple independent nodes or providers before opting to trust the response
- allowing for a consumer to audit the result of an RPC response from a remote server historically.

2.1 Brief Example

For a very brief example, let's say that I have access to two RPC nodes, and I want to know the address of the current owner of item 1 in an ERC721 collection. If I were to ask both nodes for `collection.ownerOf(1);` through something like `web3.js`, that call would be compiled down to an `eth_call` method to the collection contract, queried at the `"latest"` block. In theory I should get the same answer back from both nodes, but what if I get back different answers? Well, there could be two explanations for that:

1. One of the nodes has been compromised and is behaving maliciously.
2. A transfer of that asset happened in the last block, and one node was a block behind the other.

The lack of idempotency that comes from these all being made as `"latest"` block calls means that if I were to have logged that behavior in my interface, been alerted to it in production 5-10 minutes later, I'd have no way of easily knowing if either of these nodes are acting in bad faith, or if this is essentially just a network wide race condition.

Just as I wouldn't be able to historically detect if there had been a breach of trust, it's even more difficult to automate a system to perform this kind of check in flight. Without knowing the block number that either of these nodes computed that piece of data at, it becomes a nasty game of trying to determine the difference between if the nodes have different `"latest"` block states, or if one of the nodes is doing something malicious. With no obvious means to quickly compare answers with just a single request to the other party, all solutions would require iterating around a range of blocks for both parties after polling each one for the height, only after they've already returned incongruent responses. This turns what should have been a 2 call solution into an solution that requires an indeterminate amount of calls.

2.2 Simple Steps to Enable Mitigation

This report recommends introducing a handful of new RPC methods to the Ethereum Execution API to allow for semi-idempotency of all methods that currently allow for a `"lastest"` block parameter. This provides the ability to execute a request to the existing RPC method utilizing data returned in the response of the new RPC methods.

Using `eth_call` as an example, introduce a new method, `eth_callAndBlockNumber`, that takes the same request format as the existing `eth_call`, and simply returns a new output. Instead of simply returning the resulting hex string as the `result` field, return an object with a `value` field that contains the expected hex string, and a `blockNumber` field that gives context to when this call was made.

Introducing such calls would enable an application to utilize a middleware layer to proxy and transform any `eth_call` requests to `eth_callAndBlockNumber`. An application would then be able to either await the response before asking other parties for confirmation, or to log the request and response packets, and detect any abnormalities by simply replaying the request, and swapping the `"latest"` block number parameter for value returned in `result.blockNumber` of the response object.

2.3 What This is Not

I understand that this potentially looks like known issues, most of which aren't immediately addressable, and some may have solutions on the long-term roadmap. What I want to make clear here is that this bug report is:

- **Not about light clients**
- **Not about weak subjectivity**
- **Not about data availability**
- **Not about a lack of consumer verifiable state proofs**
- **Not about the trusted relationship between Execution API client and consumer**

I'm aware all of the above stand in and act as the potential long-term, cryptographic approach for addressing the underlying issue of the trusted Execution API.

The most important part to keep in mind is that the current Execution API makes it **impossible to catch these kinds of intrusions in logging contexts.** The proposed remediation is to allow for existing applications to **start looking for these intrusions, and start building mitigation layers.**

3 Severity

Effected Component: Ethereum Execution API Specification

SL:3/M:9/O:7/S:9/ED:5/EE:5/A:5/ID:9/LC:0/LI:0/LAV:0/LAC:8/FD:8/RD:9/NC:7/PV:0

3.1 Threat Agent Factors – 7

- **Skill Level:** 3 – An actor would be required to understand:
  - How to read incoming RPC requests for specific calls.
  - How to reconstruct RPC responses to with desired data.
  - Deploy smart contracts.
  - Either: run available RPC nodes, infiltrate existing ones.
- **Motive:** 9 – Direct theft of funds is available
- **Opportunity:** 7 – Requires either compromising or providing RPC nodes, and deploying malicious contracts
- **Size:** 9 – Anyone can stand up a public RPC and link it on chainlist, or participate in permisionless networks

