Scalable, Confidential and Survivable Software Updates

Scalable, Confidential and Survivable Software Updates

Federico Magnanini*, Luca Ferretti†, Michele Colajanni‡

Abstract—Software update systems must guarantee high availability, integrity and security even in presence of cyber attacks. We propose the first survivable software update framework for the secure distribution of confidential updates that is based on a distributed infrastructure with no single points of failure. Previous works guarantee either survivability or confidentiality of software updates but do not ensure both properties. Our proposal is based on an original application of a multi-authority attribute-based encryption scheme in the context of decentralized access control management that avoids single-point-of-vulnerability. We describe the original framework, propose the protocols to implement it, and demonstrate its feasibility through a security and performance evaluation.

Index Terms—Software updates, survivability, transparency, proprietary software

1 Introduction

The ability of deploying efficient and secure software updates is one of the most critical aspects of any modern information system. Although update systems have always existed, they do not ensure high security against advanced attacks [1], [2], [3]. Numerous efforts have identified that essential security properties of software updates are *authenticity*, *freshness* and *transparency* [4], [5], [6], [7], [8], and that software update systems must guarantee *availability* and provide *fast*, *resilient* and *scalable* dissemination of software updates to ensure prompt application of security patches [9], [10], [11].

We focus on proprietary software update systems, that impose additional design constraints that do not characterize open source software update systems. In particular, proprietary software update systems must guarantee also *access control* to prevent unauthorized clients from installing unauthorized updates, and *confidentiality* of software updates to protect from reverse engineering. Moreover, we aim at a highly secure system that provides two essential properties, that are *recoverability*, that allows administrators to rapidly recover the system to a safe state after a security incident, and *survivability*, that guarantees security even if the software update system is partially compromised [6], [7], [12], [13], [14]. *Survivability* implies avoiding the presence of single points of failure or vulnerability within the system, making it more difficult for an attacker to compromise software updates in any part of the software development or distribution process.

Previous proposals only satisfied subsets of the mentioned security requirements without presenting a unified solution for the distribution of proprietary software updates [7], [8], [11]. Some proposals focus on open source software, thereby not considering access control and confidentiality requirements [7], [8]. Other work focuses on confidentiality and access control but does not consider survivability and recoverability requirements [11].

To bridge this gap in the literature, we propose a novel comprehensive framework that is able to provide the security guarantees that modern software update systems for proprietary software should have. In particular, we design the first *survivable* framework for the secure distribution of confidential updates for proprietary software that satisfies availability, scalability, resiliency, survivability and recoverability requirements, while guaranteeing authenticity, confidentiality, freshness, timeliness and transparency of software updates to clients. The framework enforces fine-grained access control policies over untrusted distribution infrastructures, to comply with distinguished business driven practices. To this aim, our proposal includes two novel contributions of independent interest. First, we extend existing Multi-Authority Attribute-Based Encryption schemes through a novel technique that allows survivable generation of decryption keys so that compromising a threshold of key-generating actors does not allow attackers to violate update confidentiality. Second, we design a novel protocol that allows distributed authentication and encryption of software updates without single points failure. We demonstrate the practicality of the proposed framework through a performance evaluation of our novel Multi-Authority Attribute-Based Encryption extension and distributed authentication and encryption protocol.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the system and threat model. Section 4 outlines the overall design. Section 5 describes the details of each operation. Section 6 discusses the security of the proposed system. Section 7 evaluates performance and costs. Section 8 reports conclusions and future work.

* *Department of Engineering "Enzo Ferrari", University of Modena and Reggio Emilia, Italy, federico.magnanini@unimore.it*
† *Department of Physics, Computer Science and Mathematics, University of Modena and Reggio Emilia, Italy, luca.ferretti@unimore.it*
‡ *Department of Computer Science and Engineering, University of Bologna, Italy, michele.colajanni@unibo.it*

2 Related work

This paper proposes the first survivable software update framework that integrates all five attributes that should characterize software updates (authenticity, availability, freshness, transparency and confidentiality) and ensures all

five guarantees of a software update framework (to be fast, scalable, resilient, survivable and recoverable) that does not use a trusted third party for software distribution. In the following we highlight our original contributions over previous proposals, which involve the attributes of *survivability*, *confidentiality* and *authenticity*.

Confidentiality of software update binaries at rest and in motion is important to protect software updates from automatic exploit generation [15]. To guarantee confidentiality of software updates on untrusted distribution infrastructures, related works adopt different types of encryption schemes. The proposal of [10] makes black-box use of symmetric encryption to encrypt updates with a single symmetric key to allow scalability in the number of clients. The symmetric key is then broadcast to clients to allow decryption. The proposal does not protect the confidentiality of the key during broadcast and therefore is not suitable for proprietary software. The authors in [11] adopt the Ciphertext-Policy Attribute-based Encryption (CP-ABE) scheme proposed in [16] to protect the symmetric key by producing a single ciphertext for all clients. However, in both proposals the key generation procedure of the adopted encryption schemes is not designed to be distributed. This choice represents a single point of failure for the security of the system which, if compromised, would allow the attacker to issue new keys and violate the confidentiality of past and possibly future updates. For this reason, the approaches proposed in [10] and [11] do not guarantee full survivability. We enhance the survivability of the framework by decentralizing the decryption key generation process by extending the Multi-Authority Ciphertext-Policy Attribute-Based Encryption (MA-CP-ABE) scheme of [17]. In particular, with an appropriate choice of encryption policies we can tolerate the compromise of a threshold of key-generating actors while guaranteeing the confidentiality of previous and future updates. As a result, our proposal is the first survivable software update system which can guarantee confidentiality of software updates. Furthermore, our original extension is fully *recoverable* as, once a compromise is detected, administrators can easily restore the system to a safe state, thereby ensuring service continuity.

