Decentralized Software Updates for Stake-Based Ledger Systems

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 Proof of Stake 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 two-fold: First, we formally define what it means for a *decentralized* software update system to be secure, and then, we propose a decentralized secure software update protocol that covers the full lifecycle of a software update from the ideation phase (the moment in which a change to the blockchain protocol is proposed) to the actual activation of the updated blockchain protocol. To the best of our knowledge, this is the first work that aims at formalizing the notion of a decentralized secure update for a blockchain and also that takes such a holistic approach on software updates.

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; one basic form of which are software updates. Therefore the adoption of software updates 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: change requests, bug-fixes, security holes, new-feature requests, various optimizations, code refactoring etc.

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. 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, 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(s) of the system, who essentially constitutes a central authority. Even in the case where the community has initially reached consensus for an update proposal (in the form of an *Improvement Proposal* document), through 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. 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.

Our contributions. We put forth a novel mechanism for realizing software updates. In our proposed scheme, an update proposal is possible to be submitted by anyone who can submit a transaction to the blockchain. The decision of which update proposal will be applied and which will not is taken collectively by the community and not centrally. Thus, the roadmap of the system is decided jointly. Moreover, this process is no longer an informal discussion process, but part of an *update protocol*. All relevant events generated are stored within the blockchain itself, and thus recorded in the immutable update history of the system. Moreover, the role of the code maintainer, who used to take decisions on the correctness of the submitted new code and guarantees the validity of the downloaded software, is replaced by the stakeholders' community.

In the context of software updates for public stake-based blockchain systems, we introduce the capability to take stake-based decisions based on: a) the software update priority, b) the correctness of the new code, c) the maintenance of the code base and d) the authenticity and safety of the downloaded software. We introduce the problem of then activating the changes on the blockchain without risking a chain split as the *decentralized software updates problem*, and put forth the first solution to this problem. In addition, we investigate how to enable different update policies based on the software update context (i.e., update metadata) and at the same time fulfill software dependency requirements and resolve conflicts (cf. Appendix E).

Related Work. 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 [1], Bitcoin Cash [2], Ethereum [3] and Zcash [4] 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 [5]. There exist blockchain systems [6], [7] 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 solutions do not deal with the residual 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 more importantly, with the activation of the changes. That is why in the above cases, there exists a central authority (or group) that assumes the role of the source code maintainer. A similar approach is proposed by Bingsheng et al. [8], 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 et al. work is focused only on the treasury system, which covers only the initial phase in the lifecycle of an update proposal.

Goal of the paper. In this paper we propose a secure software update mechanism that enables a decentralized approach to the blockchain software updates problem. We examine all phases in the lifecycle of a software update and propose practical decentralized alternatives that can be adopted in the real world. These alternatives substitute any *central authority* with the *stakeholders' community*. In order to enable this *decentralization* of a software updates, we exploit existing primitives that we combine in order to form a novel decentralized software updates protocol. From a security perspective, we formally define what is a secure activation of changes on a blockchain and prove the security of our protocol with respect to this definition. Our protocol ensures 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 stakeholders will never be applied, c) an update proposal that it is approved by the stakeholders will be eventually applied, d) an update proposal that the stakeholders decide has a higher priority than some other proposal will take higher priority, e) downloaded software is authentic and safe, and finally f) it protects against chain splits during activation of the proposed updates.

Outline of the paper. In Section 2 we present our decentralized approach for the lifecycle of a software update. Section 3 defines what a secure software update mechanism is and proves the security of our protocol. In the appendix, we contrast with the centralized approach to the software update lifecycle, provide

more details for the voting and delegation mechanisms of our protocol and we discuss our proposal for a metadata-driven software update mechanism.

2 The Lifecycle of a Decentralized Software Update

A *software update (SU)* is the unit of change for the blockchain software. In Figure 1, we depict the full lifecycle of a software update following a decentralized approach. In this lifecycle, we identify four distinct consecutive phases: a) the *ideation phase*, b) the *implementation phase*, c) the *approval phase* and d) the *activation phase*. In the subsequent subsections, we provide a detailed description of each individual phase.

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. In the Appendix C, you can find a description of the centralized approach for each phase, for comparison reasons. 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.

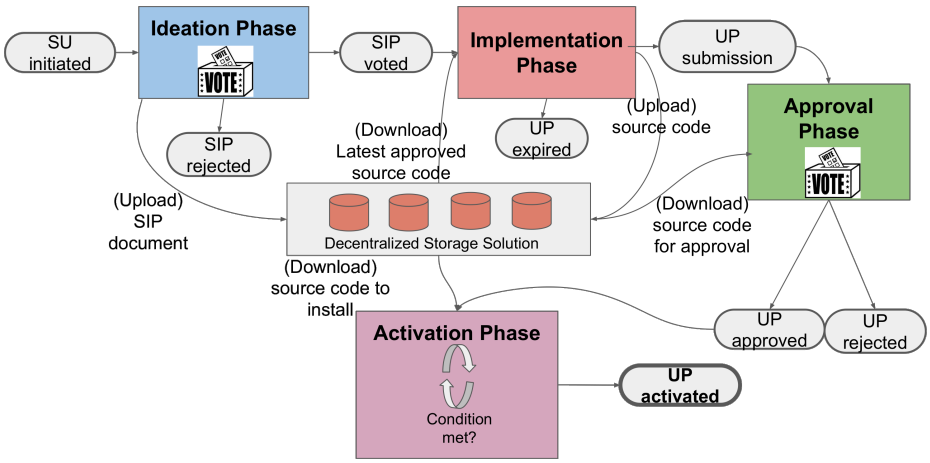

Fig. 1. The lifecycle of a software update (a decentralized approach)

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.

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

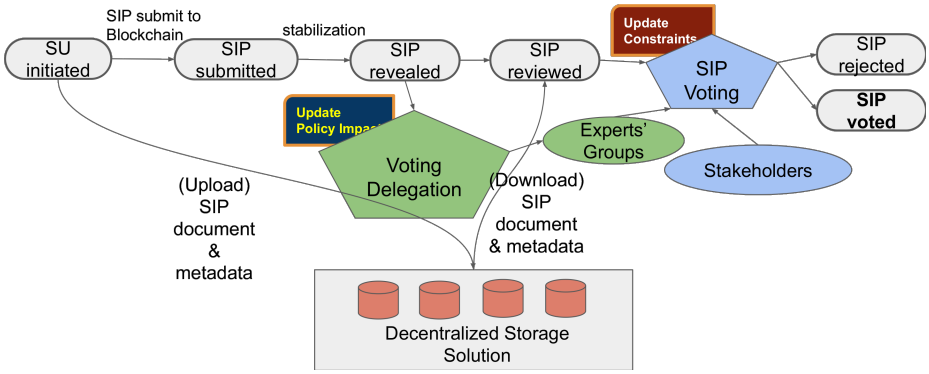

Fig. 2. The ideation phase.