3.2 Vulnerability Factors – 6

- **Ease of Discovery:** 3 – Requires at least a broad view of the entire Web3 stack
- **Ease of Exploit:** 5 – Once discovered, the actual exploits are pretty straight forward
- **Awareness:** 3 – While it can be pieced together through open source repositories, it takes a nuanced understanding of various deployments
- **Intrusion Detection:** 9 – Lack of idempotency makes detection impossible to review strictly from log events

3.3 Technical Impact Factors – 2

- **Loss of Confidentiality:** 0 – N/A
- **Loss of Integrity:** 0 – N/A
- **Loss of Availability:** 0 – N/A
- **Loss of Accountability:** 7 – Might be traceable depending on method of attack.

3.4 Business Impact Factors – 6

- **Financial Damage:** 8 – Significant loss to on chain TVL
- **Reputation Damage:** 9 – Completely undermines the trust of DeFi and NFT security at large
- **Non-compliance:** 7 – Due to growing trends, significantly more applications are gaining exposure without available mitigation measures
- **Privacy Violation:** 0 – N/A

4 Important Context

A common thought that I imagine anyone reading this report might immediately reach for is, "these are issues between the interface and consumer, and the chosen third-party provider, and under the scope of the core RPC server." While this thought is correct, its important to not let it cloud the fact that there currently **do not exist any mechanisms** in the Ethereum Execution API to address this without re-writing a sizable amount of existing interface design code. Additionally, there are no mechanisms to allow for the ability to audit a historic request–response pair to search for any previous history of this behavior.

The existing landscape of provider adoption and the utilization of `eth_call` with `"latest"` in interface design combines to **form a systemic risk** that is currently not addressable due to the current implementation design without either significantly breaking backward compatibility for existing integrations and applications, or forcing adaption in the existing economies of scale. This current landscape also implicates exposure for a significant amount of the TVL within smart contract applications on chain.

It does not feel unfounded to draw parallels to the risk that was imposed by the original DAO hack. While ultimately out of scope of the core Ethereum specification, its impact was so large that it ultimately resulted in a response regardless. Unlike, the DAO hack however, while it might be argued the scope of these attacks is technically outside of the Ethereum RPC API specification and implementation, **modifications can be made** to the Ethereum RPC API specification and implementations **that can help provide mitigation** to these systemic deployment risks without breaking backwards compatibility.

4.1 About Developer and Provider Trends

An additional point of context that I want to point out is that while it might be tempting to assume there's still an operational security barrier that's significantly protecting the provider-consumer relationship, narrative trends about decentralization and public goods have pushed developers into widely available options **without a clear understanding of the risks involved** from doing so.

Permissionless RPC node networks, while attempting to fill the decentralization void in the blockchain API layer, have provided **a trivial means of gaining access** to a subset of application interface service without having to actually compromise any existing deployments: simply stake a few thousand dollars worth of tokens and start gaining access to serving applications. This has set an expectation that other provider services can simply bundle and resell other blockchain API as a service offerings.

The public goods narrative has also driven altruistic groups and individuals to stand up publicly available RPC endpoints for free. Chainlist is an example of a popular aggregator for these public RPC endpoints, which often get quietly abused by developers building them into a round robin system in their interfaces. There's **little barrier to honey-potting** this traffic: just stand up an RPC server and throw up a PR to the chainlist repository to have it listed, and sit back and watch traffic slowly start to flow in.

These two trends severely elevate the number of applications and the TVL that's currently at risk, so much so that an orchestrated attack would have a **major erosion of trust** to smart contract application ecosystems as a whole. Paired with the erosion of trust from the centralized exchanges, it's imperative to get ahead of the risks outlined as more trust shifts towards on-chain systems.

Regardless of whether this report semantically is considered in scope for the bug bounty program, I ask that you take the time to understand the following attack vectors and **implement the proposed remediation regardless of who is ultimately the responsible party** to allow for solutions to roll out that can help applications mitigate these risks, **without needing to rewrite** any existing application logic.