Software updates frameworks should allow distribution through untrusted infrastructures, hence it is mandatory to guarantee end-to-end software *authenticity* [4]. The authors of [7] guarantee end-to-end authenticity of software updates through a public and permissioned blockchain that stores authenticated update metadata. However, their proposal cannot guarantee authenticity of confidential software updates because it is designed for open-source software. To guarantee authenticity of confidential software updates, we improve over [7] in multiple ways. First, we design a novel distributed protocol that authenticates encrypted software updates without single points of failure and that allows clients to verify that any update has been approved by a number of authorized actors by means of multi-signatures. Second, we extend the architecture of their proposed blockchain to account for additional roles required to authenticate confidential updates. Moreover, our proposal extends their proposed blockchain to include the due authenticated, survivable and non-equivocable mechanisms and procedures that indicate the software update location to clients. These mechanisms and procedures offer to system administrators the flexibility of choosing and changing the update location and the distribution infrastructure operator as needed. These possibilities are not provided by the authors in [7] that implicitly assume a way of authenticating the update location and do not provide mechanisms to authenticate a location change.

Finally, we integrate our original contributions by extending the architecture and ideas of [7], which allow our proposal to inherit the attributes of *availability*, *freshness*, *timeliness* and *transparency*. Moreover, our proposal guarantees *fast*, *scalabile* and *resilient* dissemination of software updates by adapting the strategy introduced in [11] of producing a single ABE ciphertext per update, to our protocols based on Multy-Authority ABE.

3 System and threat model

3.1 System model

The typical scenario for proprietary software update systems involves three entities: *software house*, *(software) distribution infrastructures* and *clients*.

The *software house* includes a set of roles that share the same interests. Within the software house, we denote as *developers* a set of employees that can access source code, compile it, produce software binaries, and are responsible for approving new software versions that are identified by increasing alphanumerical strings. Depending on the characteristics of the software house, this role can be accomplished by actual developers or by other specialized personnel, such as that dedicated to DevOps practices. The software house approves updates that must be delivered to clients. Each update consists of binaries and related source code.

The software house relies on *distribution infrastructures*, which could be managed by third parties such as Content Delivery Networks (CDN) or community-managed mirror servers. The software house also defines access control policies over software updates which are enforced by a trusted distribution infrastructure.

Finally, clients represent the devices that store and execute the version of the installed software binaries produced and maintained by the software house. Clients periodically query the distribution infrastructure to check whether a new update is available and, if so, they can download it from the distribution infrastructure.

In the following we describe our proposal's system model. We extend the reference software house by adding three roles: *admins*, *authentication server* and *Attribute-based Encryption* (ABE) servers. We consider a software house with n_d developers, n_a admins, n_r ABE servers and one authentication sever. Each developer $d \in [n_d]$ has a signing key pair $dk_d = \langle sk_d, pk_d \rangle$ and has access to the software source code (src). Admins are responsible for managing security-critical cryptographic material for authenticating roles and enforcing access control policies. Each admin $a \in [n_a]$ has a signing key pair $ak_a = \langle sk_a, pk_a \rangle$ and has access to the access control policies ($\mathbb{P}$) that must be used for encrypting software update binaries through MA-CP-ABE. Authentication and ABE servers are responsible for

managing cryptographic material. ABE servers are authorities responsible for issuing ABE keys to authenticated clients for decrypting software updates. Each ABE server $r \in [n_r]$ has an ABE authority key pair $rk_r = \langle sk_r, pk_r \rangle$ and the set $\mathcal{A}_{\delta(r)}$ of attributes to compute the decryption keys for clients. The authentication server maintains the database of registered clients and the corresponding authentication information, and assigns attributes to clients after a successful authentication.

The distribution infrastructure maintains encrypted software updates (encrypted binaries) associated with location information that is used by other parties to retrieve updates. Our proposal relies on untrusted distribution infrastructures by making use of Multi-Authority Ciphertext-Policy Attribute-Based Encryption (MA-CP-ABE) [17], which guarantees confidentiality and enforces policy-based access control over software updates even on untrusted distribution infrastructures. In this scheme an authority issues to clients one or more private keys each encoding an attribute. The encryption algorithm accepts a message and a policy expressed as a monotonic boolean formula over attributes, and produces a ciphertext. A client is able to decrypt the ciphertext if the attributes of his private keys satisfy the boolean formula associated to the ciphertext. To this aim, each registered client has a set SK_{cid} of ABE keys for decrypting software updates.

Finally, our system requires two additional roles that are inherited from the scenario in [7]: *validators* and *witnesses*. Validators audit and validate new software updates through reproducible builds. Witnesses are nodes of a *multi-layer skipchain* which is an authenticated append-only data structure introduced by [7], that stores public cryptographic material and software update metadata. Witnesses share the same version of the multi-layer skipchain by using a Byzantine-fault-tolerant state-machine-replication consensus algorithm.

The proposed system considers n_v validators and n_w witnesses. Each validator $v \in [n_v]$ has a signing key pair $vk_v = \langle sk_v, pk_v \rangle$ and a signed copy of the source code for update validation. Each witness $w \in [n_w]$ has a signing key pair $wk_w = \langle sk_w, pk_w \rangle$ and maintains a copy of the *multi-layer skipchain*.

3.2 Threat model

We consider an attacker that may be interested in violating confidentiality, authenticity, availability or integrity of software updates. Violating confidentiality means that the attacker can reverse engineer the update and look for vulnerabilities in the previous or in the update version. Compromising the authenticity and integrity of updates may induce clients to download and install backdoored software versions. Denying an update forces a client to keep an outdated software version which may contain vulnerabilities that an attacker can exploit.

Our proposal protects software updates binaries and clients against the mentioned attacks by using cryptographic protocols, and assuming a computationally bound attacker which is unable to break the security of the adopted protocols or the security of their underlying cryptographic primitives.

We inherit the following threshold assumptions from [7]. We assume that all actors communicate over authenticated channels that can be eavesdropped by the adversary. Survivability is guaranteed through threshold variants of cryptographic schemes.