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 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), the priority/criticality of the SU 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 other SUs (e.g., update dependencies or update conflict issues), or against any prerequisites required, in order to be applied. We call these requirements as *update constraints* (cf. Appendix E.3) and can be abstracted as a predicate, whose evaluation determines the feasibility of a software update. 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. It can be something very common, as a developer-

[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

owned Github repository, to something more elaborate as a content-addressable decentralized file system. In any case, it is in the interest of the party that made the proposal to keep the SIP available, otherwise the SIP will be rejected. This hash id is committed to the blockchain in a two-step 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 fee-supported transactions, which are committed to the blockchain. More details for the proposed voting mechanism can be found in the Appendix D.1. Note that voters are not called to vote only for the SIP per se, but also for the various characteristics of the software update, such as the type of the change, the priority/criticality, etc., which are described in the corresponding metadata. These characteristics will drive the *update policy* adopted, as we will describe in the corresponding section.

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 preserve the right to override the delegate's vote, if he/she wishes. Essentially, we propose the use of a delegation mechanism for three distinct reasons: a) for technical expertise, b) for special categories of software updates (e.g., security fixes, platform specific issues etc.) and c) for ensuring an appropriate level of stake participation in the protocol, similar to the stakepools concept described in Karakostas et. al. [9]. More details for the proposed delegation mechanism can be found in the Appendix D.2.

Implementation. 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.

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 to maintain the code-base (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 (i.e., a call for approval by the developer to the code maintainer authority) must be realized.

In Figure 3, we depict the decentralized implementation phase. 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

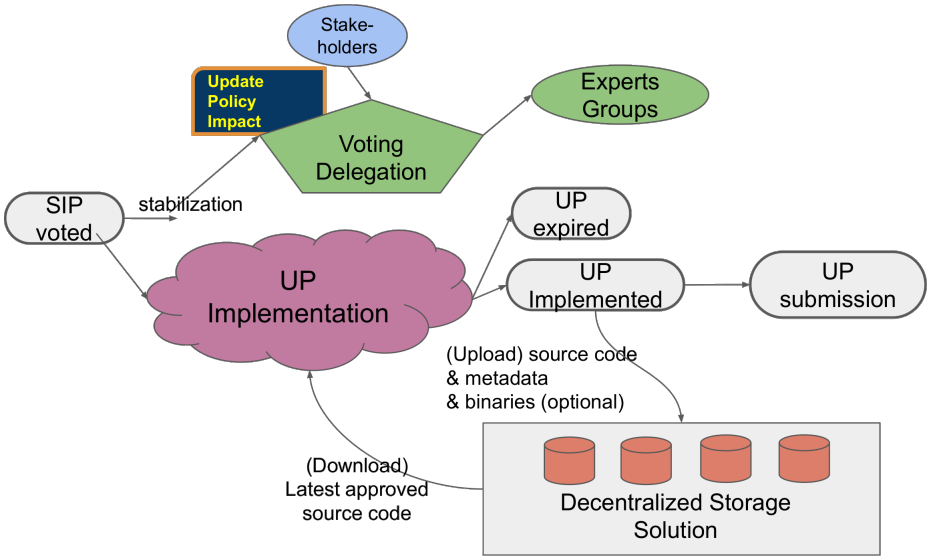

Fig. 3. The implementation phase.

the blockchain and downloads it locally, using the link to the code repository provided in the UP metadata. 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 (see the details of the proposed delegation mechanism in the Appendix D.2). Upon the conclusion of the implementation, 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 request to approval. This is accomplished with a fee-supported transaction, which represents the "decentralized equivalent" to a pull-request.

Approval. 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 guarantee its *correctness*. Overall, the approver approves the correctness and safety of the submitted UP. In the Appendix C.3, we provide a detail list of the properties that a UP must satisfy in order to justify its correctness.

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, which is enabled, similarly to the Ideation phase, by the voting and delegation mechanism in place. The approval phase in the decentralized approach is very similar to the Ideation phase, and because of space constraints we have moved the corresponding figure into the Appendix C.3.

Activation. The final phase in the lifecycle of a software update, depicted, 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 then synchronize with their peers before the changes take effect. Thus, the activation phase is clearly a synchronization period. Its primary purpose is for the nodes to activate changes synchronously.

Why do we need such a synchronization 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, or even the installation of new hardware, 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, since different versions of the blockchain will be running concurrently. Of course, this is true only for those software updates that impact the consensus protocol. For all the other SUs, the participating nodes can activate the changes asynchronously.

Once the voting period of the Approval phase ends, the votes have been stably stored in the blockchain and the tally result is positive, then the activation period is initiated. In Figure 4, we depict the activation period in the decentralized setting. The first step in the activation phase is the installation of the

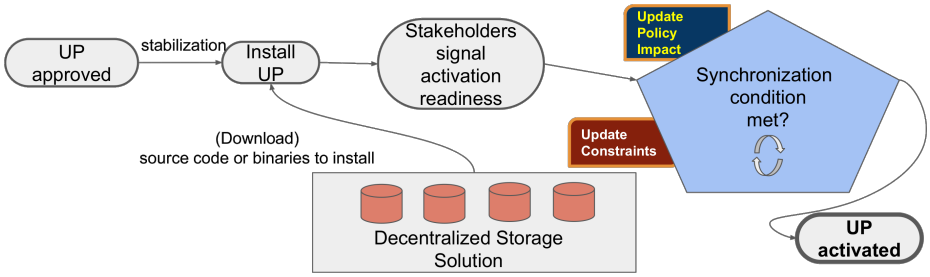

Fig. 4. The activation phase.

software update. 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 as a means to synchronization.

One popular method for signaling, that is used by Bitcoin [1], is every new block issued to be stamped with the new version of the software, signifying their readiness for the new update. This approach is simple and straightforward to implement, but it restricts signaling only to *maintainer nodes*[2], excluding other type of nodes like *full-node clients*, *light-node clients* etc. Moreover, the major drawback of this approach is that the activation of changes is delayed significantly by the block creation process, which is slow. One way to overcome this delay is to use sampling for estimating the adoption percent, instead of waiting for all the signaling blocks to be stably stored into the blockchain. For example, a *random sampling* method could be used to collect a representative subset of m signaling blocks that will be also proportional to the stake and thus we could base our calculation of the adoption threshold on this sample. Intuitively, the sampling size should reduce the more the stake distribution diverges from the uniform distribution and vice versa.

Since the adoption threshold is based on the stake that has signaled and not on the number of signaling blocks, another more flexible approach would be to signal by means of simple messages (i.e., fee-based transactions) that are issued from stakeholder keys and thus are bound to specific stake. In this way, we only have to wait for the stabilization of these messages into the blockchain and not for individual blocks to stabilize (a single block can store many such messages).

Furthermore, we could even use a separate consensus protocol, just for the purpose to agree on the binary value carried by these messages (Ready | Not Ready) (i.e., a binary Byzantine Agreement problem). The parties taking part in this new protocol would be any node (client, or maintainer) that actively participates in the underlying consensus protocol and thus needs to synchronize with its peers with respect to the activation of some change. The main assumptions (network, setup, computation as in Garay et al. [10]) of this consensus protocol, naturally will be identical to the assumptions of the underlying consensus protocol and therefore the *resiliency*[3] of the protocol will also be the same.

All in all, by departing from the block signaling solution, we can distinguish between the readiness of different types of nodes and achieve a faster calculation of the adoption threshold. Of course, the downsides in this case are the added complexity and the extra fee that has to be paid for each activation signal.