5 Steps to Reproduce Common Attack Patterns

In general, the pattern to execute any of these attacks is relatively trivial for someone who manages to compromise an RPC server:

1. Find the app you want to attack.
2. Deploy malicious contracts, assets, or malware.
3. Listen for specific calls, and swap the expected address or asset link for the malicious one.
4. Wait for a user to interact with said contract or asset.

For all of the following examples, these are all predicated on compromising the RPC provider of an application that fits the criteria of those exploits.

5.1 Theft of Funds

Theft of Funds attacks exist when a node has an opportunity to poison the app's interface by changing the expected contract addresses that the user possibly will send funds to, with contracts deployed to return the otherwise expected data, while keeping the user's funds that are sent to it. This can happen by either:

- Targeting any application that looks up contracts from an on-chain registry
- Targeting an application that hosts its interface using a combination of IPFS and an on-chain name service provider

5.1.1 Native tokens directly in the interface

1. Find an application that uses the value of a contract registry lookup on chain as the destination of a `toAddress` in a transaction.
2. Deploy a contract that:
    1. Copies the interface that is resolved
    2. Proxies as much back to the original as possible, allowing it to function in the UI
    3. Overrides any function with `transfer` logic to transfer to a personally controlled wallet
3. Listen for the registry lookup and replace the expected result address with the newly deployed contract

5.1.2 ERC20/721/1155 assets directly in the interface

1. Find an application that uses the value of a registry lookup on chain in an `approve` - `transferFrom` flow.
2. Deploy a contract that:
    1. Copies the interface that is resolved
    2. Proxies as much back to the original as possible, allowing it to function in the UI
3. Listen for the registry lookup and replace the expected result address with the newly deployed contract
4. Listen for the `approve` transaction for the malicious contract to hit on-chain, and then fire a `transferFrom` call for the approved amount from the newly deployed contract to a personally controlled wallet.

5.1.3 By jacking an IPFS hosted interface

1. Find a smart contract interface stored on IPFS that is resolved via a lookup on a smart contract name service registry.
2. Clone and modify that interface to redirect any payments made to a personally controlled address then upload the modified version to IPFS.
3. Listen for the call to the smart contract to get the IPFS hash of the interface and replace it with the hash of the modified interface.
4. Since the interface is hosted on IPFS, the only indication a user would have they're using a unsafe interface is if they already know the hash of the expected interface.

5.2 XSS Injection

XSS injection occurs on any application that resolves externally hosted media from an on-chain URL. Any application interface that renders an image is going to be susceptible to some form of this. If the value is used unsanitized in the interface, it allows for direct XSS injection. If the value is sanitized, then a malicious image can be deployed that will render to the user's expected response, but can have embedded JavaScript that runs when opening the image source directly.

5.2.1 Directly in the interface

1. Find an application that uses a smart contract response as an unsanitized input to the `src` field of an `<img>` tag.
2. Listen for the call to the smart contract to get the URI and replace with the XSS payload.

5.2.2 In an image asset

1. Find an application that uses a smart contract response as an input to the `src` field of an `<img>` tag.
2. Download the expected image and convert it to an SVG file.
3. Inject the XSS payload after the `<metadata>` tag but before the `<g>` tag, rename the file to reflect the original image filetype, and upload the file to an accessible endpoint.
4. Listen for the call to the smart contract to get the URI and replace with the URI for the injected SVG.
5. The image will display as expected in the interface but will execute the XSS payload when accessed directly.

5.2.3 Through an https redirect

1. Find an application that uses a smart contract response as a destination `http(s)` URL.
2. Deploy a webpage with a XSS payload which executes and then redirects to the expected URL.
3. Listen for the call to the smart contract to get the URL and replace with the deployed redirect webpage.

4. When the desired URL is accessed, the XSS payload will execute before sending the user to the expected webpage

5.3 Malware Distribution

Malware distribution is currently limited to the deployment configuration where an application interface is resolved from a decentralized name service which points to an IPFS hash. Due to the fact that the resulting destination is not human readable, copying the expected aesthetic of the interface would be sufficient to inject whatever code is desired in the application flow, as the user would not be able to know anything has been changed simply from the different hash.