We assume that no more than a threshold of $t_d - 1$ out of n_d developers is malicious. A developer is malicious if he colludes with the attacker, if his signing key has been compromised or if the attacker has compromised other parts of the developers' systems, such as by covertly installing a compromised compiler.

We assume that no more than a threshold $t_a - 1$ out of n_a *admins* is malicious and that no more than $t_r - 1$ out of n_r *ABE servers* is malicious. We assume that the *authentication server* is honest and that the attacker is unable to compromise it. Furthermore, we assume that the authentication mechanism adopted to authenticate clients is secure.

We assume that no more than $t_v - 1$ *validators* and no more than $t_w - 1 = \lfloor n_w/3 \rfloor$ *witnesses* are malicious. These thresholds protect the correctness of the data inserted by admins in the multi-layer skipchain and the security of its consensus mechanism, that is executed by witnesses. Moreover, we assume that validators do not leak source code. Indeed, since validators must receive the project source code in plaintext form to validate it, developers must trust all validators not to collude with the adversary. The validator role represents a trade-off between source code confidentiality and transparency of software updates. To the best of our knowledge, enabling software transparency without relying on third parties is still an open problem.

The distribution infrastructure may be untrusted that is, attackers can replace, eavesdrop and modify software updates during its distribution, but they cannot impede availability for an unbounded amount of time.

A client with proper attributes can decrypt the update, and an attacker that compromises or disguises as a client can obtain access to the decrypted update and reverse engineer it. To protect against malicious clients, it is possible to adopt orthogonal solutions, such as patch obfuscation, to thwart reverse engineering of the released binary. However, protections against similar threats are out the scope of this paper. We assume that the client is honest and that an attacker cannot break the update confidentiality by compromising or impersonating a client.

4 Framework Design

4.1 Framework components and operations

We describe the proposed framework for secure software updates by referring to Figure 1. This figure represents the components of the system and its main operation flows. The secure software distribution framework includes nine operations: (1) update authentication, (2) update validation, and (3) publish are used by the software house members to make a new update available for clients. The operations (5) check for update and (6) download and decrypt update are used by clients to detect and obtain new released updates, and possibly detect attacks. The other operations are used for additional security-related tasks: (4) client key refresh allows clients to obtain new decryption keys; (7) key management, (8) update policies and (9) change location allow admins to

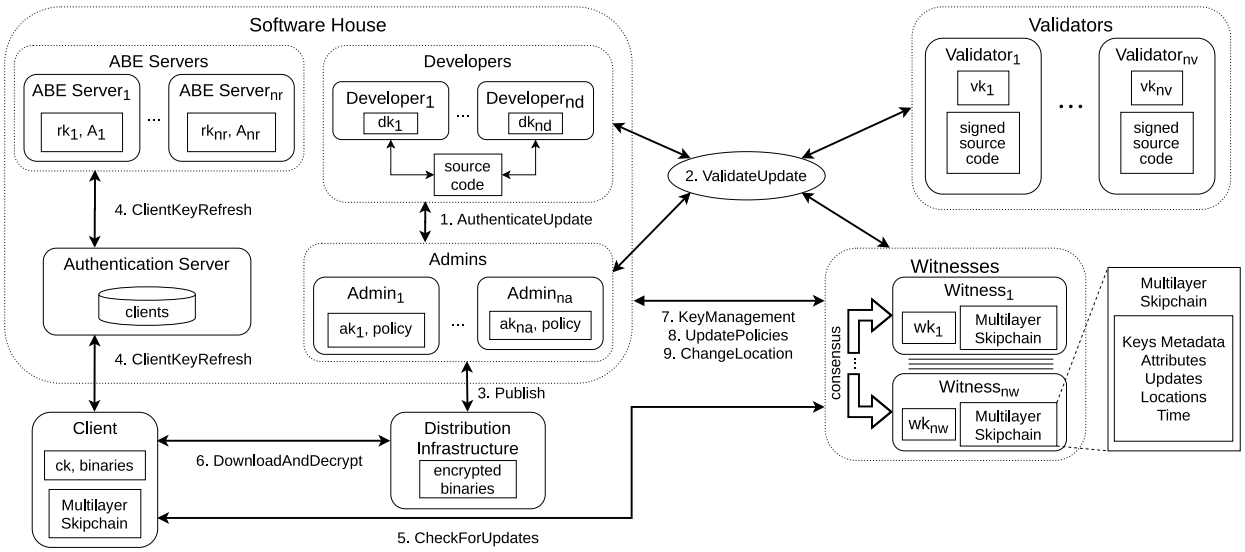

Fig. 1. Architecture of the framework for secure software updates

rotate public keys, ABE attributes and update the location of the encrypted binaries, respectively.

These operations are executed as follows. When a new update is ready, admins and developers authenticate it (1) and send the authenticated update to validators that check the source-to-binary correspondence and return an authenticated validity attestation (2). Admins append to the multi-layer skipchain the authenticated metadata and update attestations, and publish the authenticated binaries (3) to the distribution infrastructure. A client, who has already obtained a valid set of ABE keys (4), checks whether new updates are available (5) by querying the multi-layer skipchain. If available, the client downloads them from the distribution infrastructure and decrypts them with his ABE keys (6). In case the new update requires a policy change, admins execute the update policies procedure (8) before authenticating the update. If keys of any role need to be rotated because of a security incident or because of key expiration, admins execute the key management procedure (7). If admins need to change the location of the latest update, for example due to change in distribution infrastructure provider, admins execute the change location procedure (9).

In the following two subsections, we outline the operations of the multi-layer skipchain and of the information flow, respectively. Details of each operation are described in Section 5.

4.2 Multi-layer skipchain

The witnesses maintain a multi-layer skipchain that guarantees freshness and non-equivocation of software update metadata to clients and that allows admins survivable and authenticated modification of the corresponding cryptographic material. Our original design extends that proposed in [7] and consists of eight layers shown in Figure 2. Layers are stacked in the following order and are identified by labels: admins (L_{ad}), witnesses (L_w), validators (L_v), ABE servers (L_r), attributes (L_{at}), update (L_u), location (L_l), time (L_t). Each layer has specific constraints that newly appended data must satisfy.