In the absence of a central authority to set a deadline for the activation of changes, the parties need a way to synchronize, in order to avoid chain splits. Hence we need some sort of a synchronization mechanism. Signaling is indeed

[2] By *maintainer* we mean a node that runs the consensus protocol and can issue a new block (also called *minter*).

[3] According to Garay et. al. in [10], the resiliency is the fraction (t/n) of misbehaving parties a protocol can tolerate, where t are the number of adversaries and n are the total number of parties.

the most popular method for synchronization. However, signaling alone is not enough to protect from the risk of a chain split. This is exactly the topic of the next section, where we also discuss our proposal for the activation phase.

3 Defining a Secure Update System

We have seen that upon the approval of a UP, users start installing the software update (i.e., upgrade) and wait for the activation of the changes. We call the total time period from the approval of a UP until the actual activation of the changes in the blockchain system as the *activation lag*. Our protocol should try to minimize this lag and at the same time ensure a secure activation of the changes.

The Activation of Changes. In order to enable a swift and effective update mechanism, we want to minimize the activation lag. We need some minimum necessary percent of stake to have upgraded, before the actual activation of the new software takes place that will ensure that a chain split will be avoided. The *adoption threshold* is used in the activation phase and corresponds exactly to the minimum percent of stake that is necessary to have signaled activation readiness, before the actual activation takes place. It is essentially a synchronization point that ensures that a sufficient percent of stake has upgraded and thus it is safe to actually activate the changes. Please note that the adoption threshold is only relevant for software updates that impact the consensus protocol. For all other software updates the activation can take place immediately after the upgrade.

Let us assume that the adoption threshold of our software updates protocol is called τ_A. Intuitively, we would like τ_A to ensure that a sufficient percent of honest stake will activate, in order for the new version of the blockchain to be secure. Lets call r the resiliency of the new version of the consensus protocol (i.e., after the activation of changes). Naturally, if we choose $\tau_A \leq (1-r) \times 100$, then we allow the activation of changes to take place before the minimum required percent of honest stake has the chance to upgrade. Thus, we activate too-early and we risk the security of the new blockchain.

If we consider $\tau_A > (1-r) \times 100$, then a possible attack would be for the the adversary to rush to signal, so that the threshold τ_A is met, without (at least) $(1-r) \times 100$ percent of honest stake to have enough time to complete the upgrade. The activation will take place and the new blockchain will run (at least for some time) with an honest stake percent that does not ensure its security.

In order to prevent this attack, the adoption threshold should be sufficiently large, so that the required percent of stake that has signaled will ensure that the new blockchain will kick off with sufficient honest stake. It is easy to see that in this case, we need $\tau_A > \frac{1}{r} \times T$, where T is the total percent of adversary stake. If we note as h_a the *actual* percent of honest stake running the protocol, then the requirement for the adoption threshold is: $\tau_A > \frac{100-h_a}{r}$

For example, if $r > \frac{1}{2}$ and we estimate that $h_a = 60\%$, then we will need an adoption threshold of $\tau_A > 80\%$, in order to be safe from this attack. However, a

too high value of the adoption threshold, one where $\tau_A > h_a$, introduces another risk; the risk of giving the opportunity to the adversary to block an activation by refusing to signal. We call this a *Denial of Activation* attack. Intuitively, this blocking problem can be resolved by setting an *activation deadline* —a time point in the future— where the activation will take place *regardless* of the adoption threshold condition. To this end, we introduce the concept of the *safety lag*.

The safety lag T_s is a metadata-driven artificial delay, which is imposed during the activation phase, in order to give time to the honest stake to upgrade. The safety lag is determined by: 1) the time required to download the software update and 2) the time required to complete the deployment process. For example, there might be software updates that entail a very complex deployment process; one that even a hardware upgrade is required before the software upgrade. In addition, a large software update might require significant time to be downloaded over a slow network connection. In such a case, the safety lag must ensure plenty of time to the stakeholders to upgrade. We propose that the complexity of the installation process and the size of the software update to be recorded as important characteristics of a decentralized software update's metadata (cf. Appendix E.2). This information will drive the choice of the length of the safety lag and enable a metadata-driven update policy.

We propose a synchronization mechanism for the activation phase, which is based on the safety lag and the adoption threshold. Essentially the activation of changes is based on two separate conditions and it is triggered when either of the two comes true: a) Either the adoption threshold is met, based on the above equation, or b) the safety lag period expires. In other words, the safety lag acts as a time upper bound in the case where the adoption threshold is not met. Our protocol ensures that whichever of the two comes first, the new blockchain will be secure and at the same time our update mechanism is protected against the denial of activation attack. We assume that the parties agree on the safety lag by means of a block index. That is, the parties assume that when the j-th block is generated and becomes a part of the stable part of the blockchain, then there are sufficiently honest parties that have installed the new software and that are ready to run the new code. In the following section, we provide a formal description of our protocol and of our security proofs.

3.1 Ledger Consensus: Model

Before providing our definition and construction we introduce our model.

Model. In this section, we define our notion of protocol execution following [11,12]. The execution of a protocol Π is driven by an environment program $\mathcal{Z}$ that may spawn multiple instances running the protocol Π. The programs in question can be thought of as interactive Turing machines (ITM) that have communication, input and output tapes. An instance of an ITM running a certain program will be referred to as an interactive Turing machine instance or ITI. The spawning of new ITI's by an existing ITI as well as the interaction between

them is at the discretion of a control program which is also an ITM and is denoted by C. The pair $(\mathcal{Z}, C)$ is called a system of ITM's, cf. [12]. Specifically, the execution driven by $\mathcal{Z}$ is defined with respect to a protocol Π, an adversary $\mathcal{A}$ (also an ITM) and a set of parties $P_1, \ldots, P_n$; these are hardcoded in the control program C. Initially, the environment $\mathcal{Z}$ is restricted by C to spawn the adversary $\mathcal{A}$. Each time the adversary is activated, it may send one or more messages of the form $(\mathsf{corrupt}, P_i)$ to C. The control program C will register party P_i as corrupted, only provided that the environment has previously given an input of the form $(\mathsf{corrupt}, P_i)$ to $\mathcal{A}$ and that the number of corrupted parties is less or equal t, a bound that is also hardcoded in C.

We divide time into discrete units called *time slots*. Players are equipped with (roughly) synchronized clocks that indicate the current slot: we assume that any clock drift is subsumed in the slot length.

Ledger Consensus. Ledger consensus (a.k.a. "Nakamoto consensus") is the problem where a set of servers (or nodes) operate continuously accepting inputs that are called transactions and incorporate them in a public data structure called the *ledger*. A ledger (denoted in calligraphic-face, e.g. $\mathcal{L}$) is a mechanism for maintaining a sequence of transactions, often stored in the form of a blockchain (cf. Appendix A.2). In this paper, we denote with $\mathcal{L}$ the algorithms used to maintain the sequence, and with L all the views of the participants of the state of these algorithms when being executed. For example, the (existing) ledger Bitcoin consists of the set of all transactions that ever took place in the Bitcoin network, the current UTXO set, as well as the local views of all the participants. In contrast, we call a *ledger state* a concrete sequence of transactions $\mathsf{Tx}_1, \mathsf{Tx}_2, \ldots$ stored in the stable part of a ledger state L, typically as viewed by a particular party. Hence, in every blockchain-based ledger $\mathcal{L}$, every fixed chain $\mathcal{C}$ defines a concrete ledger state by applying the interpretation rules given as a part of the description of $\mathcal{L}$. In this work, we assume that the ledger state is obtained from the blockchain by dropping the last k blocks and serializing the transactions in the remaining blocks. We refer to k as the *common-prefix parameter*. We denote by $\mathsf{L}^P[t]$ the ledger state of a ledger $\mathcal{L}$ as viewed by a party P at the beginning of a time slot t and by $\check{\mathsf{L}}^P[t]$ the complete state of the ledger (at time t) including all pending transactions that are not stable yet.

