Decentralized Software Updates for Stake-Based Ledger Systems

Anonymous Author(s)

Abstract

Software updates are a synonym to software evolution and thus are ubiquitous and inevitable to any blockchain platform. In this paper, we propose a general framework for decentralized software updates in distributed ledger systems. Our framework is primarily focused on permission-less stake-based blockchains and aims at providing a solid set of enhancements, covering the full spectrum of a blockchain system, in order to ensure a decentralized but also secure update mechanism for a public ledger. Our main contribution is the proposal of a protocol that solves the decentralized software updates problem for public stake-based ledgers. Our protocol covers the full lifecycle of a decentralized software update from the ideation phase to the actual activation in the blockchain system. Moreover, we cover all types of software updates, from consensus rules changes to simple code changes and from bug-fixes to change requests. We introduce the concept of *update policies*, to describe the need for different deployment speeds, as well as deployment methods, based on the semantic context (i.e., metadata) of a software update. Also, we propose a solution to typical software updates problems, like the respect of update dependencies, the resolution of update conflicts and the verification of the authenticity and safety of downloaded updates, all in a decentralized setting. Finally, we propose a logical architecture that enables a smooth incorporation of software updates into all components of a blockchain system. We implement our ideas in a prototype update system for the Cardano blockchain and discuss implementation issues and validation results.

1 INTRODUCTION

Software updates are everywhere. The most vital aspect for the sustainability of any software system is its ability to effectively and swiftly adapt to changes (i.e., to software updates). Therefore the adoption of changes is in the heart of the lifecycle of any system and blockchain systems are no exception. Software updates might be triggered by a plethora of different reasons.

Typically, the main driver for a change will be a change request, or a new feature request by the user community. These type of changes will be planned for inclusion in some future release and then implemented and properly tested prior to deployment into production. In addition, another major source of changes is the correction of bugs (i.e., errors) and the application of security fixes. These usually produce the so-called "hot-fixes", which are handled in a totally different manner (i.e., with a different process that results into different deployment speed) than change requests. In addition, depending on the severity level of the problem, or the priority (i.e., importance) of the change request there are different levels of speed and methods for the deployment of the software update into production.

More specifically, for blockchain systems, a typical source of change are the enhancements at the consensus protocol level. There might be changes to the values of specific parameters (e.g., the maximum block size, or the maximum transaction size etc.), changes to the validation rules at any level (transaction, block, or blockchain), or even changes at the consensus protocol itself. Usually, the reason for such changes is the reinforcement of the protocol against a broader scope of adversary attacks, or the optimization of some aspect of the system like the transaction throughput, or the storage cost etc.

Finally, there are also more radical changes, which are usually caused by the introduction of new research ideas and the advent of new technology, which becomes relevant. These type of changes usually introduce new concepts and are not just enhancements and thus trigger a major change to the system.

Context of this paper. In this paper, our focus is on the software update mechanism of stake-based blockchain systems. We depart from the traditional centralized approach of handling software updates, which is the norm today for many systems (even for the ones that are natively decentralized, like permission-less blockchain systems) and try to tackle common software update challenges in a decentralized setting. We consider the full lifecycle of a software update, from conception to activation and propose "decentralized alternatives" to all phases. Essentially, we introduce a decentralized *maintenance* approach for stake-based blockchain systems.

Problem Definition. Traditionally, software updates for blockchain systems have been handled in an ad-hoc, centralized manner: Somebody, often a trusted authority, or the original author of the software, provides a new version of the software, and users download and install it from that authority's website. Even if the system follows an open source software development model Nikos: citation ?, and therefore an update can potentially be implemented by anyone, the final decision of accepting or rejecting a new piece of code is always taken by the main maintainer (or group of core maintainers) of the system, who essentially constitutes a central authority. Even in the case, where the community has initially reached at a consensus for an update proposal (in the form of an *Improvement Proposal* document Nikos: citation?), through

Research partially supported by H2020 project PRIVILEDGE # 780477

the discussion that takes place in various discussion-forums, still it remains an informal, "social" consensus, which is not recorded as an immutable historical event in the blockchain and the final decision is always up to the code maintainer to accept it, or not. Moreover, the authenticity and safety of the downloaded software is usually verified by the digital signature of a trusted authority, such as the original author of the software.

On the other hand, in a decentralized setting software updates are handled quite differently: An update proposal can be submitted by anyone (just like anyone can potentially submit a transaction to the blockchain). The decision of which update proposal will be applied and which wont, is taken collectively by the community and not centrally. Thus, the road-map of the system is decided jointly. However, this process is not an informal discussion process but it is part of an update protocol and all relevant events generated, are stored within the blockchain itself, and thus recorded in the immutable update history of the system. Moreover, in the decentralized setting, the role of the code maintainer, who takes decisions on the correctness of the submitted new code, the versions of the software and also guarantees the validity of the downloaded software, is replaced by the stake majority.

Therefore, in the context of software updates for public stake-based blockchain systems, we define the capability to take stake majority-based decisions on: a) the priorities of software updates, b) the correctness of the new code, c) the maintenance of the code-base and d) the authenticity and safety of the downloaded software, as part of the consensus protocol and recorded on-chain, while at the same time fulfilling software dependencies requirements, resolving version conflicts, enabling different update policies based on update metadata and avoiding chain splits when updates are activated, as the *decentralized software updates problem* and this is the problem that we are trying to solve in this paper.

Limitations of Existing Solutions. To the best of our knowledge, there is no related work on the problem of the decentralization of software updates in the context of blockchain systems in a holistic manner, i.e., taking into consideration all phases in the lifecycle of a software update. Bitcoin [6], Bitcoin Cash Nikos: citation?, Ethereum [3] and Zcash Nikos: citation? use a "social governance" scheme, in which decisions on update proposals is reached through discussions on social media. This type of informal guidance is too unstable and prone to chain splits, or prone to becoming too de-facto centralized [2]. There exist blockchain systems [**?**], [**?**] that adopt a decentralized governance scheme, in which the priorities, as well as the funding of update proposals is voted on-chain as part of a maintenance protocol. However, these proposals do not follow a holistic approach to the decentralization software updates problem. Instead, their focus is merely on the ideation phase in the lifecycle of a software update, where an update proposal is born as an idea, and the community is called to accept, if it will be funded, or rejected. These solution do not deal with the rest phases in this lifecycle, which pertain to the approval of the source code correctness and the authenticity of the binaries that will be distributed for downloading, the maintenance of the code-base and the activation of the changes. That is why in the above cases, there exist a central authority (or group) that assumes the role of the source code maintainer. A similar approach is proposed by Bingsheng et al. [**?**], where a complete treasury system is proposed for blockchain systems, in which liquid democracy / delegative voting is followed. We also follow the approach of voting delegation, when technical expertise is required, in order to reach a decision for an update proposal. Similarly, Bingsheng's work is focused on the treasury system, which covers only the initial phase in the lifecycle of an update proposal.

Goal of the paper. Our primary goal in this paper, is to propose a secure software update mechanism that will enable a decentralized approach to the blockchain software updates problem. We want to examine all phases in the lifecycle of a software update and propose decentralized alternatives that are practical and can be adopted in the real world. To this end, we investigate the smooth incorporation of such a decentralized software update mechanism into the architecture of a blockchain system and implement a prototype that realizes our ideas in practice. From a security perspective, our proposal must ensure that: a) any stakeholder will always be able to submit an update proposal to be voted by the stakeholders community, b) an update proposal that is not approved by the stake majority will never be applied, c) an update proposal that it is approved by the stake majority will be eventually applied and d) it will provide guarantees for the authenticity and safety of the downloaded software.

Outline of the paper.

2 THE LIFECYCLE OF A SOFTWARE UPDATE

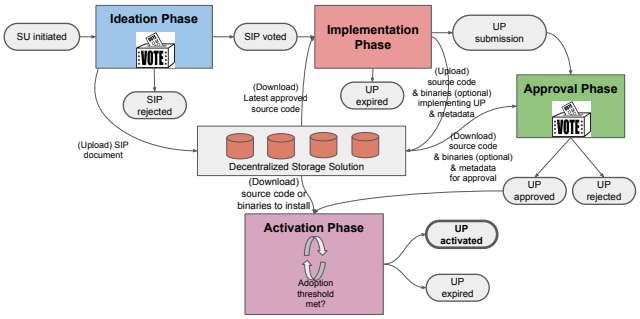
A *software update (SU)* is the unit of change for the blockchain software. It must have a clear goal of what it tries to achieve and why it would be beneficial, if applied to the system. Moreover, it should have a clear scope. In Figure 1, we depict the full lifecycle of a software update following a decentralized approach. In this lifecycle, we identify four distinct phases: a) the *ideation phase*, b) the *implementation phase*, c) the *approval phase* and d) the *activation phase*. In this section, we briefly outline each phase and at the subsequent sections, we provide all necessary details for realizing each phase in a decentralized setting.

The SU starts from the ideation phase which is the conceptualization step in the process. It is where an SU is born. During this phase, a justification must be made for the SU and this has to be formally agreed by the community (or the code owner). This justification takes the form of an improvement proposal document (appears as *SIP* in the figure and will be defined shortly). Once the SU's justification has been approved, then we enter the implementation phase. It is where the actual development of the SU takes place. The result of this phase is a bundle (UP) consisting of source

code (implementing the SU), metadata and optionally binaries produced for one, or more, specific platforms. This is submitted for approval and thus the approval phase follows. Once the UP (i.e., the source code implementing the SU) has been approved (by the community, or the code owner), the community is called for upgrading. The actual upgrading takes place in the activation phase, which is there to guard against chain-splits by synchronizing the activation of the changes.