The first four layers (admins, witnesses, verifiers, ABE servers) store public keys of admins, witnesses, verifiers, and ABE servers, respectively. They allow admins to rotate the public keys of these actors. We collectively refer to these layers as *keys metadata* layers. A new set of public keys can be added to any of the keys metadata layers only if it is authenticated by at least a threshold of admins.

The attributes layer maintains the set of ABE attributes that are adopted in the access control policy used to encrypt the latest update with ABE. A new set of attributes can be added to this layer only if the set is authenticated by at least a threshold of admins. This layer allows admins to change the set of valid attributes when needed, and allows clients to receive the list of valid ABE attributes to determine if they need to refresh their ABE keys (for details about ABE key refresh, see Section 5.6). We note that admins cannot change the set of ABE attributes of already released encrypted binaries because the skipchain is append-only. Any change would have no effect on existing ciphertexts already published in the distribution infrastructure. A policy change, and consequently an attribute change, has effect only on future updates.

The update layer maintains metadata about encrypted binaries. New update metadata can be added to the update layer only if they have been authenticated by a threshold of admins, developers and validators. In particular, validators are in charge of verifying the correspondence between source code, encrypted binaries and metadata (see Section 5.5).

The location layer maintains authenticated location information, such as a URL, about how to retrieve encrypted update binaries from the distribution infrastructure. New location information can be added to the location layer only if it is authenticated by at least a minimum threshold of admins. This layer allows admins to change the update location when necessary.

The time layer maintains multi-signed, timestamped hash pointers to the location layer, that are periodically computed by witnesses. A new timestamp can be added to the time layer only if it is authenticated by at least a

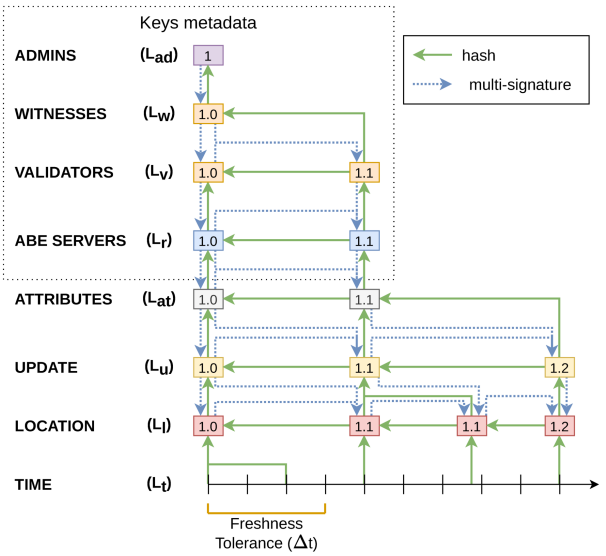

Fig. 2. The multi-layer skipchain for secure software updates

minimum threshold of witnesses and only if the timestamp is greater than the previous timestamp, and not too far into the future. Assuming that a client has a trusted and reliable time source, this layer allows clients to detect freeze attacks by comparing the most recent timestamp with the client's current timestamp, and verifying that the difference between the two timestamps is within an upper bound (*freshness tolerance*) that is determined by the admins in the update metadata. We assume that a client can safely bootstrap his copy of the multi-layer skipchain by securely obtaining the admins public keys included in the genesis skipblock of the admins layer.

4.3 Information flow

We describe the main operations supported by the system. Details of each operation are reported in Section 5 and 6.

Setup. The goal of the setup operation is to authenticate the public keys of all actors in the system. This operation is orchestrated by admins who act as trust anchor. Each admin, witness, validator, ABE server and developer generates a key pair and maintains the secret key confidential. Then, admins act as certification authorities and gather the public keys of witnesses, validators and ABE servers. They collectively authenticate them through multi-signatures and by assigning validity periods that produce key metadata. Finally, admins store key metadata in the appropriate skipchain layers.

Update policies. When the software house defines a new access control policy or modifies an existing policy or when admins rotate the ABE servers keys, admins compute a new set of attributes and assign each attribute to an ABE server. Admins communicate the new set of attributes to the authentication server and append the new set to the attributes skipchain layer.

Update authentication. When a new software update is ready to be released, the source code must be approved by validators. To this aim, developers and admins must issue to them the source code together with cryptographic material that assesses the compilation and encryption procedures used to produce the encrypted binaries distributed to clients. The overall procedure includes five phases. A threshold of admins authenticates the access control policy ($\mathbb{A}$), the freshness tolerance value that must be enforced on the released software, and the set of developers that are authorized to sign the update source code. Then, the same admins communicate the authenticated access control policy, the freshness tolerance and the developers public keys to each authenticated developer. After these operations, a threshold of developers compiles the source code through reproducible builds procedures, and all of them obtain the same binary data. To obtain the same encrypted binaries, the same developers collaboratively generate a shared secret cryptographic key and encrypt the resulting software binaries through a deterministic symmetric encryption procedure. The developers also compute a digest of the encrypted binaries and ABE-encrypted key. Finally, the involved developers use a multi-signature scheme to authenticate the source code and the generated public cryptographic material.

Update validation. Admins send the authenticated source code, the encryption key and authentication material to validators for validation. Validators compile the source code through reproducible build procedures and encrypt it through deterministic encryption using the received encryption key. Then, they verify the correctness of the resulting encrypted binaries against the received authentication material. If verification is valid, the validators apply a multi-signature to the received authentication material by interacting with admins according to the required validation phase of the consensus protocol used by witnesses. Finally, admins send the location information, the authentication material and the validators multi-signature to witnesses to update the multi-layer skipchain.