5.3.1 Through an IPFS Webpage

1. Find an application that uses a smart contract service registry to resolve to an IPFS hosted html webpage.
2. Clone the existing html webpage and modify to whatever extent needed to either redirect the user to a malicious download, or leverage techniques to do so without the user's awareness, then upload the modified version to IPFS.
3. Listen for the call to the smart contract to get the IPFS hash of page and replace with the hash of the deployed page.
4. Since the interface is hosted on IPFS, the only indication a user would have they're using an unsafe interface is if they already know the hash of the expected interface.

6 Impact

A large motivation for pursuing the needed changes through the bug bounty program is the large scale impact that these exposed vectors currently have to the network at large. Trends in the RPC provider space have normalized the use of less secure options amongst developers, and the broad nature of the attack patterns implicate a significant number of existing smart contract applications already deployed in production.

6.1 Existing Provider Risk

While it might be tempting to point out that the Ethereum RPC Execution API was only designed to be used in a one RPC server per consumer configuration, the reality is that the vast majority of public smart contract interfaces do not operate under this model, and instead rely on one of the following 4 primary options:

1. Through a provider that solely runs a shared load balanced fleet of nodes.
2. Through a provider that provides dedicated access to an individually managed node.
3. Through a "decentralized" provider network, providing access to either a load balanced or dedicated nodes from independent 3rd party providers.
4. Through publicly exposed RPC endpoints, a mixture of the above 3.

The model of hosted Ethereum RPC API access was championed by Infura, and has since been adopted a large amount of competing services. The centralization of Infura's Ethereum access led to two leading narratives emerging:

- The decentralization of RPC node networks through economic systems.
- RPC API access as a public good.

Both of these have led to some very troubling patterns for both the developers and providers under the assumption that since the transactions were already signed, there isn't a financial risk in this interaction layer. The decentralization aspect of the system has been particularly troubling as it's created a perverse financial mechanism of SaaS arbitrage on API providers, creating both a race to the bottom and profit

extraction layers that siphon away the value otherwise needed to sufficiently protect these systems. Similarly, driving the price down so low – to zero in the case of public RPC options – has flocked developers to the least operationally secure options.

6.2 Existing Application Exposure

The very broad requirements – that an application needs to only rely on a registry or resource resolution – implicate a vast collection of applications. This makes poorly secured RPC nodes very attractive targets due to the likelihood that there is exploitable traffic flowing through the network. Without listing any applications by name, the following are examples of use cases that would be exposed to the following attack vectors:

6.2.1 Theft of Funds – Vulnerable Application Patterns

- Decentralized exchanges (that use an on-chain registry)
- Lending contract aggregators (that use an on-chain registry)
- Market places (that use an on-chain registry)
- Auction houses (that use an on-chain registry)
- Wallets and apps that resolve text records to addresses
- Social profiles that link wallet ownership
- Single actor bridges/multi-actor bridges that use the same RPC infra

6.2.2 XSS Injection – Vulnerable Application Patterns

- Anything that resolves media resources from an external URI.

6.2.3 Malware Distribution – Vulnerable Application Patterns

- Decentralized name spaces that resolve to an IPFS hash as the URI.

7 Proof of Concept

Due to the nature that the above attacks impacting users of smart contract applications, but aren't directly caused by the underlying RPC API logic, including a proof of concept of any of the above situations offers no direct benefit for the developers who would be mitigating these vectors by patching the RPC implementation. If any of the above examples in the "Steps to Produce" are unclear, I am happy to provide any further clarification needed, short of any reference implementation out of respect for any of the impacted projects unknowingly exposed to these vectors.

8 Remediation and Disclosure

Full remediation of this problem is a multi-pronged approach, with part of the responsibility eventually falling back to the application designer and/or the RPC provider. This path to remediation has 2 vital steps:

1. Introduce New RPC Methods
2. Library and Middleware Compatibility for New Methods

The goal for the remediation is to have a minimal lift mitigation layer available for the following 2 main usecases:

1. Applications that utilize a single private RPC provider.
2. Applications that utilize a collection of public RPC providers.