Interestingly, the phases in the lifecycle of a SU are essentially independent from the approach (centralized or decentralized) that we follow. They constitute intuitive steps in a software lifecycle process that starts from the initial idea conception and ends at the actual activation of the change on the client software. Based on this observation, one can examine each phase and compare the traditional centralized approach, used to implement it, to its decentralized alternative. Moreover, not all phases need to be decentralized in a real world scenario. One has to measure the trade-off between decentralization benefits versus practicality and decide what phases will be decentralized. Our decomposition of the lifecycle of a SU in distinct phases helps towards this direction.

Figure 1: The lifecycle of a software update (a decentralized approach)



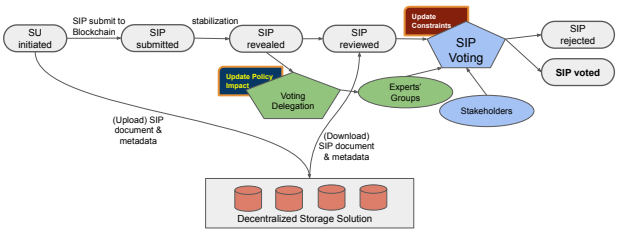
1 Ideation

A SU starts as an idea. Someone captures the idea of implementing a change that will serve a specific purpose (fix a bug, implement a new feature, provide some change in the consensus protocol, perform some optimization etc.). The primary goal of this phase is to capture the idea behind a SU, then record the justification and scope of the SU in some appropriate documentation and finally come to a decision on the priority that will be given to this SU.

Traditionally, in the centralized approach, a SU is proposed by some central authority (original author, group of authors, package maintainer etc.), who essentially records the need for a specific SU and then decides when (or, in which version) this could be released. In many cases, (e.g., Bitcoin [6], Ethereum [3]) the relevant SU justification document (called BIP, or EIP respectively) is submitted to the community, in order to be discussed. Even when this "social alignment" step is included in this phase, the ultimate decision (which might take place at a later phase in the lifecycle), for the proposed SU, is taken by the central authority. Therefore, the roadmap for the system evolution is effectively decided centrally. Moreover, this social consensus approach is informal (i.e., not part of a protocol, or output of an algorithm) and is not recorded on-chain as an immutable historical event.

The ideation phase in the decentralized approach is depicted in Figure 2.

Figure 2: The ideation phase.



In the decentralized setting, a SU starts its life as an idea for improvement of the blockchain system, which is recorded in a human readable simple text document, called the *SIP (Software* [1] *Improvement Document)*. The SU life starts by submitting the corresponding SIP to the blockchain by means of a special fee-supported transaction. Any stakeholder can potentially submit a SIP and thus propose a SU.

A SIP includes basic information about a SU, such as the title, a description, the author(s) etc. Its sole purpose is to justify the necessity of the proposed software update and try to raise awareness and support from the community of users. A SIP must also include all necessary information that will enable the SU validation against previous SUs (e.g., update dependencies or update conflict issues), or against any prerequisites required, in order to be applied.

A SIP is initially uploaded to some external (to the blockchain system) *decentralized storage solution* and a hash id is generated, in order to uniquely identify it. This is an abstraction to denote a not centrally-owned storage area, which is content-addressable, i.e., we can access stored content by using the hash of this content as an id. A change in the content will produce a different hash and therefore this will correspond to a different id and thus to different stored content. This hash id is committed to the blockchain in a two-step

[1] "Software" and "System" are two terms that could be considered equivalent for the scope of this paper and we intend to use them interchangeably. For example, a SIP could also stand for a System Improvement Proposal

approach, following a hash-based commitment scheme, in order to preserve the rightful authorship of the SIP.

Once the SIP is revealed a voting period for the specific proposal is initiated. Any stakeholder is eligible to vote for a SIP and the voting power will be proportional to his/her stake. Votes are specialized fee-supported transactions, which are committed to the blockchain.

Note that since a SIP is a document justifying the purpose and benefit of the proposed software update, it should not require in general sufficient technical expertise, in order for a stakeholder to review it and decide on his/her vote. However, in the case that the evaluation of a SIP requires greater technical knowledge, then a voting delegation mechanism exists. This means that a stakeholder can delegate his/her voting rights to an appropriate group of experts but also preserving the right to override the delegate's vote, if he/she wishes.

The delegation mechanism will also be used in order to implement the concept of an *update policy* that will be described in a later section and enables different activation speeds for a SU depending on its type (e.g., a bug-fix versus a change request, a SU that has a consensus protocol impact versus a no-impact one, etc.). For all these, special *delegation groups* will be considered, as we will discuss in the relevant section. A SIP after the voting period can either be voted or rejected. Details on the voting and delegation protocols can be found in the relevant section.

Note that in the decentralized approach the ideation phase could very well be implemented by a treasury system (e.g., similar to the one proposed by Bingsheng et al. [8]). A treasury is a decentralized and secure system aimed at the maintenance of a blockchain system that allows the submission of proposals (i.e., candidate projects) for improvement of the system. These proposals go through a voting process, in order to select the surviving ones. More importantly, the system is supported by a funding mechanism, where funds raised are stored in the treasury. These funds are used for funding the approved projects. Implementing the ideation phase with a treasury system, would enable additionally the appropriate management of the funding of each SU.

2 Implementation

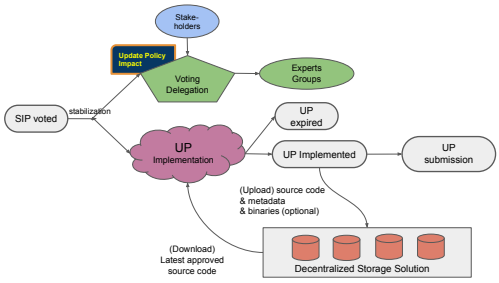
The voting of a SIP is the green light signal for entering the implementation phase. This is the period where the actual implementation of a SIP takes place. So one could very roughly imagine this phase as a box, where a SIP comes in as input and source code implementing the SIP comes out as output.

The scope of this phase is twofold: a) to develop the source code changes that implement a specific voted SIP and b) to execute a second voting delegation round, in order to identify the experts that will approve the new source code. At the end of this phase, the developer creates a bundle comprising the new source code, the accompanied metadata and optionally produced binaries, which we call an *update proposal (UP)*. The newly created UP must be submitted for approval, in order to move forward.

In the centralized setting, it is typical (in the context of an open source software development model), when a developer wants to implement a change, first to download from a central code repository the version of the source code that will be the base for the implementation and then, when the implementation is finished, to upload it to the same central code repository and submit a pull-request. The latter is essentially a call for approval for the submitted code. The central authority responsible for the maintenance of the codebase, must review the submitted code and decide, if it will be accepted, or not. Therefore, in the centralized approach the implementation phase ends with the submission of a pull-request.

The decentralized alternative for the implementation phase is identical to its centralized counterpart as far as the development of the new code is concerned. However, in the decentralized setting, there exist these major differences: a) there is not a centrally-owned code repository (since there is not a central authority responsible for the maintenance of the code), b) a delegation process is executed, in parallel to the implementation, as a preparation step for the (decentralized) approval phase that will follow and c) the conceptual equivalent to the submission of a pull-request must be realized.

Figure 3: The implementation phase.



In Figure 3, we depict the decentralized implementation phase. Similar to the centralized case, the implementation of a change must be based on some existing code, which we call the base source code and the developer must download locally, in order to initiate the implementation. However, in the decentralized setting there is not a centrally-owned code repository. All the approved versions of the code are committed into the blockchain (i.e., only the hash of the update code is stored on-chain). Therefore, we assume that the developer finds the appropriate (usually the latest) approved base source code in the blockchain and downloads it locally, using the link to the developer-owned code repository provided in the UP metadata. We abstract this code repository in Figure 3 with

the depicted decentralized storage solution. This conceptually can be any storage area that is not centrally-owned; from something very common, as a developer-owned Github repository to something more elaborate as a content-addressable decentralized file system.

It is true that the review of source code is a task that requires extensive technical skills and experience. Therefore, it is not a task that can be assumed by the broad base of the stakeholders community. A voting delegation mechanism at this point must be in place, to enable the delegation of the strenuous code-approval task to some group of experts. In a similar logic with the delegation process, within the ideation phase, discussed above, the delegation process could be leveraged to implement different update policies per type of software update.

As we have seen, the voting approval of a SIP signals the beginning of the implementation phase for this SIP. The SIP has an estimated implementation elapsed time that was included in the SIP metadata, submitted along with the SIP at the ideation phase. This time period, increased by a contingency parameter, will be the available time window for a SIP to be implemented. Upon the conclusion of the implementation, a bundled (source code and metadata) UP is created. The UP must be uploaded to some (developer-owned) code repository and a content-based hash id must be produced that will uniquely identify the UP. This hash id will be submitted to the blockchain as a signal for a request to approval. This is accomplished with a specialized fee-supported transaction, which represents the decentralized equivalent to a pull-request. SIPs that fail to be implemented within the required time framework, will result to expired UPs and the SIP must be resubmitted to the ideation phase, as a new proposal. The UP submission transaction signals the entering into the approval phase.

3 Approval

The submission of an UP to the blockchain, as we have seen, is the semantic equivalent to a pull-request, in the decentralized approach. It is a call for approval. Indeed, the main goal of the approval phase is to approve the proposed new code; but what exactly is the approver called to approve for?