For two ledger states (or, more generally, any sequences), we denote by $\preceq$ the prefix relation. Recall the definition of secure ledger protocol given in [10].

Definition 1. *A ledger protocol $\mathcal{L}$ is secure if it enjoys the following properties.*

Persistence. *For any two honest parties P_1, P_2 and two time slots $t_1 \leq t_2$, it holds $\mathsf{L}^{P_1}[t_1] \preceq \check{\mathsf{L}}^{P_2}[t_2]$.*

Liveness. *If all honest parties in the system attempt to include a transaction Tx then, at any slot t after s slots (called the liveness parameter), any honest party P, if queried, will report $\mathsf{Tx} \in \mathsf{L}^P[t]$.*

In this work we also explicitly rely on the property of *Chain Growth (CG)* that is defined as follows.

Chain Growth (CG); with parameters $\tau \in (0,1]$ **and** $s \in \mathbb{N}$. Consider the chain $\mathcal{C}$ adopted by an honest party at the onset of a slot and any portion of $\mathcal{C}$ spanning s prior slots; then the number of blocks appearing in this portion of the chain is at least τs.

We consider a setting where a set of parties run a protocol maintaining a ledger $\mathcal{L}_1$. Following [13] we denote by $\mathbb{A}_1$ the assumptions for $\mathcal{L}_1$. That is, if the assumption $\mathbb{A}_1$ holds, then ledger $\mathcal{L}_1$ is secure under the Definition 1. Formally, $\mathbb{A}_i$ for a ledger $\mathcal{L}_i$ is a sequence of events $\mathbb{A}_i[t]$ for each time slot t that can assume value 1, if the assumption is satisfied, and 0 otherwise. For example, $\mathbb{A}_i$ may denote that there has never been a majority of hashing power (or stake in a particular asset , on this ledger or elsewhere) under the control of the adversary; that a particular entity (in case of a centralized ledger) was not corrupted; and so on. Without loss of generality, we say that the assumption $\mathbb{A}_1$ for the ledger $\mathcal{L}_1$ holds if and only if the number of corrupted parties (the parties that received the input $(\mathsf{corrupt}, \cdot)$) is below the threshold t_1 (where t_1 is part of the control function as described in the beginning of this section).

3.2 Defining Secure Activations

In this section, we provide the definition of secure activations. Our definition is generic in the sense that can be applied to a large class of ledgers (e.g., PoS, PoW and so on). Let $\mathcal{L}_1$ be the ledger that the parties are running with assumption $\mathbb{A}_1$. We now consider a new ledger $\mathcal{L}_2$ with assumption $\mathbb{A}_2$. We say that the assumption $\mathbb{A}_2$ for the ledger $\mathcal{L}_2$ holds if and only if the number of corrupted parties that received the command $(\mathsf{activate}, \cdot)$ and the command $(\mathsf{corrupt}, \cdot)$ is below the threshold t_2 (where t_2 is part of the control function as described previously). We say that $\mathbb{A}_1 = \mathbb{A}_2$ if and only if $\mathsf{t}_1 = \mathsf{t}_2$.

$\mathcal{L}_2$ represents un updated version of $\mathcal{L}_2$ (i.e., an improved protocol with respect to $\mathcal{L}_1$). In the scenario that we are considering in this paper, each party that is running $\mathcal{L}_1$ could receive the input $(\mathsf{activate}, \mathcal{L}_2)$. A party that receives this command starts the *activation process* in order to run $\mathcal{L}_2$ (and maybe stop running $\mathcal{L}_1$). Let t_{P_i} denote the time in which a party P_i receives the activation command and let $\mathcal{P}^{\mathsf{u}}$ be the set of parties that received this command. Without loss of generality, let P_1 be the first party to receive the update command (note that $t_{P_1} \leq t_{P_i}$ for all $P_i \in \mathcal{P}^{\mathsf{u}}$).

Informally, a secure activation process guarantees that if the set of honest parties that are willing to run $\mathcal{L}_2$ is such that $\mathbb{A}_2[t] = 1$ for some $t \geq t_{P_1} + \Delta_1$, then the state of $\mathcal{L}_2$ at time $t_{P_1} + \Delta_1 + \Delta_2$ corresponds to the state of $\mathcal{L}_1$ at time t_{P_i} with $P_i \in \mathcal{P}^{\mathsf{u}}$. The parameter Δ_1 defines a *synchronization* parameter to ensure that enough honest parties that have received $(\mathsf{activate}, \cdot)$ (i.e., the number of honest parties is such that $\mathbb{A}_2$ holds). Δ_2 instead represents the time required for the update process to be completed.

The above implies that L_2 extends L_1 and that $\mathcal{L}_2$ is secure (i.e. it enjoys consistency and liveness). In a nutshell, a secure update process guarantees that the state of the old ledger is moved into the new ledger, and that the new ledger is secure. We now give a more formal definition.

Definition 2 (Activation Process). *We say that an activation process with activation parameters (Δ_1, Δ_2) (where $\Delta_1, \Delta_2 \in \mathbb{N}$) is secure if the following condition is satisfied: If $\mathbb{A}_2[t] = 1$ for all $t \geq t_{P_1} + \Delta_1$ and $\mathbb{A}_1[t'] = 1$ for all $t' \leq t_{P_1} + \Delta_1 + \Delta_2$, then the state of $\mathcal{L}_2$ at some time $T \in [t_{P_1} + \Delta_1, t_{P_1} + \Delta_1 + \Delta_2]$ is such that $\mathsf{L}^\star = \mathsf{L}_2$ where $\mathsf{L}_1^{P_i}[t_{P_i}] \preceq \mathsf{L}^\star$ for some $P_i \in \mathcal{P}^{\mathsf{u}}$.*

We note that this definition says nothing on the security of $\mathcal{L}_1$ after the time slot $t_{P_1} + \Delta_1 + \Delta_2$. Indeed, the Definition 2 implies that if after this time slot $\mathcal{L}_1$ becomes insecure then the security of $\mathcal{L}_2$ is not compromised.