Publish. Any admin can publish the encrypted software binaries to the distribution infrastructure at the location inserted in the multi-layer skipchain. We observe that the encrypted binaries could also be published before the completion of the validate update algorithm to ensure the availability of the update to clients as soon as the multi-layer skipchain is updated.

Client key refresh. To decrypt the encrypted binaries, a client must obtain valid ABE keys from ABE servers of the software house. We describe the procedure by assuming, without loss of generality, that clients are already registered in the authentication server's clients database. Each client authenticates himself at the authentication server. This server determines the ABE attributes that qualify it and sends them to a threshold of ABE servers. ABE servers respond with ABE keys and the authentication server forwards them to the client. We highlight that the keys obtained by a client are related to the access control policy used to encrypt binaries, and can be reused for unlimited software releases as long as admins do not operate key rotations that invalidate the client decryption keys, or use new attributes for encrypting new software releases that are mandatory to decrypt binaries. Only in these cases, and only if the client still complies to the software house policies, a client must obtain new ABE keys by re-executing the client key refresh procedure.

Check for updates. Clients periodically check for updates by requesting to witnesses the last skipchain updates. Clients must always be able to obtain a response from

witnesses, and the response must always include updated timestamps of the time layer. A missing response or a response with stale or old time information are considered as violations of the system availability, possibly due to ongoing attacks. Clients verify the authenticity and integrity of the received skipblocks by using admins and witnesses public keys, and verify the authenticity of skipblocks payloads by using admins, validators and developers public keys. A new software release is available if and only if there is a new authentic skipblock in the update layer. In this case, clients extract the most-updated location information and can proceed to obtain encrypted binaries.

Download and decrypt. Clients use the obtained location information to query the distribution infrastructure for the encrypted binaries. We assume that the distribution infrastructure is always available and responds to client queries. Clients download the encrypted binaries, verify their authenticity and decrypt them by using ABE decryption keys to obtain plaintext software binaries. At this point, the clients can install the software update binaries.

Key management. Occasionally, admins may have to rotate, revoke or issue new keys to substitute compromised keys, replace or remove misbehaving actors or add new actors to the system. Admins rotate, revoke and issue keys by collectively authenticating a new set of public keys and sending it to witnesses that update the appropriate keys metadata layer.

Change location. Finally, admins may need to change the location of an update. Admins can notify clients about the location change by appending new location information in the corresponding layer of the skipchain. Thanks to the design of the skipchain layers, this operation does not require any modification to the update metadata and can be operated without the intervention of the validators.

5 Framework Details

We describe the details of the proposed framework. For lack of space and simplicity, we assume that the parameters of the adopted schemes are already defined, including elliptic curves, symmetric ciphers and different types of hash functions (standard hash functions and those required to produce elliptic curve points). Moreover, we omit technicalities such as translations between monotonic boolean formulas, used in this paper, and monotone span programs, on which ABE schemes are based (see [18] for details).

5.1 Adopted schemes and notations

We describe the operations frameworks of the adopted Multi-Authority Ciphertext-Policy Attribute-Based Encryption scheme (MA-CP-ABE) [17], multi-signatures scheme [19] and multi-layer skipchain (see Section 4.2).

MA-CP-ABE includes the following algorithms:

$\langle sk_r, pk_r \rangle \leftarrow \mathsf{AuthSetup}(r)$: is a randomized algorithm executed by each authority $r \in \mathcal{R}$ where $\mathcal{R}$ is the set of authorities identifiers. The algorithm returns a key pair $\langle sk_r, pk_r \rangle$.

$sk \leftarrow \mathsf{KeyGen}(cid,\ sk_r,\ e_r)$: is a randomized algorithm that takes a client global unique identifier cid, authority's r secret key sk_r and an attribute e_r. The algorithm is executed by the authority r responsible for attribute e_r and returns a secret key sk for attribute e_r to the client identified by the global unique identifier cid.

$c \leftarrow \mathsf{Encrypt}_{\mathsf{ABE}}(m, \mathbb{A}, \{pk_r\})$: is a randomized algorithm that takes a message m, an access-control policy $\mathbb{A}$ expressed as a monotonic boolean formula over attributes, and the set of public keys $\{pk_r\}$ of the authorities that control at least an attribute in the policy. The algorithm, which can be publicly executed, returns the ciphertext c.

$m \leftarrow \mathsf{Decrypt}_{\mathsf{ABE}}(\{sk\},\ c)$: is a deterministic algorithm run by a client that takes the set of secret keys $\{sk\}$ and a ciphertext c. The algorithm returns plaintext m only if the attributes associated with the client's secret keys satisfy the ciphertext policy.

Multi-signatures allow any subgroup L of a group of n players $\{V_1, \ldots, V_n\}$ to collectively sign a message m and prove to a verifier that all members of L participated in producing the message signature. We consider using the scheme proposed in [19] based on BLS signatures [20], that includes the following three algorithms.

$\langle sk_i, pk_i \rangle \leftarrow \mathsf{KeyGen}(1^\lambda)$: is a randomized key generation algorithm run by each player V_i that returns key pair $\langle sk_i, pk_i \rangle$.

$\langle L, \sigma, m \rangle \leftarrow \mathsf{MultiSign}(\{sk_i\}_{i \in L},\ m)$: is a possibly randomized two-step interactive algorithm run by any subset of players $L \subseteq \{V_1, \ldots, V_n\}$.

$b \leftarrow \mathsf{Verify}(\{pk_i\}_{i \in L}, \sigma, m)$: is a deterministic algorithm run by the verifier and outputs $b = 1$ if and only if signature σ has been generated with $\mathsf{MultiSign}(\{sk_i\}_{i \in L},\ m)$, 0 otherwise.

Multi-layer skipchain includes the following algorithms:

$\sigma \leftarrow \mathsf{Append}(l, d)$: is an interactive algorithm executed between a client and the nodes maintaining the multi-layer skipchain. The algorithm, started by the client, takes a layer identifier l and the data d the client wants to append to layer l. If data d satisfies the constraints of layer l, then d is added as payload of layer's l new skipblock and nodes return to the client a multi-signature σ of the pair $\langle l, d \rangle$ to confirm that d has been appended to layer l.