The submitted UP, which as we have seen, is a bundle consisting of source code, metadata and optionally produced binaries, must satisfy certain properties, in order to be approved:

- *Correctness and accuracy.* The UP implements correctly (i.e., without bugs) and accurately (i.e., with no divergences) the changes described in the corresponding voted SIP.
- *Continuity.* Nothing else has changed beyond the scope of the corresponding SIP and everything that worked in the base version for this UP, it continues to work, as it did (as long as it was not in the scope of the SIP to be changed).
- *Authenticity and safety.* The submitted new code is free of any malware and it is safe to be downloaded and installed by the community; and by downloading it, one downloads the original authentic code that has been submitted in the first place.
- *Fulfillment of update constraints.* We call the dependencies of an UP to other UPs, the potential conflicts of an UP with other UPs and in general all the prerequisites of an UP, in order to be successfully deployed, *update constraints*. The fulfillment, or not, of all the update constraints for an UP, determines the feasibility of this UP.

From the centralized approach perspective the above properties of the new code that the approver has to verify and approve are not uncommon. In fact, one could argue that these are the standard quality controls in any software development model. The first property has to do with testing; testing that verifies that the changes described in the SIP have been implemented correctly and accurately. In the centralized approach this means that the main maintainer of the code has to validate that the new code successfully passes specific test cases, either by reviewing test results, of executed test cases, or by running tests on his/her own. Regardless, of the testing methodology or type of test employed (unit test, property-based test, system integration test, stress test etc.), this is the basic tool that helps the central authority to decide on the correctness and accuracy of the new code.

The second property for approving the new code has to do with not breaking something that used to work in the past. In software testing parlance, this is known as regression testing. Again, in the centralized approach, it is the main maintainer's responsibility to verify the successful results of regression tests run against the new code.

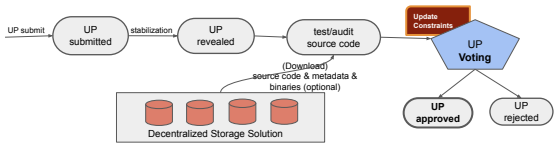
The third property has to do with the security of the new code and the authenticity of the downloaded software. The former calls for the security auditing of the new code. The latter, in the centralized case, is easy. Since, there is a trusted central authority (i.e., the main code maintainer), the only thing that is required, is for this authority to produce new binaries based on the approved source code, sign them and also the source code with his/her private key and distribute the signed code to the community. Then, the users only have to verify that their downloaded source code, or binaries, has been signed by the trusted party and if yes, then to safely proceed to the installation.

Finally, the last property that has to be validated by the approver pertains to the fulfillment of the update constraints. All the prerequisites of an UP must be evaluated and also the potential conflicts triggered by the deployment of an UP must be considered. For example, an UP might be based on a version of the software that has been depreciated, or rejected; or, similarly, it might be based on a version that has not yet been approved. Moreover, it might require the existence of third party libraries that it is not possible to incorporate into the software (e.g., they require licenses, or are not trusted). Then, we have the potential conflicts problem. What if the deployment of an UP cancels a previously approved UP, without this cancellation to be clearly stated in the scope of

the corresponding SIP? All these are issues that typically a code maintainer takes into consideration, in order to reach at a decision for a new piece of code.

Once more, the essential part that differentiates the decentralized from the centralized approach is the lack of the central authority. All the properties that have to be validated basically remain the same but in this case the approval must be a collective decision.

Figure 4: The approval phase.



The approval phase in the decentralized approach is depicted in Figure 4. As we have seen, an UP is a bundle consisting of source code, update metadata and optionally binaries produced from the source code, aiming at a specific platform (e.g., Windows, Linux, MacOS etc.). The update metadata have to include basic information about the update, its justification, they have to clearly state all update constraints and finally declare the type of the change (e.g., bug-fix, or change request, soft/hard fork etc.) and priority, in order to enable the appropriate *update policy* (we will return to these concept in the relevant section). The UP bundle is uploaded to some developer-owned code repository and a unique hash id, from hashing the content of the UP is produced. This UP hash id is submitted to the blockchain, along with a link to the code repository.

Similar to the ideation phase (where the corresponding SIP was submitted), the submission of a UP is a special fee-supported transaction that can be submitted by any stakeholder. The UP is committed to the blockchain following again a hash based commitment scheme, in order to preserve the rightful authorship of the UP.

Once the UP is revealed the delegated experts (remember the delegation that took place during the implementation phase) for this UP, will essentially assume the role of the main code maintainer that we described in the centralized setting. In other words, they have to download the source code, metadata and possible binaries and validate the aforementioned properties. The tools (e.g., testing) that the experts have available for doing the validation are no different than the tools used by the main maintainer in the centralized approach. Moreover, if binaries for a specific platform have been uploaded by the UP submitter, then the delegated experts must go through the process of reproducing a binary from the source code and verifying that it matches (based on a hash code comparison) the one submitted. If not, then the submitted binary must be rejected and this will cause a rejection of the UP as a whole. So there must be some extra caution when binaries are submitted along with source code, since the metadata need to include sufficient information for the approver to be able to reproduce the same binaries per platform.

Therefore the revealing of a UP, initiates a voting period for the specific proposal, in which the delegated experts must validate *all* the UP properties posed and approve, or reject it, with their vote. Any stakeholder is eligible to vote for an UP and the voting power will be proportional to his/her stake. If a stakeholder wishes to cast a vote, although he/she has already delegated this right to an expert, then this vote will override the delegates vote. Votes are specialized fee-supported transactions, which are committed to the blockchain. We will return to the voting protocol in the relevant section.

One final note is that, as we have described, the decentralized approval phase that we propose, entails transaction fees. This means that the approval phase is not so flexible from a practical perspective, as to be used iteratively (although technically this is possible). In other words, to reject an UP, then fix some bugs and upload a new version for review etc. An UP rejection means that a resubmission must take place, with all the overhead that this entails (transaction fees, storage costs, a new voting must take place, etc.). This is a deliberate design choice that guards the system against DoS attacks. From a practical perspective though, it means that the submitted UPs must be robust and thoroughly tested versions of the code, in order to avoid the resubmission overhead. We do not want to pollute the immutable blockchain history with intermediate trial-and-error UP events.

4 Activation

The final phase in the lifecycle of a software update, depicted in Figure 1, is the activation phase. This is a preparatory phase before the changes actually take effect. It is the phase, where we let the nodes do all the manual steps necessary, in order to upgrade to an approved UP and at the end, send a signal to their peers that they are ready for the changes to be activated. Thus, the activation phase is clearly a signaling period. Its primary purpose is for the nodes to signal upgrade readiness, before the actual changes take effect (i.e., activate).

Why do we need such a signaling period in the first place? Why is not the approval phase enough to trigger the activation of the changes? The problem lies in that there are manual steps involved for upgrading to the new software, such as downloading and building the software from source code, which entail delays that are difficult to foresee and standardize. This results into the need for a synchronization mechanism between the nodes that upgrade concurrently. The lack of such a synchronization between the nodes, prior to activation, might cause a chain split. This synchronization

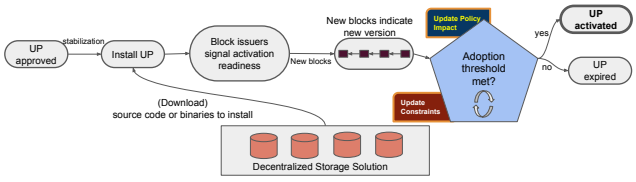
mechanism exactly is the activation phase and for this, it is considered very important.

Clearly, the activation phase is not aimed as a re-approval phase for the UP. It is there to allow a smooth incorporation of the software update into the network. Therefore it becomes relevant only for those UPs that impact the consensus and can risk a chain split. For UPs that don't impact the consensus (e.g., a code refactoring, or some short of optimization, or even a change in the consensus protocol rules, which is a velvet fork [7]) there is essentially no need for an activation phase and the change can activate, as soon as the software upgrade takes place.

Traditionally, when a software update needs to be activated and it is known that it is likely to cause a chain split, a specific target date, or better, a target block number is set, so that all the nodes to get synchronized. Indeed, this is a practice followed by Ethereum [3]. All major releases have been announced enough time before the activation, which takes place when a specific block number arrives (i.e., the corresponding block is mined). All nodes must have upgraded by then, otherwise they will be left behind. In Bitcoin [6], there also exists a signaling mechanism[2]. In this case, the activation takes place, only if a specific percentage of blocks (95%) within a retargeting period of 2016 blocks, signal readiness for the upgrade.

Once the UP approval result has been buried under a sufficient number of blocks (i.e., the stabilization period passes), then the activation period is initiated. In Figure 5, we depict the activation period in the decentralized setting.

Figure 5: The activation phase.



The first step in the activation phase is the installation of the software update. Typically, as soon as the UP approval is stabilized in the blockchain, the GUI of the client software (e.g., the wallet) prompts the user to download and install the update, using the link that accompanies the UP. If in the UP bundle there exist an approved binary, then the user can download and install this, otherwise the user must download the approved source code. In the latter case, there exist an extra step of producing the binary code from the source code. In any case, it is important to note that the new software is just installed but not activated. It will remain in a latent state until the actual activation takes place.

For the nodes participating in the consensus protocol the installation of a software update means that they are ready to activate, but wait to synchronize with their other peers. To this end, they initiate signaling. This means that every new block issued will be stamped with the new version of the software, signifying their readiness for the new update.

When the first block with the new version appears, we enter the adoption period for the specific UP. During the adoption period the following conditions have to be met, in order for the activation to take place: a) The number of blocks with a signal must exceed a specific threshold, b) the update constraints for the specific UP must be fulfilled and c) the adoption time period must not be exceeded, otherwise the UP will become expired.

The blocks generated in a proof-of stake protocol are proportional to the stake and therefore, we can assume that the signaling mechanism is also proportional to the stake. We can also assume that the honest stake majority, will follow the protocol and eventually will upgrade and thus signal this event with their generated blocks. This means that the minimum expected percent of signals (i.e., the activation threshold) cannot be other that the minimum percent of honest stake majority required by the proof-of-stake consensus protocol. Of course, as we have noted above, for changes that don't impact the protocol, the activation threshold could be zero; meaning that even if only one node upgrades, then the changes can be immediately activated.