Candidate Protocol. Let $\mathcal{L}_1$ be the ledger that the parties are running and $\mathcal{L}_2$ be a new ledger. In our construction we assume that $\mathcal{L}_1$ and $\mathcal{L}_2$ have same common-prefix parameter k, same chain-growth parameter (τ, s) and that they are secure under the same assumption $\mathbb{A}_1 = \mathbb{A}_2$. We also assume that a block B of $\mathcal{L}_1$ can be transformed into a *special genesis block* B' for $\mathcal{L}_2$. B' is special because not only it represents a valid genesis block for $\mathcal{L}_2$, but it also points to the state of $\mathcal{L}_1$ up to the block B. We use this special genesis block to guarantee that: 1) the state of $\mathcal{L}_1$ at the end of the activation is moved into the state of the new ledger and 2) that we can use a block of $\mathcal{L}_1$ to bootstrap $\mathcal{L}_2$. At a very high level our protocol works as follows. When enough parties have received the command $(\mathsf{activate}, \cdot)$ such that $\mathbb{A}_2$ holds, a block of L_1 is chosen as the special genesis block for $\mathcal{L}_2$. At this point the parties start running $\mathcal{L}_2$ and gradually abandon $\mathcal{L}_1$. In the description of our update protocol we assume that there is a time t^{u} such that $\mathbb{A}_2[t]$ holds for all $t \geq t^{\mathsf{u}} = t_{P_1} + \Delta_1$.[4] We also assume that the parties in $\mathcal{P}^{\mathsf{u}}$ know an index j such that when the j-th block is added to the state of $\mathcal{L}_1$ then $\mathbb{A}_2$ holds. More formally, at time t^{u} the j-th block is part of $\mathcal{L}_1^{P_i}$ for all $P_i \in \mathcal{P}^{\mathsf{u}}$. To ensure that $\mathbb{A}_2$ actually holds and to keep all the honest parties synchronized on the time t^{u} (and on the index j) we use the approach proposed in Sec. 3. In more details t^{u} represents either the time at which the threshold is met, or it represents the time in which the safety lag period ends.

We denote our update protocol with Π. In this, each honest party $P_i \in \mathcal{P}^{\mathsf{u}}$ at time t^{u} executes the following steps.

1. Run $\mathcal{L}_1$ and when the $(j+k)$-th block B^i_{j+k} becomes part of $\check{\mathsf{L}}^{P_i}[t]$ for some $t \geq t^{\mathsf{u}}$ start running also $\mathcal{L}_2$ using $B^{i'}$ as the candidate special genesis block (where $B^{i'}$ is generated using B^i_{j+k}).[5]
2. Run $\mathcal{L}_2$ (and keep running $\mathcal{L}_1$) until there is an agreement on a special genesis block. That is, one of the candidate special genesis block becomes part of $\mathsf{L}_2^{P_i}[t]$ for each honest $P \in \mathcal{P}^{\mathsf{u}}$ for some $t \geq t^{\mathsf{u}}$.
3. Stop running $\mathcal{L}_1$.

Theorem 1. *Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two ledgers that are secure under the assumption $\mathbb{A}_1 = \mathbb{A}_2$, have common-prefix parameter k and chain-growth parameter (τ, s)*

[4] We could require $\mathbb{A}_2$ to hold only for bounded interval of time and our protocol would still work if this interval is large enough.

[5] Note that the special genesis block $B^{i'}$ obtained from B^i_{j+k} could not be adopted as the genesis block for $\mathcal{L}_2$.

then Π is a (Δ_1, Δ_2)-secure update system with $\Delta_1 = k\tau^{-1}$ and $\Delta_2 = k\tau^{-1} + k\tau^{-1}$.

We refer the reader to Appendix B for the formal proof. Although the above protocol is generic it has the limitation that the parameters of $\mathcal{L}_1$ and $\mathcal{L}_2$ have to be the same. We could get rid of this limitation by requiring an additional property on $\mathcal{L}_2$. If $\mathcal{L}_2$ supports temporary dishonest majority [14] then we do not need the assumption on the equality parameters to hold anymore. Indeed, in this case we can modify Π as follows. At time t^{u} the honest parties wait for the block B_j to be stable and then they start using it as a genesis block for $\mathcal{L}_2$. We note that here it is crucial for $\mathcal{L}_2$ to support temporary dishonest majority since the adversary could potentially see the genesis block way sooner than honest parties can see it. This means that there is an interval of time in which only the adversary is extending the genesis block and this corresponds to a situation in which the adversary has temporary dishonest majority. For the best of our knowledge, the positive results regarding temporary dishonest majority are for PoW blockchain like Bitcoin as showed in [14]. Another approach that can be currently applied to PoS protocol as well is the following. We let the parties in $\mathcal{P}^{\mathsf{u}}$ to run a multi-party computation (MPC) protocol that guarantees output delivery in order to generate a special genesis block for $\mathcal{L}_2$. If we assume that the number of honest parties in $\mathcal{P}^{\mathsf{u}}$ represents the majority then they can execute this MPC protocol and then start running $\mathcal{L}_2$ using the obtained genesis block.

References

1. Nakamoto S.: Bitcoin: A peer-to-peer electronic cash system (2008)
2. Bitcoin-Cash: The Bitcoin Cash Roadmap. https://www.bitcoincash.org/roadmap.html (2019)
3. Buterin V., et al.: A next-generation smart contract and decentralized application platform. white paper (2014)
4. Ben-Sasson E., Chiesa A., Garman C., Green M., Miers I., Tromer E., Virza M.: Zerocash: Decentralized Anonymous Payments from Bitcoin. http://zerocash-project.org/media/pdf/zerocash-extended-20140518.pdf (2014)
5. Buterin V.: Notes on Blockchain Governance. white paper (2017)
6. Duffield E., Diaz D.: Dash: A Payments-Focused Cryptocurrency. https://github.com/dashpay/dash/wiki/Whitepaper (2018)
7. Decred: Decred White Paper. https://docs.decred.org/ (2019)
8. Zhang B., Oliynykov R., Balogun H.: A Treasury System for Cryptocurrencies: Enabling Better Collaborative Intelligence. Cryptology ePrint Archive, Report 2018/435: https://eprint.iacr.org/2018/435 (2018)
9. Karakostas D., Kiayias A., Larangeira M.: Account Management and Stake Pools in Proof of Stake Ledgers (2018)
10. Garay J. A., Kiayias A.: SoK: A Consensus Taxonomy in the Blockchain Era. IACR Cryptology ePrint Archive: vol. 2018, p. 754: URL https://eprint.iacr.org/2018/754 (2018)
11. Garay J. A., Kiayias A., Leonardos N.: The Bitcoin Backbone Protocol: Analysis and Applications. In Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II: pp. 281–310 (2015)
12. Canetti R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In 42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA: pp. 136–145 (2001)
13. Gaži P., Kiayias A., Zindros D.: Proof-of-Stake Sidechains. Cryptology ePrint Archive, Report 2018/1239: IEEE Security Privacy 2019 https://eprint.iacr.org/2018/1239 (2018)
14. Avarikioti G., Kaeppeli L., Wang Y., Wattenhofer R.: Bitcoin Security under Temporary Dishonest Majority. CoRR: vol. abs/1908.00427: URL http://arxiv.org/abs/1908.00427 (2019)
15. Kant P., Brunjes L., Coutts D.: Engineering Design Specification for Delegation and Incentives in Cardano – Shelley - An IOHK Technical Report (2019)
16. Zamyatin A., Stifter N., Judmayer A., Schindler P., Weippl E. R., Knottenbelt W. J.: 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: pp. 31–42: doi:10.1007/978-3-662-58820-8_3: URL https://doi.org/10.1007/978-3-662-58820-8_3 (2018)

A Additional Background

We denote the security parameter by λ and use "||" as concatenation operator (i.e., if a and b are two strings then by $a||b$ we denote the concatenation of

a and b). For a finite set Q, $x \xleftarrow{\$} Q$ denotes a sampling of x from Q with uniform distribution. We use the abbreviation PPT that stands for probabilistic polynomial time. We use $\mathsf{poly}(\cdot)$ to indicate a generic polynomial function. When it is necessary to refer to the randomness r used by and algorithm A we use the following notation: $A(\cdot; r)$. We say a function ν is *negligible* if for every positive integer c there is an integer N_c such that for all $x > N_c$, $|\nu(x)| < 1/x^c$. We say $a \overset{\text{negl}}{\leq} b$ for real values a, b to mean that $a \leq b + \nu(\lambda)$ for negligible function ν. We denote with $[n]$ the set $\{1, \ldots, n\}$, with $\mathbb{F}$ an arbitrary (but fixed) finite field and with $\mathbb{N}$ the set of non-negative integer.