$(\pi, S) \leftarrow \mathsf{GetLatestSkipblocks}(t)$: is an interactive algorithm executed between the client and the nodes maintaining the multi-layer skipchain. The algorithm, started by the client, takes the latest timestamp t indicating the last time the client updated the multi-layer skipchain, and returns the set of skipblocks S the client lacks, and a proof π that the set S is valid and fresh. In particular, the proof π contains the latest timestamp t' of the *time* layer, and the minimum set of forward pointers and *witnesses* layer skipblocks required to validate the set S.

$0, 1 \leftarrow \mathsf{Validate}(\mathsf{now}, \langle \pi, S \rangle)$: is an algorithm executed by the client that takes the client's current time, denoted as now, and the pair $\langle \pi, S \rangle$ obtained by the client with $\mathsf{GetLatestSkipblocks}(\cdot)$. The algorithm returns 1 if the proof π for skipblocks S is valid and if the difference between now and the proof timestamp t' is within a certain threshold, 0 otherwise. We note that the threshold value may be an application-defined value included in skipblocks payload.

5.2 Setup

We model the setup operation as:

$$\mathsf{Setup}(1^\lambda) \tag{1}$$

where 1^λ is the security parameter.

Key metadata attestations are attached to multi-signatures to demonstrate the chain of trust between signers and *admins*, and include metadata established by *admins* to define authenticity threshold requirements. We define the *Key Metadata Attestation* (KMA) as follows:

$$\mathsf{KMA} = \langle \langle \{pk\}, s, v_b, v_e \rangle, t_a, \sigma_{\mathsf{KMA}} \rangle \tag{2}$$

$$\sigma_{\mathsf{KMA}} \leftarrow \mathsf{MultiSign}(SK,\ \langle \langle \{pk\}, s, v_b, v_e \rangle, t_a \rangle\ ),\quad SK \subseteq \{sk_a\} : |SK| \geq t_a \tag{3}$$

where s is the threshold on the minimum number of public keys in $\{pk\}$ used to verify multi-signatures, v_b and v_e denote the begin and end of the attestation validity period and t_a is the threshold of signing *admins* required to update the set $\{pk\}$ in key rotation procedures. KMA is valid only if σ_{KMA} is a valid multi-signature computed by at least t_a *admins*, and $v_b < v_e$.

Each *admin* generates his multi-signature key pair:

$$\langle sk_a, pk_a \rangle \leftarrow \mathsf{KeyGen}(1^\lambda) \tag{4}$$

Admins compute the *admins* KMA (AKMA), which includes the *admins* public keys, as following:

$$\mathsf{AKMA} = \left\langle \langle \{pk_a\}_{a \in [n_a]}, t_a, v_b, v_e \rangle, t_a, \sigma_{\mathsf{AKMA}} \right\rangle \tag{5}$$

We note that, in this particular case, AKMA has the further constraint of requiring verification by using a subset of the public keys included in the attestation itself, similarly to self-signed root certificates in PKI systems. We highlight that the signing operation denotes a distributed protocol for computing the multi-signature among mutually untrusted parties, in this case *admins*.

Witnesses generate their multi-signature signing keys:

$$\langle sk_w, pk_w \rangle \leftarrow \mathsf{KeyGen}(1^\lambda) \tag{6}$$

and send their public keys $\{pk_w\}$ to *admins*. *Admins* compute the *witnesses* KMA (WKMA), which includes the *witnesses* public keys, as following:

$$\mathsf{WKMA} = \left\langle \left\langle \{pk_w\}_{w \in [n_w]}, t_w, v_b, v_e \right\rangle, t_a, \sigma_{\mathsf{WKMA}} \right\rangle \tag{7}$$

where $t_w = \lfloor n_w/3 \rfloor + 1$.

Validators generate their multi-signature signing keys:

$$\langle sk_v, pk_v \rangle \leftarrow \mathsf{KeyGen}(1^\lambda) \tag{8}$$

and send their public keys $\{pk_v\}$ to *admins*. *Admins* compute the *validators* KMA (VKMA), which includes the *validators* public keys, as following:

$$\mathsf{VKMA} = \left\langle \left\langle \{pk_v\}_{v \in [n_v]}, t_v, v_b, v_e \right\rangle, t_a, \sigma_{\mathsf{VKMA}} \right\rangle \tag{9}$$

In the following we denote as $r \in \mathcal{R}$ the identifier of an *ABE server*, where $\mathcal{R}$ is the set of identifiers of all *ABE servers*. Each *ABE server* generates his key pair $\langle sk_r, pk_r \rangle$:

$$\langle sk_r, pk_r \rangle \leftarrow \mathsf{AuthSetup}(r)\ \forall r \in \mathcal{R} \tag{10}$$

All *ABE servers* send their public keys $\{pk_r\}$ to *admins*. *Admins* compute the *ABE servers* KMA (RKMA), which includes the *ABE servers* public keys, as following:

$$\mathsf{RKMA} = \left\langle \langle \{pk_r\}_{r \in \mathcal{R}}, t_r, v_b, v_e \rangle, t_a, \sigma_{\mathsf{RKMA}} \right\rangle \tag{11}$$

Any *admin* initializes the appropriate skipchain layer by executing $\mathsf{Append}(\cdot, \cdot)$, as described in Section 5.1, using all metadata obtained so far (AKMA, WKMA, VKMA, RKMA). The *admin* then verifies that the returned multi-signatures are valid and that the set of signers is a subset of the *witnesses* specified in WKMA.

Developers generate their multi-signature signing keys:

$$\langle sk_d, pk_d \rangle \leftarrow \mathsf{KeyGen}(1^\lambda) \tag{12}$$

and send their public keys $\{pk_d\}$ to *admins*. We note that *developers* public keys are authenticated during the *update authentication* procedure (see Section 5.4).