Moreover, the adoption time period is not fixed for all UPs. It varies based on the type of the change, which is something recorded in the UP metadata. One size does not fit all, and this is indeed true for the adoption time period of UPs. For example, major updates that require a lot of manual steps, or significant build time, or even hardware upgrade, should be adopted in a sufficient period of time, while small updates should be activated more swiftly.

Finally, before the actual activation of a change, the validation of all the update constraints must take place once more. This is true, although we make the assumption that from the approval phase all relevant update constraints' issues (like conflicts, or dependencies) have been considered. The fact that the adoption period might require significant time for a UP, whose update constraints were fulfilled at the approval phase, while concurrently there are other UPs that become activated, means that the conditions might have changed and the update constraints must be reevaluated to make sure that no problems will arise upon activation of a UP.

3 UPDATE GOVERNANCE

With the term *update governance* we mean the processes used to control the software updates mechanism. As we have seen, in the centralized setting, update governance is quite straightforward. The mere existence of a central authority

[2]see BIP-9 at https://github.com/bitcoin/bips/blob/master/bip-0009.mediawiki

(owner of the code) simplifies decision making significantly. On the other side, in the decentralized approach, we have seen that all decision-making procedures have been replaced by a voting process, where a decision is taken collectively by the whole stake. Thus, we are dealing with a *decentralized governance model*.

It is true that in the decentralized setting the tools that one has at one's disposal for controlling the software updates process, but more importantly, for leading the participating stakeholders towards an *update consensus*, i.e., to reach at an agreement regarding the acceptance, or not, of the updates and the order (i.e., priorities), by which these will be applied, are the voting and delegation mechanisms. In this section, we describe both of these mechanisms and end with a description of the software updates protocol that encompasses them.

1 Voting for Software Updates

1.1 Voting for SIPs and UPs. Voting is the main vehicle for driving democracy and in our case is indeed the main mechanism for decision making in a decentralized setting for software updates. During the lifecycle of a software update, we have identified two phases, where a voting process is required. The first appears in the ideation phase and has to do with the approval of SIPs, while the second lies in the approval phase and deals with the approval of submitted source code and metadata (optionally binaries also), namely the UP bundle. Apart from the fact that the object of the voting process differs in these two phases, everything else is the same, so in the text that follows, whatever we describe pertains to both of the voting phases, unless explicitly specified.

A vote is a fee-based transaction that lives for a specific period of time called the *voting period*. In particular, after the object of voting (a SIP or a UP) has been submitted to the blockchain (initially encrypted and then at a second step revealed - based on a commit-reveal scheme),then the voting period for this software update begins.

We acknowledge the fact that not all software updates are equal and therefore, we cannot have a fixed voting period. Moreover, software updates (especially UPs, which are essentially source code) are complex technical objects, where sufficient time must be provided for their evaluation. Therefore, the voting period must be adaptive to the complexity of the specific software update. We propose to have a *metadata-driven* voting period duration, based on the size of the software update, expressed as required man-effort in the accompanying metadata of the software update. So the voting period duration $v_{duration}(su_{size})$, which can be expressed as a number of blocks (or slots), is a function of the software update size (e.g., expressed in man-days).

We introduce a *vote* as a new transaction type, the *Software Update Vote Transaction (SUVT)* that can only be included in a block during the voting period. The core information conveyed by the vote transaction is summarized in the following tuple:

$$\begin{aligned}(&H(< SIP/UP >),\\ &SU_{Flag},\\ &< confidence >,\\ &vks_{source},\\ &\sigma_{sks_{source}}(m))\end{aligned}$$

$H(< SIP/UP >$ is the hash of the content of the SIP/UP and plays the role of the unique id of a software update. SU_{Flag} is a boolean flag (SIP/UP), which discriminates an SIP vote from an UP vote. $< confidence >$ is the vote per se, expressed as a three-valued flag (for/against/abstain). vks_{source} is the public key of the party casting the vote. Finally, $\sigma_{sks_{source}}(m)$ is the cryptographic signature, signed with the private key corresponding to vks_{source}, on the transaction text m.

Anyone who owns stake has a legitimate right to vote and this vote will count proportionally with the owned stake. Furthermore, as we will describe in detail in the next section, the right to vote can also be delegated to another party. However, we want to give the power to a stakeholder to override the vote of his/her delegate. Therefore, if within the same voting period appear both a private vote and a delegate's vote, for the same software update, then the private vote will prevail.

Moreover, as we have stated above, we acknowledge the fact that software updates are complex entities and their evaluation is a challenging task. This is true especially, when someone has to evaluate source code. Therefore, we cannot exclude the possibility that a voter/evaluator changes his/her mind after voting for a specific SU (for example, the evaluator identifies a software bug after voting positively for a software update). We want to provide the flexibility to the evaluators to change their minds. Therefore, we allow for a voter to vote multiple times, within a voting period for a specific software update. At the end, we count only the last vote of a specific public key in the voting period for a specific software update.

1.2 Voting Results. After the end of the voting period for a specific software update and after we allow some stabilization period (in order to ensure that all votes have been committed into the blockchain), the votes are tallied and an outcome is decided. First, the private votes are counted. The tallying is performed as follows: For each slot within the voting period, if a block was adopted pertaining to that slot, each SUVT in that block is examined. If the staking key for that transaction has been tallied on a private vote previously, the previous vote is discounted and the new vote is counted. This allows voters to modify their votes until the end of the voting period. For every SUVT which has been counted, the stake that votes for it is summed and this constitutes the *private stake in favour*, the *private stake against* and the *private abstaining stake*.

Subsequently, the delegated votes are counted. For each slot within the voting period, if a block was adopted pertaining to that slot, each SUVT in that block is examined. If the

delegatee staking key for that transaction has been tallied on a delegatable vote previously, the previous vote is discounted and the new vote is counted. For each of the delegatable votes, the keys delegating to it are found. Each of the keys delegating is checked to ensure that the delegating key has not cast a private vote; if the delegating key has also cast a private vote, then the private vote is counted instead of the delegated vote. For each delegatable SUVT which has been counted, the stake delegating to it which hasn't issued a private vote is summed and this constitutes the *delegated stake in favour*, the *delegated stake against*, and the *delegated abstaining stake*.

The sum of the private stake in favour and delegated stake in favour forms the *stake in favour*; similarly, we obtain the *stake against* as well as the *abstaining stake*.

Assuming that the *honest stake threshold assumption* of our software updates protocol is h, then at the end of the tallying, a software update (i.e., a SIP or an UP) is marked one of the following:

- *Approved*. When the $stake\ in\ favour \geq h$
- *Rejected*. When the $stake\ against \geq h$
- *No-Quorum*. When the $abstaining\ stake \geq h$. In this case, we revote (i.e., enter one more voting period) for the specific software update. This revoting can take place up to $rv_{no-quorum}$ times, which is a protocol parameter. After that, the software update becomes *expired*.
- *No-Majority*. In this case none of the previous cases has appeared. Essentially, there is no majority result. Similarly, we revote (i.e., enter one more voting period) for the specific software update. This revoting can take place up to $rv_{no-majority}$ times, which is a protocol parameter. After that, the software update becomes expired.
- *Expired*. This is the state of a software update that has gone through $rv_{no-quorum}$ (or $rv_{no-majority}$) consecutive voting periods, but still it has failed to get approved or rejected.

As you can see, in our proposal we have chosen a three-value logic for our vote (for/against/abstain). In this way, apart from the actual result, we can extract the real sentiment (positive, negative, neutral) of the community for a specific software update. This is very important in a decentralized governance model, because it clearly shows the appeal of a software update proposal to the stakeholders. If we did not allow negative votes, then the negative feeling would be hidden under the abstaining stake.

Moreover, the "abstain" vote is a sign that the stakeholder has not formed an opinion for, or against, a specific software update. However, it might also denote that the evaluator has not made yet a decision and maybe he/she needs more time. Therefore, the abstain vote can be also used as a way for the evaluator to say that the evaluation of the SU has not finished, a conclusion can not be drawn yet and indirectly submit a request for a time extension (i.e., a new voting period).

1.3 On Assumptions and Thresholds. We require that for each voting period and for every software update submitted, there are always at least h honest parties, actively participating in the voting process (i.e., submitting a vote with a value of for/against/abstain), where h is the *honest stake threshold assumption* of our software updates protocol. In order to achieve this high degree of availability for our protocol, we leverage delegation to *stake pools*, or *expert pools*, which is a topic that we describe in detail in the next section. Therefore, our software update protocol (and especially the voting phases), similarly (in-concept) to the underlying consensus protocol, make the assumption that h percent of honest stake is being active at all time.

If the underlying consensus protocol has an honest stake threshold x (e.g., $x = 51\%$), then we require $h \geq x$. The rationale behind this is that the software updates protocol is built *on top* the consensus protocol, since all the software updates protocol events are essentially transactions committed into the blockchain. Therefore, the software updates protocol cannot require less honest stake than the consensus protocol on which it is based. However, there might be cases (i.e., software updates), where a greater majority is required for the voting process and therefore a greater percent of honest stake h might be used for the software updates protocol.

Moreover, the stake considered during the tally is the stake that the voters have at that moment. That is, we only consider the stake of the voters at the moment of the tally, without taking into account the stake that the voters had in the moment that the votes were casted. Therefore, the stake distribution is not known at the moment where the voting takes place, which is a security measure against voters' coercion.