A.1 Definitions

Definition 3 (Signature scheme). *A triple of* PPT *algorithms* $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ *is called a* signature scheme *if it satisfies the following properties.*

Completeness: *For every pair* $(s, v) \xleftarrow{\$} \mathsf{Gen}(1^\lambda)$*, and every* $m \in \{0,1\}^\lambda$*, we have that* $\Pr[\mathsf{Ver}(v, m, \mathsf{Sign}(s, m)) = 0] < \nu(\lambda)$.

Consistency (non-repudiation): *For any* m*, the probability that* $\mathsf{Gen}(1^\lambda)$ *generates* (s, v) *and* $\mathsf{Ver}(v, m, \sigma)$ *generates two different outputs in two independent invocations is smaller than* $\nu(\lambda)$.

Unforgeability: *For every* PPT $\mathcal{A}$*, there exists a negligible function* ν*, such that for all auxiliary input* $z \in \{0,1\}^\star$ *it holds that:*

$$\Pr[(s, v) \xleftarrow{\$} \mathsf{Gen}(1^\lambda); (m, \sigma) \xleftarrow{\$} \mathcal{A}^{\mathsf{Sign}(s,\cdot)}(z, v) \wedge \mathsf{Ver}(v, m, \sigma) = 1 \wedge m \notin Q] < \nu(\lambda)$$

where Q *denotes the set of the messages requested by* $\mathcal{A}$ *to the oracle* $\mathsf{Sign}(s, \cdot)$.

To simplify the description of our protocol we denote a signature of a message m computed using the secret key s with $\sigma_s(m)$.

A.2 The blockchain abstraction

Following [8], we abstract the following concepts.

- Coin. We assume the underlying blockchain platform has the notion of Coins or its equivalent. Each coin can be spent only once, and all the value of coin must be consumed. Each coin consists of the following 4 attributes:
  - Coin ID: it is an implicit attribute, and every coin has a unique ID that can be used to identify the coin.
  - Value: It contains the value of the coin.
  - Cond: It contains the conditions under which the coin can be spent.
  - Payload: It is used to store any non-transactional data.

– Address: conventionally, an address is merely a public key, pk, or hash of a public key, $h(\mathsf{pk})$. To create coins associated with the address, the spending condition of the coin should be defined as a valid signature under the corresponding public key pk of the address. In this work, we define an address as a generic representation of some spending condition. Using the recipient address, a sender is able to create a new coin whose spending condition is the one that the recipient intended; therefore, the recipient may spend the coin later.
– Transaction: Each transaction takes one or more (unspent) coins, denoted as $\{\mathsf{Inp}\}_{i\in[n]}$, as input, and it outputs one or more (new) coins, denoted as $\{\mathsf{Oup}\}_{j\in[m]}$. Except special transactions, the following condition holds:

$$\sum_{i=1}^{n} \mathsf{Inp}_i.\mathsf{Value} \geq \sum_{j=1}^{m} \mathsf{Oup}_j.\mathsf{Value}$$

and the difference is interpreted as transaction fee. The transaction has a Verification data field that contains the necessary verification data to satisfy all the spending conditions of the input coins $\{\mathsf{Inp}\}_{i\mathsf{Inp}[n]}$. In addition, each transaction also has a Payload field that can be used to store any non-transactional data. We denote a transaction as $\mathsf{Tx}(A;B;C)$, where A is the set of input coins, B is the set of output coins, and C is the Payload field. Note that the verification data is not explicitly described for simplicity.

B Proof of Theorem 1

Proof. To simplify our proof we introduce the notion of *canonical scenario* for the the ledger $\mathcal{L}_2$. In a canonical scenario the ledger $\mathcal{L}_2$ is executed in the standard way. More precisely, we assume the existence of a genesis block and that $\mathbb{A}_2[t]$=1 for all $t \geq 0$. Let $\mathcal{P}$ be the set of parties that is running $\mathcal{L}_2$. Also, let t_j be the smallest time slot in which B_j appears in $\mathsf{L}_2^{P_i}[t]$ for each $P_i \in \mathcal{P}$ and let t_{j+k} be smallest time slot in which B_{j+k} appears in $\mathsf{L}_2^{P_i}[t']$ for each $P_i \in \mathcal{P}$. We are now ready to prove the security of Π.

In the protocol Π, by assumption, we have that $\mathbb{A}_2[t] = 1$ for all $t \geq t^{\mathsf{u}}$. From the description of Π we can claim that $t^{\mathsf{u}} \leq t_{P_1} + k\tau^{-1} = t_{P_1} + \Delta_1$. That is, when an honest party is activated she waits to be sure that also the other honest parties are activated. Hence, from the chain-growth and the common-prefix parameters each party needs to wait at most $\Delta_1 = k\tau^{-1}$ time slots. From the moment when $\mathbb{A}_2$ becomes true the activation process takes $\Delta_2 = k\tau^{-1} + k\tau^{-1}$ time slots more to be completed. This is because the parties need to wait that the $(j+k)$-th block of L_1 is part of $\mathsf{L}^{P_i}[t]$ for all P_i and that k blocks are generated in L_2. Note that when k blocks are generated in L_2 at least k blocks are generated in L_1 since $\mathcal{L}_2$ and $\mathcal{L}_1$ have the same parameters and that the honest parties that maintain $\mathcal{L}_1$ are greater or equal than the parties that maintain $\mathcal{L}_2$. Therefore, the parties need to wait the *special genesis block* of $\mathcal{L}_2$ to appear in $\mathsf{L}_2^{P}[t]$ for each honest $P \in \mathcal{P}^{\mathsf{u}}$. Given that a block in $\mathcal{L}_1$ ($\mathcal{L}_2$) takes at most τ^{-1} time slots then

we have that $\Delta_2 = k\tau^{-1} + k\tau^{-1}$. In the moment that a *candidate* block B^i_{j+k} becomes available to an honest party $P_i \in \mathcal{P}^{\mathsf{u}}$ (i.e., B^i_{j+k} is part of $\check{\mathsf{L}}_1^{P_i}$) then she starts running $\mathcal{L}_2$ using $B^{i'}$ which is computed from B^i_{j+k} as described earlier (we recall that at this time slot the assumption $\mathbb{A}_2$ holds). Let t' be smallest time slot in which B_{j+k} appears in $\mathsf{L}_2^{P_i}[t']$ for each $P_i \in \mathcal{P}$. If we take the execution of the protocol from time t^{u} and t' this can be seen as a canonical execution of $\mathcal{L}_2$ given that the parameters of $\mathcal{L}_1$ and $\mathcal{L}_2$ are the same. The only difference between this and the canonical scenario is that the blocks $B_j, B_{j+1}, \ldots, B_{j+k}$ are generated using $\mathcal{L}_1$, but this does not represent an issue since we are assuming that any block of L_1 can be turned into a block of L_2. □

C The Lifecycle of a Software Update - Centralized vs. Decentralized Approach

In this section, we follow the distinct phases of the lifecycle of a typical software update and discuss the realization in the centralized setting. This will help the reader to compare with the decentralized software update lifecycle proposed in the main part of the paper. In all the phases, the reader will conclude that the two approaches follow exactly the same logical steps, in order to apply a software update; the essential aspect that differentiates the two approaches is who decides (a central authority vs. the stakeholders community).