We highlight that *admins*, *developers*, *witnesses* and *verifiers* also generate the due cryptographic keys to establish authenticated and confidential point-to-point communication channels, and to generate the required cryptographic material in following phases. For ease of exposition we do not specify their generation and usage as we rely on well known cryptographic primitives.

5.3 Update policies

We model the update policies procedure as $\mathsf{UpdatePolicy}(\mathbb{P}, v, t_r, \mathcal{R})$, where $\mathbb{P}$ denotes the access control policy, v denotes the minimum version number to which the policy applies, t_r denotes the security threshold for *ABE servers*, and $\mathcal{R}$ is the set of *ABE servers*.

The procedure must be used before releasing the first software update, before releasing a new software update that requires a novel policy, or whenever *ABE servers* public keys are rotated. It transforms access control information defined by the *software house* based on a single-authority paradigm to the multi-authority setting of the proposed architecture. It is composed of two phases:

- *attribute derivation* transforms the access control attributes and produces an *authenticated attributes matrix* $\mathcal{A}$ that assigns multi-authority attributes to each ABE server. The matrix $\mathcal{A}$ is also appended to the *attributes* layer of the multi-layer skipchain to be available for *clients* to detect whether a key refresh is needed (Section 5.6);
- *policy translation* transforms the single-authority policy $\mathbb{P}$ into the multi-authority policy $\mathbb{A}$, which is used in the *update authentication* procedure (Section 5.4).

We observe that *attribute derivation* must be operated only in case of policies modifications that use novel attributes or in case of ABE servers key rotations. Moreover, we note that introducing novel attributes and rotating ABE servers keys does not require re-executing the setup procedure because the ABE scheme adopted in our proposal does not fix the set of ABE servers and attributes during its setup (see Section 5.1 for details about the ABE scheme, and Section 5.9 for details about ABE servers key rotation). This is a very important property because it enables recoverability as we discuss in Section 6. We highlight that, to the best of our

knowledge, the proposed approach in *attribute derivation* and *policy translation* procedures is the first practical solution that enables a survivable generation of ABE keys.

Attribute derivation. *Admins* receive the set of the attributes that are used by the *software house* to define the access control policy $\mathbb{P}$. In the following, we denote these attributes as *original attributes* for disambiguation, and we model them as binary strings of potentially variable length. To guarantee survivability, all *ABE attributes* of each *ABE server* must be associated to *original attributes* in a bijective relation. To this aim, *admins* enumerate all of *original attributes* in an ordered set that we denote as P (e.g., by sorting them with lexicographic comparisons). We denote as $p_j \in P$ the j^{th} *original attribute*, where $j \in [|P|]$.

Given the set P of all *original attributes* and the set $\mathcal{R}$ of all *ABE servers*, a selected *admin* computes the set of *ABE attributes* assigned to each *ABE server* as following. To this aim, he generates the *ABE attributes matrix* $\mathcal{A}$ of size $|\mathcal{R}| \times |P|$. Each *original attribute* p_j is mapped to column j. Moreover, we assume that a function $\delta(\cdot) : \mathcal{R} \rightarrow [|\mathcal{R}|]$ exists to map each *ABE server* $r \in \mathcal{R}$ to a row of the matrix. Each element $\alpha^j_{\delta(r)}$ in the matrix $\mathcal{A}$ is computed as the concatenation of the public key of the row's *ABE server* with the column's *original attribute*:

$$e_{r,j} := pk_r || p_j, r \in \mathcal{R}, j \in [|P|] \tag{13}$$

$$\mathcal{A} := \left(\alpha^j_{\delta(r)} : \alpha^j_{\delta(r)} \leftarrow e_{r,j}\right), \forall r \in \mathcal{R}, j \in [|P|] \tag{14}$$

Uniqueness of public keys implies uniqueness of *ABE attributes*. All *ABE attributes* that differ only for the public key part are syntactically different and semantically equivalent. The j^{th} column of $\mathcal{A}$, denoted as $\mathcal{A}^j$, contains all semantically equivalent representations of *original attribute* p_j assigned to different *ABE servers*. The $\delta(r)^{th}$ row of $\mathcal{A}$, denoted as $\mathcal{A}_{\delta(r)}$, contains all *ABE attributes* of *ABE server* r.

Admins compute the *Authenticated Attributes Map* (AAM) as:

$$\mathsf{AAM} := \langle \langle \mathcal{A}, v \rangle, \sigma_{\mathsf{AAM}} \rangle \tag{15}$$

where v is the update version and σ_{AAM} is *admins* multi-signature on tuple $\langle \mathcal{A}, v \rangle$. A designated *admin* writes AAM to the *attributes* layer of the multi-layer skipchain, by executing $\mathsf{Append}(\mathsf{L_{at}}, \mathsf{AAM})$. Witnesses append AAM only if σ_{AAM} is valid.

Finally, a designated *admin* sends the pair $\langle \langle \mathcal{A}_{\delta(r)}, v \rangle, \sigma_r \rangle$ to *ABE server* r, $\forall r \in \mathcal{R}$. Each *ABE server* obtains the latest version of *admins* public keys from the *admins* skipchain layer and accepts the pair $\langle \langle \mathcal{A}_{\delta(r)}, v \rangle, \sigma_r \rangle$ only if σ_r is valid.