With respect to the voting threshold, we have seen that for a software update (either a SIP, or an UP) to get approved the following condition should hold: $stake\ in\ favour >= h$. Our voting mechanism, from a security perspective, has essentially two goals: a) a software update that is not approved by the stake majority will never be applied and b) a software update that it is approved by the stake majority will be eventually applied.

Let's assume that we imposed another threshold:

$$stake\ in\ favour \geq h + d$$

, where $d > 0$. We know that we have at least h honest parties actively voting. Then, since we need more that h votes for the SU to get approved, then the adversaries could block the approval, either by voting 'against', or 'abstain', or by not voting at all.

Similarly, if we imposed as a threshold

$$stake\ in\ favour \geq h - d$$

, where $d > 0$, then the adversaries could potentially approve a malicious SU (assuming that we have $h-d$ adversary stake), if the honest stake does not vote a unanimous rejection (i.e., some part of the honest stake votes "against" and the rest part votes "abstain").

In other words, the *stake in favour* $\geq h$ threshold that we have chosen, guarantees that the adversaries cannot block a good software update, since there is enough honest stake majority to approve it. Moreover, due to the *liveness* property of the underlying consensus protocol, all honest parties' votes will be eventually committed to the blockchain, as long as the tallying takes place after a stabilization period. At the same time, the adversaries cannot approve a malicious software update, since they do not have the majority to approve a malicious software update.

2 Delegation

Each stakeholder has the right to participate in the software updates protocol of a proof-of-stake blockchain system.This participation entails: the submission of update proposals (in the form of SIPs and later as UPs), the approval, or rejection, of SIPs or UPs and last the adoption signaling, when an upgrade has taken place. In this section, we discuss the delegation of the protocol participation right to some other party. As we will see next, this delegation serves various purposes and copes with several practical challenges.

2.1 Delegation for Technical Expertise. One of the first practical challenges that one faces, when dealing with the decentralized governance of software updates is the requirement of technical expertise, in order to assess a specific software update proposal. Indeed, even at the SIP level, many of the software update proposals are too technical for the majority of stake to understand. Moreover, during the UP approval phase, the approver is called for approving, or rejecting, the submitted source code, which is certainly a task only for experts.

Our proposal for a solution to this problem is to enable delegation for technical expertise. Stakeholders will be able to delegate their right to participate in the update protocol to an *expert pool*. The proposed delegation to an expert pool comprises the following distinct responsibilities:

- The voting for a specific SIP
- The voting for a special category of SIPs
- The voting for any SIP
- The approval of a specific UP
- The approval of a special category of UPs
- The approval of any UP

As you can see, we distinguish delegation for voting for a SIP document and that for approving an UP. We could have defined delegation for SIP voting to imply also the approval of the corresponding UP. However, since both have a totally different scope, there might be a need to delegate to different expert pools for these two. Indeed, a SIP is an update proposal justification document and the expert who is called to vote for, or against, a specific SIP, must have a good sense of the road-map of the system. On the contrary, the approval of a UP is a very technical task, which deals with the review and testing of a piece of code against some declared requirements (i.e., the corresponding SIP) and has nothing to do with the software road-map.

We allow voting for a specific SIP/UP, or for *any* SIP/UP. In the former case, the id of the specific SIP/UP must be submitted along with the delegation. In the latter case, if one wants to override the "any" delegation and delegate to some other party for a specific SIP/UP, then this is possible via the submission of a new delegation for the specific SIP/UP.

There is also delegation for special categories of SIP/UPs, as well as the delegation for the adoption signaling. These will be the topics that we will describe next.

2.2 Delegation for Specialization. One size does not fit all and surely, all software updates are not the same. There are many different angles (a.k.a. dimensions) by which, one can view software updates and distinguish them into different categories. For example, the reason for a SU can be such an angle (i.e., dimension). A SU can be a bug-fix (or security fix), or it can be a change request (or a new feature request). Another dimension is the priority (high/medium/low) of the SU. Furthermore, in blockchain systems, a typical dimension for distinguishing SUs is the impact to the consensus protocol (impact/no-impact). Moreover, for those SUs that impact the consensus protocol, the type of change that they trigger (soft/hard fork) is another dimension. For those SUs that do not impact the consensus protocol another dimension could be, if they are platform specific (e.g., applicable only to Linux, Windows, MacOS etc. versions of the client software).

All these are valid ways to categorize software updates to certain categories. However, special categories might justify some specialized treatment. We do not propose any specific set of categories in this paper. However, we do propose that: a) software updates are tagged with a specific category and b) to use delegation for enabling specialized treatment on special categories.

Let us consider for example security fixes. It is common sense, that security fixes are software updates that: a) have a high priority and b) require significant technical expertise to be evaluated. Therefore, by having a special expert pool as a *default delegate* for this category of software updates (both SIPs and UPs) enables: a) a faster path to activation and b) sufficient expertise for the evaluation of such SUs.The former is due to the omission of the delegation step in the process and that the evaluation (i.e., voting of SIPs/UPs) will take place generally in shorter times; exactly because this is a specialized and experienced expert pool that deals only with security fixes; we assume that they can do it faster than anybody else.

So, for software updates with a special tag, our proposal is, to have default specialized expert pools that will participate in the software updates protocol on behalf of the delegated stake. Of course, this default delegation based on SU tagging can be overridden. Any stakeholder can submit a different delegation for a specific SIP/UP regardless of its tag. What about software updates without a special tag? These will be the topic of discussion of the next subsection.

2.3 Default Delegation for Availability. Blockchain protocols based on the Proof-of-Stake (PoS) paradigm are by nature dependent on the active participation of the digital

assets' owners –i.e., stakeholders– (Karakostas et. al. [5]). Practically, we cannot expect stakeholders to continuously participate actively in the software updates protocol. Some users might lack the expertise to do so, or might not have enough stake (or technical expertise) to keep their node up-and-running and connected to the network forever.

One option to overcome this problem, which is typical in PoS protocols, is to enable stake representation, thus allowing users to delegate their participation rights to other participants and, in the process, to form "stake pools"([5]). The core idea is that stake pool operators are always online and perform the required actions on behalf of regular users, while the users retain the ownership of their assets ([5]).

In this paper, we propose to utilize the stake pools mechanism for our software updates protocol in tandem with the consensus protocol. In particular, we propose to allow each stakeholder to define a default delegate for participating in the software updates protocol from the list of available stake pools that participate in the core consensus protocol. This will be a "baseline" representative of each stakeholder to the software updates protocol, just for the sake of maintaining the participation to the protocol at a sufficient level and minimizing the risks of non-participation. This delegate will coincide with the delegate for the participation in the consensus protocol. A stakeholder will be able at any time to override this default delegation. A delegation to an expert pool for a specific software update, or a specific category of software updates, due to specialization, described in the previous section, will override the default delegation to a stake pool.

Therefore, for each stakeholder, all software updates without a special category tag that have not been explicitly delegated to some expert pool, will be *by default* delegated to the same stake pool that the stakeholder has delegated to run the proof-of-stake consensus protocol. This will allow the stakeholder to be able to abstain from the software updates protocol, for periods of time, without causing a problem. This also has another suitable consequence that we discuss next.

We have seen that one of the responsibilities of a stakeholder that participates in the software updates protocol, at the activation phase, is to signal the adoption of a UP. This signal is placed within each generated block after the node upgrades. We know that in a proof-of-stake consensus protocol blocks are generated by the stakeholders (with possibility proportional to their stake) and in a more realistic setting by the stake pools, which have been delegated to do so. So in practice, the stake pools are called for signaling the adoption event, merely because they are the block issuers. Therefore, the choice of stake pools as the default delegates for the software updates protocol, fits nicely with the fact that stake pools will be responsible for signaling adoption anyway. If a stakeholder has not delegated his participation to the proof-of-stake consensus protocol and is chosen to generate a block, then he/she will also be responsible for signaling the adoption of a specific UP. In other words, adoption signaling simply follows the delegation of the proof-of-stake consensus protocol and ignores delegation to expert pools.

2.4 Delegation Mechanics. For the realization of the stake pool delegation mechanism that we described above, we closely follow the work of Karakostas et al. [5], so we refer the interested reader to this work for all the relevant details. In this subsection, we would like to focus on the most basic mechanics (i.e., technical details) that will enable such a delegation mechanism to work. Please note that many of our ideas are based on the design of the delegation mechanism for the Cardano blockchain system [4].

Staking keys. Following the Karakostas et. al. [5] approach, we separate for each address the control over the movement of funds (i.e., executing common transactions, such as payments) and that over the right for participation in the proof-of-stake protocol and consequently, in the software updates protocol, due to the ownership of stake. Intuitively, this separation of control is necessary, since we only want to delegate the management of stake to some other party, by means of participation in the software updates protocol and not the management of the funds owned by this stake. This is achieved in practice by assuming that each address consists of two pair of keys: a) a *payment key pair* $K^p = (skp, vkp)$ and b) a *staking key pair* $K^s = (sks, vks)$. With the former a stakeholder can receive and send payments, while with the latter a stakeholder can participate in the proof-of-stake consensus protocol and in the software updates protocol. skp and sks are the secret keys for signing, while vkp and vks are the public keys used to verify signatures.

Stake Delegation. In its simplest form, delegation of stake from some party A to another party B (typically a stake pool) for participation in the proof-of-stake consensus protocol, also delegates the right for participation in the software updates protocol as well. The rationale of this has been described in subsection 2.3 and it holds only on the supposition that there is no explicit delegation to some expert pool. So in the rest of this text, when we refer to stake delegation, we mean for the participation in both the proof-of-stake consensus protocol and the software updates protocol, unless an explicit statement is made for delegation to an experts pool.