C.1 Ideation

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 [1], 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 road-map 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.

C.2 Implementation

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 centrally-owned 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 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 code-base, 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.

C.3 Approval

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 guarantee its *correctness*. Overall, the approver approves the correctness and safety of the submitted UP. We provide a detail list of the properties that a UP must satisfy in order to justify its correctness:

- *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.

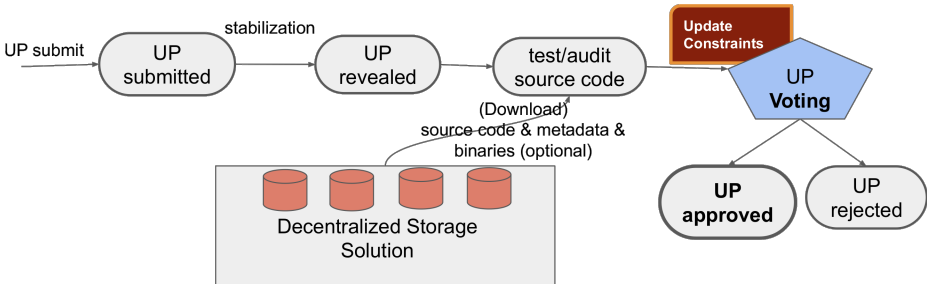

Fig. 5. The approval phase.

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 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.

C.4 Activation

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 by the central authority, 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 [1], there

also exists a signaling mechanism[6]. 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.

D Update Governance

With the term *update governance* we mean the processes used to control the software updates mechanism. In the centralized setting, the mere existence of a central authority (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*. Therefore voting and delegation are key components of a decentralized approach to software updates. In this section, we describe both of these mechanisms.

D.1 Voting for Software Updates

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 *A vote* is a fee-based transaction that is valid 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. 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, or complexity, of the software update.

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 (see Appendix A.2 for a formal definition of a transaction). The core information conveyed by the vote transaction (i.e., included into the transaction Payload field) is summarized in the following tuple:

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

$H(< SIP/UP >$ 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 (see Appendix A.2 for a formal definition of a signature scheme).

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

Anyone who owns stake has a legitimate right to vote and this vote will count proportionally to 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,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.

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 issued 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. 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*.

Suppose 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.

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 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).

On Assumptions and Thresholds We require that for each voting period and for every software update submitted, there is always at least a stake percent of 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. If the underlying consensus protocol has an honest stake threshold x (e.g., $x = 51\%$), then we require $h \geq x$.

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: $stake\ in\ favour \geq h$. 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.

D.2 Delegation

Each stakeholder has the right to participate in the software updates protocol of a proof-of-stake blockchain system. 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.

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.

Delegation for Specialization It is known that there exist special categories of software updates. 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.

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.

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.

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. [9]). 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"([9]). 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 ([9]).

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.

Delegation Mechanics For the realization of the stake pool delegation mechanism that we described above, we closely follow the work of Karakostas et al. [9], 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 [15].

Staking keys. Following the Karakostas et. al. [9] 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 D.2 and it holds only on the assumption 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 (i.e., Payload) of a transaction (see Appendix A.2) 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 (see Appendix A.2 for a formal definition of the signature scheme).

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*[7]. 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.

[7] 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 [15].

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 D.2), 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.
– $\sigma_{sks_{expool}}(m)$: 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 on Delegation

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 [15] 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. [9], 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, real 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 adopted. 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.

E Update Logic

One size does not fit all and surely, all software updates are not the same. There are software updates with totally different characteristics that require a totally different update logic to be applied. For example such characteristics could be:

- The reason of change (a bug-fix, or a security-fix versus a change request, or a new feature request).
- The priority of change (e.g., high/medium/low).
- Impact of change (e.g., whether it impacts the consensus protocol, or not).
- Type of protocol change (hard/soft/velvet[8] fork)
- The complexity of deployment (e.g., high/medium/low)

These and many other characteristics comprise the essential context of a software update and should be clearly described in the respective metadata, which are submitted along the SIPs, or UPs and then reviewed and approved by the stakeholders. We distinguish several categories of metadata that result to a holistic view of a software update. The proposed list of update metadata can be found in the Appendix E.2.

[8] See Zamyatin et. al. [16]

We propose a software update logic that is *metadata-driven*. A logic that distinguishes one SU from another and applies the appropriate *update policy* and evaluates the required *update constraints* (cf. Appendix E.3) based on these metadata.

E.1 Update Policies

An *update policy* is a method to apply a customized activation of a software update driven by its metadata. For example, it is reasonable to assume that a high priority security fix must be activated with a different speed than a "nice to have" new feature. At the same time, software updates with a complex deployment process should take a longer time to activate than others with a simple upgrade procedure.

We have discussed delegation for special categories (cf. Appendix D.2) as a method to speed-up (indirectly) the time to activation for a software update by delegating the approval of such software updates to a specialized group of experts. Moreover, in the previous section we have discussed the concepts of the adoption threshold and the safety lag as the main condition to activate the changes of a software update.

HIGH	**Max Delay** (D, High, High)	(D, Medium, High)	(Sp.D, Low, High)
MEDIUM	(D, High, Medium)	(D, Medium, Medium)	(Sp.D, Low, Medium)
LOW	(D, High, Low)	(D, Medium, Low)	**Min Delay** (Sp.D, Low, Low)
Complexity/ Priority	**LOW**	**MEDIUM**	**HIGH**

Table 1. An example of an update policy based on the SU context.

In Table 1 , we present an indicative example of an update policy that is driven by the software update context. In this example, an update policy is expressed as the triple: (Delegation[9], Adoption Threshold τ_A, Safety Lag T_s). We see for different types of deployment complexities (High/Medium/Low) and approved and agreed priorities (High/Medium/Low), the proposed update policy that will achieve our activation goal. We see the possible values of both the adoption threshold and the safety lag to be expressed in three distinct bands (High/Medium/Low). Of course, all the relevant constraints for these values that we have discussed in the previous section, in order to ensure the security of the activation, should hold.

For example, a low priority software update with a high deployment complexity, should be activated with a maximum delay (top left cell). At the opposite

[9] "D" stands for (standard) delegation while "Sp.D" stands for special category delegation.

end (bottom right cell), a high priority software update (e.g., a security fix) that also bears a low deployment overhead, should be activated with a minimum of delay.

E.2 Decentralized Software Update Metadata

The list of metadata categories presented next is indicative and aims at justifying the concept, therefore it is by no means complete, or restrictive in any way.