Only the first action, introducing RPC methods, is relative to the audience of this report. Simply introducing those mechanisms, however, is not sufficient protection to disclose the existence of these vectors to the public. Only after step 2 should any disclosure of these vectors be made public. At minimum, applications are going to need some `web3.js/ethers` layer to allow for them to check a collection of provider responses against each other.

If there's commitment to introducing these methods as defined, then both steps can operate in parallel, but a clear commitment to the implementation of the newly introduced methods in 1 is required before any work towards 2 can begin.

8.1 New RPC Methods

The proposed solution to allow for means of detection and mitigation of compromised RPC nodes is to introduce the following 5 new RPC methods. This allows for applications to start having access to methods that can be used to take a stateful `"latest"` block call of their existing counterparts that are used throughout production, catch it, and easily transform it into a method whose response can then be utilized to reconstruct the original call in an idempotent form.

8.1.1 `eth_callAndBlockNumber`

- **Request `params` structure:** identical to `eth_call`.
- **Response `result` structure:**

```
class CallAndBlockNumberResult:
    value: str # value of eth_call's result
    blockNumber: str # value of eth_blockNumber's result
```

8.1.2 `eth_getBalanceAndBlockNumber`

- **Request `params` structure:** identical to `eth_getBalance`.
- **Response `result` structure:**

```
class GetBalanceAndBlockNumberResult:
    value: str # value of eth_getBalance's result
    blockNumber: str # value of eth_blockNumber's result
```

8.1.3 `eth_getStorageAtAndBlockNumber`

- **Request `params` structure:** identical to `eth_getStorageAt`.
- **Response `result` structure:**

```
class GetStorageAtAndBlockNumberResult:
    value: str # value of eth_getStorageAt's result
    blockNumber: str # value of eth_blockNumber's result
```

8.1.4 `eth_getTransactionCountAndBlockNumber`

- **Request `params` structure:** identical to `eth_getTransactionCount`.
- **Response `result` structure:**

```
class GetTransactionCountAndBlockNumberResult:
    value: str # value of eth_getTrasnactionCount's result
    blockNumber: str # value of eth_blockNumber's result
```

8.1.5 `eth_getCodeAndBlockNumber`

- **Request `params` structure:** identical to `eth_getCode`.
- **Response `result` structure:**

```
class GetCodeAndBlockNumberResult:
    value: str # value of eth_getCode's result
    blockNumber: str # value of eth_blockNumber's result
```

8.2 Library and Middleware Changes

Once the above methods are implemented, the following 2 criteria are needed before making the motivations for these changes public:

1. The `AndBlockNumber` variants are the methods that Web3 libraries use over the network.
2. A middleware layer exists that allows for only accepting responses that a majority of providers agree upon.

These two changes should provide developers with the tools they need to easily implement a multi-provider approach should they wish, as well as leaving proprietary providers an interface layer that allows them to check the validity of a response historically.

8.2.1 Using the New Variants Over the Network

This change should happen in any web3 utility libraries such as `web3.js` and `ethers`. These changes should introduce the `AndBlockNumber` calls directly, and, the legacy methods should be transformed such that calling `eth_call` would actually call `eth_callAndBlockNumber` outward to the network, and then the response would simply be transformed for the consumption context. What this does is make sure that a majority of traffic that is ultimately routed through providers is now able to at least be double checked.

8.2.2 A Middleware Mitigation Layer

Additionally, some middleware mitigation layer is needed. This layer should be able to take a list of RPC provider URLs and, for a given RPC call, it should be able to check the response of that call against all providers before moving forward. Having this in place would allow applications to add this mitigation simply by wrapping their API provider layer, minimizing any changes to just a few lines.

8.3 Disclosure

Only after there exists a means for the majority of traffic to begin flowing through providers in a form that can be audited and a minimal change mitigation layer exists, then can the motivations for introducing RPC layer changes be disclosed publicly. Arguably, prioritizing disclosure to the large scale providers before the public has the potential to allow for applications to have available mitigation without any changes but ultimately isn't required.