At its core, the delegation of stake to some other party, essentially requires two things: a) stake registration and b) issuance of a delegation certificate:

Stake key registration. This step is a public declaration of a party that it wishes to exercise its right for participation in the proof-of-stake protocol, due to its ownership of stake. In order for a stakeholder to exercise these rights, he/she must first issue a stake key registration certificate. This is a signed message stored in the metadata of a transaction and thus it is published to the blockchain. The key registration certificate must contain the public staking key vks, and the signature of the text of the transaction m by the staking private key sks, which is the rightful owner of the stake. In other words, the key registration certificate r is the pair: $r = (vks, \sigma_{sks}(m))$. The signature σ of the certificate, authorizes

the registration and plays the role of a witness. Symmetrically, there is also a de-registration certificate for a stake key, which is a declaration that a party no longer wishes to participate in the proof-of-stake protocol.

Delegation registration. In order to register the delegation of stake from one party (source) to another (target), a delegation certificate must be issued and posted to the blockchain by the source party. This certificate publicly announces to the network that the source party wishes to delegate its stake right (for participation in the proof-of-stake protocol) to the target party and this is recorded forever in the immutable history of the blockchain. At a minimum, a delegation certificate consists of the following information:

$$(H(vks_{source}), H(vks_{target}), \sigma_{sks_{source}}(m))$$

Where, $H(vks_{source})$ is the hash of the source party's public staking key, $H(vks_{target})$ is the hash of the target party's public staking key and $\sigma_{sks_{source}}(m)$ is the signature of the text m of the transaction (within which the delegation certificate is embedded) by the source party's private staking key sks_{source}, which authorizes the certificate and plays the role of a witness.

If at some point, the source party wishes to re-delegate to some other party, or even to participate in the protocol on its own, then it must simply issue a new delegation certificate. For self-participation in the protocol, a party must issue a delegation certificate to its own *private stake pool*[3]. If the source staking key is de-registered, then the delegation certificate is revoked.

Delegation to an Expert Pool. We have seen that by default, the participation right in the software updates protocol is delegated to the stake pool that the delegation for participation in the proof-of-stake consensus protocol has taken place. So by default, some stake pool will participate in the software updates protocol. Next, we will discuss the case where a stakeholder wants to override the default behavior and explicitly delegate to an expert pool.

An expert pool is an entity consisting of one or more experts, who are willing to participate in the software updates protocol as delegates of other stakeholders. Their main task is to vote for (or against) SIPs and to approve (or reject) UPs. We call them "experts", because they need to have sufficient technical expertise, in order to evaluate a software update.

In order to enable delegation to an expert pool, we extend the delegation certificate presented above, with additional information. In particular, a delegation certificate to an expert pool is defined as the following tuple:

$$\begin{aligned}(&H(vks_{source}),\\ &H(vks_{target}),\\ &\sigma_{sks_{source}}(m),\\ &SU_{Flag},\\ &H(< SIP/UP >),\\ &< category >)\end{aligned}$$

In this case, the $H(vks_{target})$ is the hash of the public staking key of the expert pool. We have extended the delegation certificate to include a boolean flag SU_{Flag}, which denotes, if the delegation pertains to a SIP, or an UP. We have explained previously (see subsection 2.1), the rationale for distinguishing the delegation for these two. Finally, the hash $H(< SIP/UP >)$ is the hash of the content of the SIP, or UP, in question and plays the role of the unique id for this SIP, or UP respectively. Note that if instead of a specific SIP/UP id, a special value is provided for this field (e.g., '*'), then this corresponds to a delegation for *any* SU of this type (SIP or UP). Finally, if the SU id field is empty (or $NULL$, it depends on the implementation), then we take into account the last field, which specifies the *category* of the SU (e.g., "security-fix", "linux-update", etc.) that, we wish to delegate for. This will be a simple string value chosen from a fixed set of values (a list of acknowledged SU categories).

In summary, with this certificate, a party can delegate its participation right in the software updates protocol to an expert pool: a) for a specific software update (SIP or UP), b) for a specific category of software updates, or c) for any software update. Of course, in order for this delegation registration to be valid, the target expert pool must have been appropriately registered first in the blockchain. This is the topic to be discussed next.

Expert Pool Registration. In order for someone to publicly announce his/her intention to play the role of an expert, or equivalently, to run an expert pool, two things are required: a)to issue an expert pool registration certificate and b)to provide appropriate *metadata* describing the expert pool.

Expert pool registration certificate. The certificate contains all the information that is relevant for the execution of the protocol. At its most basic form this certificate comprises the following:

- vks_{expool}: This is the public staking key of the expert pool. This must be used as the target public key in the delegation certificate, as discussed in the previous subsection.
- $(< URL >, H(< metadata >))$: A URL pointing to the metadata describing the expert pool and a content hash of these metadata. The URL points to some storage server and the hash of the content retrieved must match the one stored in the certificate for the pool registration to be considered as valid.

[3] A *private stake pool* is a trivial case of a stake pool. By treating self-staking as a special case of stake pool delegation is a design decision for the sake of simplicity [4].

- $\sigma_{sks_{expool}}(m)$: The certificate must be authorized by the signature σ of the expert pool sks_{expool} on the text m of the transaction that includes the certificate.

Symmetrically, there should be also an *expert pool retirement certificate* for allowing an expert pool to cease to operate. This should include the public staking key of the expert pool, as well as a time indication (e.g., expressed in block number, or an epoch number etc.) of when the pool will cease to operate.

Expert pool metadata. The expert pool metadata are necessary information that describe sufficiently an expert pool, so as the stakeholders community can decide, which expert pool to choose for their delegations. Typically, this information will be displayed by the wallet application, in order to assist the users to select the expert pool of their choice. Examples of useful information to be included in the expert pool metadata are the name of the pool, a short description, the area of expertise, the years of expertise, preferences to specific SU categories, URLs to sites that exhibit the claimed experience and in general any information that can help the stakeholders to choose the appropriate delegate for the right software update.

Miscellaneous Considerations.

Chain delegation. Chain delegation is the notion of having multiple certificates chained together, so that the source key of one certificate is the delegate key of the previous one. In principle there is no reason to prevent the formation of delegation chains. However, an implementation of this proposal must take into account the possibility to form (deliberately or by accident) delegation cycles. This means that a target delegate ends up to be one of the sources. In this case, the delegation is essentially canceled and the system should detect it and prevent it pro-actively.

Certificate replay attacks. For all our certificates, namely: stake key registration, delegation registration and expert pool registration, we have provided signatures of the text of the encompassing transaction (the certificates are included as transaction metadata), signed by the party(ies) authorized to issue the certificate. This is a design choice made in [4] that prevents against a certificate replay attack. In this attack, an attacker re-publishes an old certificate, in order for example to change a delegation to a new expert pool. In particular, since the certificate includes a signature on a specific transaction text, then this certificate is bound forever with the specific transaction, and just like in blockchains with a UTxO accounting model, a transaction cannot be replayed (a UTxO can be only spent once), similarly the specific certificate cannot be replayed either. For account based blockchains there are other approaches that one can follow, in order to prevent a replay attack, such as the *address whitelist* proposed in Karakostas et. al. [5], where the transaction that includes the certificate must be issued from a specific whitelisted address. Of course there are other common solutions like the counter-based mechanism (known as the *nonce*) used in Ethereum [3].

Identity theft. Significant expertise on difficult technical issues is not a skill that is easy to acquire. Moreover, experience comes after a long period (probably many years) of struggle with technical issues. Therefore, truly experienced specialists on a technical domain are hard to find and for this reason they are invaluable. Typically, these experts are well-known and well-respected figures in the community. Therefore, such a well-known expert is expected to receive a significant amount of delegations, if he/she chooses to register an expert pool. This fact makes expert pool registration susceptible to identity theft. This is the case where an expert pool falsely claims to be the famous "expert A", just for the sake of receiving the delegations drawn from the fame of the expert. This identity assurance problem is external to the software updates protocol and also to the underlying consensus protocol and thus some out-of-band solution could be exploited for dealing with it. For example, a famous expert, could post his/her public key (or its fingerprint) to social media, so that the people who follow him/her, will know, which is the genuine key that they can delegate to. Of course, other similar in concept, solutions can be exploited as well. However at the end of the day, in a decentralized setting, it is the stake via delegation that will be the ultimate judge of an expert pool.

4 UPDATE LOGIC

We have pointed out many times that software updates are not all the same. There are many different perspectives for viewing SUs, which call for a specialized *software update policy*. Software updates can be distinguished by the following dimensions:

- Reason of change (a bug-fix versus a Change Request).
- Priority of change (high/medium/low).
- Size of change (typically in man-effort required to be implemented).
- Impact of change (e.g., impacts the consensus protocol or not).
- Type of (protocol) change (hard/soft/velvet fork)
- Platform-specific change (Linux, Windows MacOS etc.)

To this end, we propose a *software update logic* that is *metadata driven*. A "logic" that can distinguish one SU from another and apply the appropriate update policy. In the section, we describe our proposal for achieving such a decentralized metadata-driven, software update mechanism.

1 Update Proposal Metadata

An Update Proposal is inherently accompanied by meta-information that describes the proposal and sets it into the appropriate context. Therefore, we could say that every Update Proposal comprises a rich set of update meta-data that ultimately should drive the whole upgrade process.

The update meta-data provide basic information such as the name and a description for the Update Proposal. Moreover, they declare the urgency of the update as well as its type (Change Request or bug-fix), in order to guide the prioritization of the update. They provide values for critical parameters of the Update protocol (e.g., the `voting_duration`), as well as declare the type of consensus rule changes (hard fork, soft fork, velvet fork etc.). Finally, they declare the dependencies and the potential conflicts with other Update Proposals, which is a very useful information that should guide the software deployment process. This metadata-driven update process is based on the notion of *Update Policy*, which we will discuss further in the following subsections. A complete list of our proposed list of Update Proposal meta-data can be found in the Appendix.