Basic information. The software update metadata should provide basic information about the software update, such as its title and a basic description. Also, the metadata should include the unique id of the SU, which can be the content hash of the SU and also the *category tag* that will enable delegation for specialization, as we have described above. Other basic information include, the author of the SU, version information and a link pointing to the storage area, where the software update is stored, as well as a link to the metadata itself. Finally, since in our proposal, we distinguish software updates from SIPs to UPs, the basic metadata information should include a flag to separate one from the other.

Justification. This is a very important part of the metadata of a software update, especially if this is a SIP. It is the information, with which the author of the update will try to convince the rest of the stakeholders on the merit of the proposed software update. Missing information in this part, or unclear justification statements, might cause the rejection of a proposal. The reason, the scope and the expected benefits of a software update must be clearly stated in this part.

Urgency. This is where the priority requirement of a software update must be declared. If it is a bug-fix for example, then it should specify a severity level, in order to declare the urgency for deployment. Similarly, if it is a change request, a priority specification, will help the stakeholders prioritize the demand.

Consensus impact. There are software updates that impact the consensus protocol and others that do not. For the former, it is important to declare the type of the change. If the validation rules of the protocol become less restrictive, then this is a *hard fork* type of change. In this case, non-upgraded nodes reject the new type of blocks and thus there is a significant risk for a chain split. If the validation rules become more restrictive, then this is called a *soft fork* type of change. In this case, the non-upgraded nodes accept the new type of blocks but the blocks generated buy these nodes are not accepted by the upgraded nodes. So, if the old nodes do not eventually upgrade, then they can not continue to issue new blocks. Finally, if the new validation rules are neither more restrictive, nor less restrictive, then we have a *velvet fork* [16]. This type of change does not entail the risk of a chain split. Naturally, the type of change is really important, because it signifies the risk for a *chain split*. For example, a hard fork type of change entails much more risk for a chain split, in the case of nodes failing to upgrade on time, than a soft fork, or a velvet fork (which has no risk at all).

Implementation. Next comes meta-information on the implementation of the software update. Possible examples could be, if this software update entails code development, or if it is a parameter change. For the latter, a distinction could be made between static and dynamic parameter changes. Essentially, dynamic parameter changes are "code-less" software updates - no deployment of new code is required. An explicit list of protocol parameters affected by the software update must be provided. This will be also exploited for the conflict resolution between software updates, mentioned below. Other aspects of the implementation have to do with the size and the complexity of a software update. Typically an estimation of the required man-effort for its implementation could be included.

Deployment. The deployment of a software update is really critical, especially for those updates that impact the consensus protocol. This is because a lack of synchronization at the deployment phase might result into a chain split. This part of the metadata comprises instructions on the deployment process, maybe deployment scripts, or declarative statements of the deployment process (if some automated software provisioning tool is used, e.g., ansible playbooks). Moreover, the *size* of the download-able code must be provided as well as an estimation of the complexity of the deployment that would help to calculate parameters, such as the *safety lag* described in section 3.

Rollback. Symmetrically to the deployment, the metadata must include information for an un-install process, in the case of a problem. The type of information provided is similar to the deployment category.

Update constraints. Software updates do not live in vacuum. They are strongly related to one another. In fact, there exist dependencies between software updates that if they are not followed, then the upgrade will fail. We call all the dependencies and the conflicts between software updates *update constraints*. It is very crucial to clearly define the update constraints of a software update in its metadata. This will illustrate how feasible is a specific software update. For example, a dependency of a SU on another SU, which has not been implemented yet, is a clear indication of non-feasibility. Also, if two candidate software updates both change the value of the same protocol parameter, then they are in direct conflict and some sort of resolution should take place.

Update prerequisites. In this part, we define other prerequisites apart from the ones that have to do with other software updates that we have discussed before and therefore, we can consider them as prerequisites from "external" factors. For example, platform requirements, software requirements (e.g., external libraries), or specific hardware requirements, in order for a software update to be applied successfully, should be mentioned in this section of the metadata.

Budget information. Last but not least, comes the budget information. This is extremely useful information, especially, if the update system is backed up by a treasury system [8] that will assume the funding of the implementation of the software update.

E.3 Update Constraints

With the term *update constraints*, we mean all the prerequisites of a software update for a successful deployment and in particular those that deal with other software updates. More specifically, we are interested in *dependencies* with other software updates and *conflicts* with other software updates.

These constraints should be clearly stated, if possible, even from the ideation phase, where the software update is just a SIP, but certainly, they must be declared in the metadata part of a UP. Essentially, an update constraint is a predicate, which evolves, as the software update matures from an SIP to a UP, and which is evaluated in three phases: a) at the ideation, b)the approval and c) at the activation phase. This evaluation acts as a filter, to protect from software updates that should not be activated due to missing prerequisites, or due to conflicts with other software updates.

Dependencies Commonly, software dependencies are expressed in terms of version requirements. For example, a software update defines in its metadata, on which versions of the software it is applicable and therefore anyone who downloads it, knows if it can be applied, or not. In concept, the proposed method is not far from this idea.

Each instance of the blockchain system is uniquely characterized by a version number. This version number will be based on some versioning scheme, which can be as elaborate as required by the software needs. Every time a software update is activated in the activation phase, the current version of the system changes to some new value. The value that the version will reach after the activation of the software update is recorded in the update metadata. The current version of the system is also evident in each generated block. Finally, for each software update the base version(s) should be clearly stated also in the metadata. These are the versions that this update can be applied to, with no problem.

All in all, for each software update, we know on what versions it is applicable, to what version it will take the system to and what is the current version of the system. These are sufficient information (even in the centralized setting) to build the rules (i.e., the logic) that will evaluate the applicability of a software update with respect to software dependencies.

Conflict Resolution Conflicts of software updates in general have to do with conflicts of different versions of the source code. It is typical when a new version of a software is merged on some other version (main branch) to have a conflict at the source code level. This is typically reviewed and maybe resolved by the code maintainer. In the decentralized setting these type of conflicts are resolved in a similar manner. We assume that each approver in the approval phase, maintains a local code repository and tries to verify the (metadata-declared) applicability of a software update on the indicated version of the code, by a simple merge operation. If this merge raises a conflict, then the approver must reject the software update.

More specifically in blockchain systems, there is another type of conflicts that is of particular importance. These are the conflicts that are related to the consensus protocol parameters. For example, a software update increases the maximum transaction size and another software update decreases it. These two SUs are clearly in conflict. However, they are in conflict only if the are both *open* at the same time, otherwise they are just a valid sequence of changes of the system. We call a software update as *open*, if it is in some of the four phases comprising the lifecycle of a SU, namely: the ideation, the implementation, the approval, or the activation phase. If there are two (or more) SUs, in any of these phases and they impact the same protocol parameters (which is a metadata-recorded piece of information), then we have a conflict and must perform some resolution action. The resolution action might vary based on the needs of the system and the context of the software update. One possible action would be to reject all conflicting SUs. Another possible action would be to reject all conflicting SUs, except one (if there is only one), at the latest phase. All these decisions, have to do with the criticality of a software update, the type of the software update and other parameters, which should all be defined in the software update metadata, in order to help to define some update policy. So we see once more, how it is possible with an appropriate definition of software update metadata, to drive also the update constraint evaluation process.