9 Appendix

9.1 Quantifying Impact

I've included methods to allow for quantifiable estimation for the financial impacts and profitability of undertaking such an attack. While these formulas are going to require some estimations, they can quickly point to some intuitions for mentally gauging the existing network risks.

Some of these trends and insights include:

- Unvetted public RPCs that attract a large variety of applications can be dangerous
- Market driven networks have fully measurable costs to capturing a majority of service of the applications that use them
- Large volume apps that operate through a dedicated node setup are juicy targets
- While it might mean playing a waiting game, load balanced deployments likely provide more scalable opportunities than dedicated deployments

9.1.1 Expected number of compromised links from compromising a dedicated node

If a dedicated node for a given application is compromised, if its interface, i makes a total of s exploitable smart contract lookups across any configuration of API calls, it should be expected to compromise X^d total impressions with a malicious target, where:

$$X^d(i) = s$$

9.1.2 Expected number of compromised links from compromising nodes in a load balancer

Assume a provider has a fleet of N nodes, and each application interface serviced by the provider, i, makes a total of r_i requests, l_i of which are calls to lookup a contract(s) exploitable in the above fashion, with a density of ρ_i contracts per exploitable call. For an application that doesn't batch these calls, $\rho = 1$, and for one that batches these into 20 call batches, $\rho = 20$. In a load balancer that distributes uniformly an actor that has targeted a set of interfaces, I, someone able to compromise n nodes should be expected to compromise $X^{lb}(n, I)$ total impressions with a malicious target, where:

$$X^{lb}(n, I) = \frac{n}{N} \sum_{i \in I} \frac{\rho_i l_i}{r_i}$$

9.1.3 Expected compromised value from compromising a dedicated node

Again, assuming the interface, i being exploited will point to a set of contracts C_i, each that facilitate a total volume of V_c in financial value, with η_c being the fraction of a volumes that happens through the interface directly, then it can be assumed that a node is able to compromise $\alpha^d(C)$ in total value, where:

$$\alpha^d(i) = \sum_{c \in C_i} \eta_c V_c$$

9.1.4 Expected compromised value from compromising nodes in a load balancer

Assuming the set of interfaces, I, being exploited will point to a set of contracts, with lookups for those contracts being sent in batches B_i, each batch containing a set of contracts, C_b that facilitate a total volume of V_c in financial value, with η_c being the fraction of volume that happens via the interface directly, and the interface converts impressions to transactions on the interface at a rate of β_c, with lookups for those contracts sent as batches of B_i then in a load balancer that distributes uniformly an actor that has targeted a set of interfaces, I, someone able to compromise n nodes should be expected to compromise $\alpha^{lb}(n, I)$ in total value, where:

$$\alpha^{lb}(n, I) = \frac{n}{N} \sum_{i \in I} \frac{1}{r_i} \sum_{b \in B_i} \sum_{c \in C_b} \beta_c \eta_c V_c$$

9.1.5 Expected Profit

Assume that an attacker is gain control of sets of nodes that service applications, through various different providers for varying costs. The total profit, P, of an attack can be estimated as follows:

- Sets of M, L:
  - A set of n_μ nodes acquired through market systems in a decentralized network, μ, each having the same fixed acquisition cost γ_μ, and having access to interfaces I_μ on the network.
  - A set of n_λ nodes compromised on a load balancer λ, all sharing the same one acquisition cost γ_λ, and having access to interfaces I_λ on the load balancer.
- The set of compromised dedicated nodes D, each having a unique acquisition cost γ_d, and matching interface i_d.
- The set of all interfaces I^{sup} across all compromised deployments, each with a deployment cost of γ_i to target, found by taking the union of the sets of interfaces in D, M, and L

$$P(D, L, M) = \sum_{d \in D} \alpha^d(i_d) - \gamma_d + \sum_{\mu \in M} \alpha^{lb}(n_\mu, I_\mu) - n_\mu \gamma_\mu + \sum_{\lambda \in L} \alpha^{lb}(n_\lambda, I_\lambda) - \gamma_\lambda - \sum_{i \in I^{sup}} \gamma_i$$