2 The Activation Phase Revisited

Activation Phase. the activation phase is not a re-approval phase, it is just there to guard against chain split.s

It is not relevant to UPs that dont impact consensus (adoption threshold = 0%) For those that do impact, adoption threshold = the honest stake threshold assumption of the consensus protocol. We make the assumption that if a UP is approved then all honest stake will eventually upgrade.

2.1 The Activation Lag. To mitigate risk of chain split by accident (activation too early), we propose the concept of the *activation lag*.

2.2 On Activation Thresholds.

3 Update Policies

3.1 Update Policies Realized. An update policy is a way to customize the activation speed of a SU based on the âĂIJtypeâĂİ of the SU, which is deduced by the SUâĂŹs metadata. We want to follow a metadata-driven activation approach. An update policy can be enabled by two things: A) Delegation to expert pools B) adoption threshold (activation phase) and activation lag (activation phase) Activation lag is determined by Deployment complexity Soft/Hard fork type of change

3.2 Update Constraints.

Conflict Resolution. How do we ensure that multiple concurrent requests for updates are handled simultaneously in a way that

- conflicts are resolved and the adopted updates are consistent,
- so that no contradictory updates are to be deployed at the same time
- Community splits over controversial updates are avoided

Dependencies. How do we impose respect for update dependencies, so that the system reaches a consistent state?

1.Nikos: Maybe we could use the version_from field from the metadata for this purpose. An Update Proposal cannot be applied if the version requirement that it poses (version_from) is not the current version

Update Policies per type of updates – old stuff. An *Update Policy* is defined as the pair *(Speed of Activation, Method of Deployment)*. We need to differentiate the speed of deployment and the method of deployment based on: a) the type of change (bug -fix or change request), b) the part of the system that is affected by the change (consensus rules impact, or only software impact), c) the urgency of the change (severity level) and d) soft vs. hard forks

2.Nikos: We have 3 levels of speed: high, medium and low. What are our different deployment methods?

- How do we discriminate between different types of updates (e.g., software vs. protocol, bug-fix vs. change request)
- We need to provide a different deployment path for each Type of Update. For example, critical hot-fixes might need to bypass some of the governance steps.
- We need to incorporate into our update logic the notions of bug Severity and "Required Speed of Deployment"

Also, deployment/activation time must vary according to the type of change. We see the following "change categories":

- Bug-fix vs. Change Request
- Consensus Protocol impact vs. No Impact
- High severity vs. Low severity

3.3 Rollbacks. How can we smoothly rollback an update, in the case of a problem?

5 SECURITY ANALYSIS

A cryptographic modeling of our software updates protocol, in order to prove its security properties. Nikos: do we need something like that?

From a security perspective, our proposal must ensure that: a) any stakeholder will always be able to submit an update proposal to be voted by the stakeholders community, b) an update proposal that is not approved by the stake majority will never be applied, c) an update proposal that it is approved by the stake majority will be eventually applied and d) it will provide guarantees for the authenticity and safety of the downloaded software.

6 LEDGER ENHANCEMENTS

1 Recording the History of Updates

The historical tracking of software updates is very important. This is not only for the obvious reasons of reporting and statistical analysis but also for troubleshooting. Maintaining a log of all SUs is critical, when something has gone wrong and we want to investigate the root-cause. We use the ledger as the single version of the truth for software updates. But why is this necessary? Why we could not just use some off-chain logging mechanism to record all update events?

In a centralized setting, a simple logging mechanism would be enough. However, software updates in a decentralized setting, as we have seen, have a decentralized governance mechanism. In this case, we do not need only to record what software updates and when, these have been applied, but also

how and when, they have been approved by the community. Moreover, this historical fact must be an immutable event that cannot be tampered with, because that would alter the evolution history of the system. Finally, in the decentralized approach there is no central owner of the code and thus, this history cannot be under central control. The historical records must be stored in some repository, which must be inherently distributed. So, for these reasons, we propose the ledger as the home for the events generated in the lifecycle of a software update.

To this end, we enhance the ledger with new types of events to be permanently recorded in the immutable history of updates. Moreover, we propose the maintenance of some local stake for software updates. This stake will provide necessary information to the update protocol without the need to access old blocks for reading individual events. This state can be calculated anytime from the events stored on the blockchain and therefore there is no need to store it on-chain. In this section, we will follow the course of a software update through out its lifecycle, as this has been described in the previous section, and describe in detail the events that will be generated along the way and those that need to be stored on-chain, in the distributed ledger, as well as to be maintained as local state.

2 Software Improvement Proposals

A software update starts its life with the issuance of a *SIP document* by a stakeholder. This proposal must achieve the consensus of the majority of stake, through a voting mechanism, in order to become adopted...

2.1 Submission.

2.2 Voting.

2.3 Local State.

3 Update Proposals

3.1 Submission.

3.2 Voting.

3.3 Local State.

4 Activation Events

4.1 Signals.

4.2 Adoption.

4.3 Local State.

Update Proposals - Old stuff

An Update Proposal can be issued by any party that owns stake. The party first creates the Update Proposal, hashes it, and uploads it to a decentralized storage service (to be discussed in later sections). Subsequently, they place the Update Proposal on the Blockchain.

We introduce a new type of transaction, an *Update Issuance Commit Transaction (UICT)*, which commits the Update Proposal to the blockchain. Since it is a transaction it will be included into a block and thus into the ledger. The UICT contains the following data:

commit: A salted commitment to the Update Proposal hash as well as the public saking key of the issuer.

```
commit = H(salt || pk || H(Update Proposal))
```

pk: The public key of the Update Proposal issuer.

sig: A signature on commit by the Update Proposal issuer public key pk

A UICT is broadcast to the network by the issuer. Upon receiving the UICT, the validators ensure that the signature sig verifies on the plaintext commit with public key pk and that the signing staking key pk satisfies a configurable minimum threshold of stake prior to relaying it or including it in their block. While relaying and inclusion is conditioned on threshold, if a UICT is included in a block generated by another miner, the block and UICT are accepted as valid regardless of stake. If a new UICT is broadcast pertaining to the same commitment and pk, it is rejected.

Once the UICT has been stabilized (by being buried under k blocks), the Update Issuer issues an *Update Issuance Reveal Transaction (UIRT)*, which reveals the Update Proposal hash to the blockchain and makes it available for download. The UIRT contains the following data:

proposal: The hash of the Update Proposal, i.e.,

```
H(Update Proposal).
```

salt: The salt previously used in the UICT.

pk: The public key of the Update Proposal issuer.

Upon receiving a UIRT, any validator ensures that a respective UICT has been included in the most recent 12k blocks. On the one hand, this number has to be large enough to allow for block stability and liveness. On the other hand, it must be small enough so that unclaimed UIRTs can expire and garbage collected to avoid UTXO pollution; in addition, it must be small enough to bound the stake-shift that could have occured from UICT issuance to UIRT issuance. The validator finds the respective UICT by calculating `commit = H(salt || pk || H(Update Proposal))` based on the data provided in the UIRT and looks for the commit value in the most recent 12k blocks. Furthermore, the validator ensures the pk in the UIRT matches the previously claimed pk in the UICT as well as within the UICT commit value.

The reason why the Update Proposal is issued on the blockchain in two stages, with UICT first and UIRT later, is so that the rightful author can claim authorship of a particular Update Proposal. If the Update Proposal hash were to be revealed immediately, a dishonest party could create a competing transaction, signing the same proposal with their own key to claim authorship. This follows a similar model to Namecoin name claims [**?**]. If multiple valid UIRTs pertaining to the same commitment are received, only the first is included in the blockchain. Only the first such transaction is necessary, as any two UIRTs will necessary contain the same data due to it having been committed in the UICT and

by the collision resistance property of the underlying hash function.

The slot at which the UIRT of an Update Proposal has been included is referred to as the *issuance-time* of the proposal. After a UIRT transaction is validated, the node automatically attempts to download the Update Proposal, by hash, from the Decentralized Storage service. The node hashes the downloaded content to ensure that the hash included in the blockchain matches the hash of the downloaded content.

Update Proposal Validation - Old stuff

3.Nikos: We need to define the rules for a legitimate Update Proposal For example, a legitimate Update Proposal cannot have a dependency with an Update Proposal who has not been activated yet. So the version requirement of an Update Proposal must be the current adopted version.

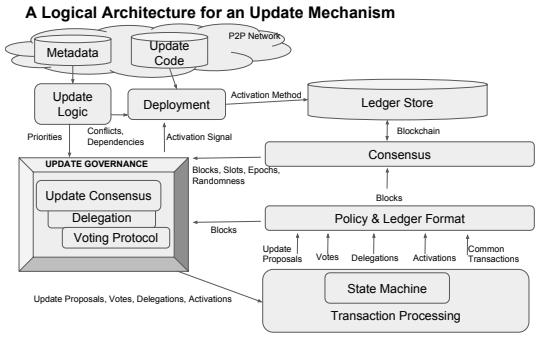
```
sw_version_from == current_sw_version
&&
prot_version_from == current_prot_version
```

4.Nikos: Just validating an Update Proposal when it is revealed is not enough. It has to remain valid also right before activation

7 A LOGICAL ARCHITECTURE FOR AN UPDATE MECHANISM

The proposed architecture must identify the core components of a decentralized update system and define how the various new events (Update Proposals, Votes, Delegations, Update Activations etc.) are incorporated into the blockchain system and finally describe the interactions between these components.

Figure 6: A Logical Architecture for a Decentralized Update Mechanism



In figure 6, we depict a logical architecture that incorporates a software update mechanism in a distributed ledger. In this figure, we can see that the set of events flowing into the ledger system has been enhanced. Apart from the traditional transactions, which deal with the transfer of value between stakeholders and trigger changes to the global state, now we can see other events such as *Update Proposals*, *Votes*, or even *Vote Delegations* and *Activation events* being generated from the *Update Governance* component and entering the ledger system.

Our primary goal is to store all these new events that pertain to the update mechanism, within the ledger itself, as an immutable record of historical events. To this end, these new events are transformed to special types of transactions and as such, have to go through the rigorous transaction validation process, performed by each node of the network. This is depicted at the Transaction Processing component, where all the logic for what makes a legitimate transaction is implemented.

Ordinary transactions change the global state of value distribution but in the case of smart contracts [**?**] can also change the state of smart contract persistent data by triggering the execution of smart contract code. Update events do not change the global state in the same sense; however, one can imaging an "Update Proposal state", representing the number of Update Proposal submitted by a user. Similarly, one could also define a "Voting state" representing the votes gathered by each proposal.

Transactions are included into blocks by the Policy and Ledger Format component and these newly generated blocks are passed on to the next layer. The consensus component is where we achieve consensus on what block is legitimate and which blockchain branch is the legitimate one and a new block can be attached to it safely. Regardless of the "type" of the consensus protocol (either proof-of-work, or proof-of-stake) the consensus rules that guide the block validation logic must be enhanced to incorporate the new events (i.e., transaction types) that are to be stored in the ledger. As we depict in the figure, the ultimate destination of all types of events is the distributed ledger, which serves as the *single version of the truth* for the updating history of the system.

In particular, for the Cardano [**?**] case one has to consider the possible enhancements necessary for the Ouroboros family of protocols [**?**]. Apart from the block validation affecting the consensus rules, one has to take into account the possible impact from the protocols from the *Update Governance* component depicted in the figure.

Update Governance is the component in the architecture that enables a truly decentralized and democratic updating mechanism. It allows all users to freely vote on update proposals and then considers the stake distribution for forming the final result. This is achieved via a secure *voting protocol* that must be tightly integrated with the Blockchain consensus protocol. Moreover, for these users that might be owners of stake but lack the skills and expertise to vote for or against a software update proposal, a *voting delegation protocol* must be in place. This will enable the delegation of the voting right to some other user (regardless of his/her stake), a so-called *expert*, that will assume the voting task in his/her stead. The ultimate goal of the voting and delegation protocols is all the participating users to reach at an *update consensus*. In other words, to reach a common agreement on the update proposal priorities and on the evolution roadmap of the ledger system.

Voted update proposals eventually, must be deployed to the system and become *adopted updates*, which are activation events that are stored in the ledger, as well.

5.Nikos: We need to decide if we will store activation events in the ledger

The *Update Logic* component, handles the rules that will guarantee a seamless and versatile activation of software updates into the ledger system. More specifically this component implements the necessary "logic" for conflict resolution and secures the updating mechanism from updates that might lead the system to an unstable state. In addition, it supports the correct enforcement of update dependencies and prevents update patches to be installed, if required previous patches are missing. Finally, it implements the notion of *Update Policies*, which differentiate speed of deployment and method of deployment based on: a) the type of change (bug -fix or change request), b) the part of the system that is affected by the change (consensus rules impact, or only software impact) and c) the urgency of the change (severity level). The method of deployment determines the actual mechanism that will be used in order to activate an update in the ledger system. It is executed by the *Deployment* component depicted in the figure. For protocol changes one must consider the most appropriate method for such a deployment and choose among a set of well-known practices such as hard-forks and soft-forks but also from more niche techniques, such as velvet-forks [7] and the sidechains mechanism [**?**].

6.Nikos: I am not sure that hard/soft/velvet forks are a deployment method. Sidechains is. I think there are just different types of consensus rules change

For all these to work, the Update Logic component must be based on an appropriate set of *Update Metadata*. These metadata must obligatory accompany each submitted Update Proposal and sufficiently describe the change, its expected benefit, its type, its urgency for deployment, its possible conflicts with other updates and many more. In addition to the metadata, every update proposal must be accompanied by the update patch that comprises the actual code to be installed. Both of these two, the metadata and the code, cannot be stored in the ledger, due to their sizing requirements. In order to satisfy the requirement for a truly decentralized update mechanism, one should avoid to store these data into any type of centrally owned servers (e.g., into the cloud). One should consider the use of a P2P file storage or database solution, with *content addressable* [**?**] storage capabilities. Moreover, the proposed solution must guarantee the security of the downloaded software, as well as the authenticity with respect to the original update proposal.

8 PROTOTYPE IMPLEMENTATION

The Prototype Architecture

Validation of the Prototype

9 CONCLUSION

APPENDIX

1 Definition of Update Proposal Metadata

We define an Update Proposal to be a .tar file containing the update information and necessary files. The Update Proposal MUST contain a JSON document, the Update Manifest, named "manifest.json", which must exist at the root of the Update Proposal tree and constitutes the heart of the update meta-data. It can contain additional files and folders which are referenced by the manifest.

name: A computer-readable name for the proposal for reference; this must only consist of the characters `[a-z0-9_]`.

title: A human-readable title of the proposal

description: A human-readable description of the proposal. This is recommended to be limited to one paragraph. A more extended rationale and discussion can be provided in the *url* (see below).

url: (optional) A URL pointing to a discussion forum in which the proposal is discussed and social consensus is reached. The URL can contain a BIP/EIP/RFC-style document, comments, and so on. It can be on reddit, GitHub, or other public fora that are typically used for such purposes. This URL is only used for human purposes and does not play any role in the actual update.

sw_version_from: This is the software version that is required in order to apply this Update Proposal.

sw_version_to: This is the software version after the activation of this Update Proposal

prot_version_from: This is the consensus protocol version that is required in order to apply this Update Proposal.

prot_version_to: This is the consensus protocol after the activation of the consensus rules changes of this Update Proposal.

voting_delay: Integer. The number of slots after the inclusion of the Update Proposal on the blockchain at which the signalling period starts.

voting_duration: Integer. The number of slots the voting period duration will equal to.

activation_delay: Integer. The number of slots after the voting period is over at which the activation slot occurs.

consensus_type: One of "soft", "hard", "velvet", or "sidechain", indicating the consensus rules change type for this version. 7.Nikos: I think that sidechains is not a type of consensus rules changes as the rest, is a deployment mechanism

update_type: One of "Change Request", "New Feature", or "Bug Fix". It denotes the type of the update.

urgency: One of "low", "medium", or "high". It denotes the required speed of deployment. Hot fixes and bug fixes pertaining to security-critical updates must be marked as "high". All other updates must be marked as "medium" or "low". "medium" is reserved for

minor version updates and bugfixes which are not security-critical, but may be affecting user experience. ”low” is reserved for major version updates as well as any new features.

tags: (optional) An array of tags. Each tag is a string satisfying the regular expression `[a-z0-9_]`. These tags can be used for selective governance vote delegation.

apply: The filename of an ansible playbook [, ansible] which resides within the Update Proposal, to be executed when the update is to be applied. It is recommended to place the playbook in the root directory of the Update Proposal and call it `apply.yml`.

unapply: (optional) The filename of an ansible playbook, which resides within the Update Proposal, to be executed when the update is to be uninstalled. It is recommended to place the playbook in the root directory of the Update Proposal and call it `unapply.yml`.

depends: An array of strings containing the hashes of Update Proposals which must be applied prior to the particular Update Proposal. The current update cannot be applied unless the Update Proposals in depends have been installed. An empty array means that there are no dependencies.

recommends: An array of strings containing the hashes of Update Proposals which are recommended to have been applied prior to the particular Update Proposal. While not necessary if the user doesnâĂŹt want to install them, these updates will work well if installed prior to the current update.

conflicts: An array of strings containing the hashes of Update Proposals which conflict with the new proposal. If any updates in the conflicts array have already been installed, the current update cannot be installed, unless the updates have been uninstalled. An empty array means that there are no conflicts.

REFERENCES

[1] Lars Brünjes, Aggelos Kiayias, Elias Koutsoupias, and Aikaterini-Panagiota Stouka. 2018. Reward Sharing Schemes for Stake Pools. *CoRR* abs/1807.11218 (2018). arXiv:1807.11218 http://arxiv.org/abs/1807.11218

[2] Vitalik Buterin. 2017. Notes on Blockchain Governance. *white paper* (2017).

[3] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* (2014).

[4] Philipp Kant, Lars Brunjes, and Duncan Coutts. 2019. Engineering Design Specification for Delegation and Incentives in Cardano – Shelley - An IOHK Technical Report. (2019).

[5] Dimitris Karakostas, Aggelos Kiayias, and Mario Larangeira. 2018. Account Management and Stake Pools in Proof of Stake Ledgers. (2018).

[6] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[7] Alexei Zamyatin, Nicholas Stifter, Aljosha Judmayer, Philipp Schindler, Edgar R. Weippl, and William J. Knottenbelt. 2018. A Wild Velvet Fork Appears! Inclusive Blockchain Protocol Changes in Practice - (Short Paper). In *Financial Cryptography and Data Security - FC 2018 International Workshops, BITCOIN, VOTING, and WTSC, Nieuwpoort, Curaçao, March 2, 2018, Revised Selected Papers*. 31–42. https://doi.org/10.1007/978-3-662-58820-8_3

[8] Bingsheng Zhang, Roman Oliynykov, and Hamed Balogun. 2018. A Treasury System for Cryptocurrencies: Enabling Better Collaborative Intelligence. Cryptology ePrint Archive, Report 2018/435. (2018). https://eprint.iacr.org/2018/435.