Policy translation. In this phase, *admins* translate access control policy $\mathbb{P}$ into a semantically equivalent policy $\mathbb{A}$ expressed over the *ABE attributes* computed in the previous *attribute derivation* phase. Without loss of generality, we describe the translation phase by representing access control policies $\mathbb{P}$ and $\mathbb{A}$ as boolean formulas expressed over *original* and *ABE attributes*, respectively. The boolean formula representing $\mathbb{P}$ must be translated so that satisfying a threshold t_r of semantically equivalent *ABE attributes* implies satisfying the corresponding *original attribute*. We recall that $t_r - 1$ is the maximum amount of malicious *ABE servers*. A designated *admin* translates the original access control policy by substituting each *original attribute* p_j with a boolean expression that returns *true* only if at least a threshold t_r of *ABE attributes* that are semantically equivalent to the *original attribute* are *true*. To this aim, the *admin* computes the set $\mathbb{S}_j$ of all possible subsets S_j of $\mathcal{A}^j$ of cardinality equal to t_r, that is:

$$\mathbb{S}_j := \{S_j : S_j \subseteq \mathcal{A}^j, |S_j| = t_r\} \tag{16}$$

where n_r is the total number of *ABE servers* and is greater than t_r. The resulting boolean formula $\mathbb{A}$ is computed by substituting each attribute p_j of the original boolean formula $\mathbb{P}$ as following:

$$p_j \leftarrow \bigvee_{S_j \in \mathbb{S}_j} \left(\bigwedge_{e \in S_j} e \right), \forall p_j \in \mathbb{P} \tag{17}$$

Example. To clarify the *update policies* procedure we propose an example, where we consider a scenario in which a new software update for "premium" users who have paid a subscription fee is about to be published. The *software house* defines the access control policy $\mathbb{P}$ as the following formula:

$$\text{“premium”} \wedge \text{“paid”} \tag{Ex. 1}$$

We assume that *admins* have configured three *ABE servers* ($\mathcal{R} = \{r_1, r_2, r_3\}$), and that they want to ensure 1-out-of-3 survivability, that is, tolerating the compromise of one *ABE server*. *Admins* extract and enumerate attributes of Formula (Ex. 1) obtaining "premium" and "paid" ($P = \{\text{“premium”}, \text{“paid”}\}$). In the following we use the binary operator $\|$ to denote the concatenation of the binary representation of the operands. The attribute matrix $\mathcal{A}$ is:

ABE servers	Attributes	
	premium	paid
r_1	$\text{“}pk_{r1}\|\text{premium”}$	$\text{“}pk_{r1}\|\text{paid”}$
r_2	$\text{“}pk_{r2}\|\text{premium”}$	$\text{“}pk_{r2}\|\text{paid”}$
r_3	$\text{“}pk_{r3}\|\text{premium”}$	$\text{“}pk_{r3}\|\text{paid”}$

TABLE 1
Example attribute matrix $\mathcal{A}$

In Formula Ex. 2 we represent semantically equivalent attributes of Table 1 with the original attribute name and with the row index as subscript. *Admins* can finally translate formula Ex. 1 with the following semantically equivalent formula:

$$\begin{aligned} &(\text{“}pk_{r1}\|\text{premium”} \wedge \text{“}pk_{r2}\|\text{premium”}) \vee \\ &(\text{“}pk_{r2}\|\text{premium”} \wedge \text{“}pk_{r3}\|\text{premium”}) \vee \\ &(\text{“}pk_{r1}\|\text{premium”} \wedge \text{“}pk_{r3}\|\text{premium”}) \\ &\qquad \wedge \\ &(\text{“}pk_{r1}\|\text{paid”} \wedge \text{“}pk_{r2}\|\text{paid”}) \vee \\ &(\text{“}pk_{r2}\|\text{paid”} \wedge \text{“}pk_{r3}\|\text{paid”}) \vee \\ &(\text{“}pk_{r1}\|\text{paid”} \wedge \text{“}pk_{r3}\|\text{paid”}) \end{aligned} \tag{Ex. 2}$$

5.4 Update authentication

We model the update authentication procedure as $\mathsf{AuthenticateUpdate}(\mathsf{src}, \{pk_d\}, v, \mathbb{A}, \Delta t)$, where src is the update source code, $\{pk_d\}$ is the set of *developers* public

keys authorized to authenticate src, v is the update version, $\mathbb{A}$ is the multi-authority policy and Δt is the freshness tolerance value. The goal of this phase is to compute two categories of authenticated update metadata: *Authenticated Update Validation Metadata* (AUVM), intended to be used by *validators* in the update validation phase (Section 5.5), and *Authenticated Binaries Metadata* (ABM), intended to be used by *clients* during update retrieval (Section 5.7 and Section 5.8). This procedure includes two phases operated by *admins* and *developers*, respectively.

Admins bind the update version to the *multi-authority policy* $\mathbb{A}$ and to the *freshness tolerance value* Δt by multi-signing tuples $\langle \mathbb{A}, v \rangle$ and $\langle \Delta t, v \rangle$, producing signatures $\sigma_{\mathbb{A}v}$ and $\sigma_{\Delta tv}$:

$$\sigma_{\mathbb{A}v} \leftarrow \mathsf{MultiSign}(\{sk_a\},\ \langle \mathbb{A}, v \rangle) \tag{18}$$

$$\sigma_{\Delta tv} \leftarrow \mathsf{MultiSign}(\{sk_a\},\ \langle \Delta t, v \rangle) \tag{19}$$

We observe that the two bindings are computed separately because they must be verified in different procedures. *Admins* compute the *developers* KMA (DKMA), which includes the *developers* public keys $\{pk_d\}$ authorized to authenticate the update at version v, as following:

$$\mathsf{DKMA} = \langle \langle \{pk_d\}_{d \in [n_d]}, v, t_d \rangle, t_a, \sigma_{\mathsf{DKMA}} \rangle \tag{20}$$

We note that the DKMA attestation has the update version v in place of the validity period bounds v_b and v_e defined in KMA because the set of keys $\{pk_d\}$ is valid only for version v.

Admins send DKMA and the tuples $\langle \sigma_{\mathbb{A}v}, \mathbb{A}, v \rangle$ and $\langle \sigma_{\Delta tv}, \Delta t, v \rangle$ to each *developer* who verifies the multisignatures $\sigma_{\mathbb{A}v}$ and $\sigma_{\Delta tv}$.

A subset $D \subseteq [n_d]$ such that $|D| \geq t_d$ of developers authorized in DKMA participates in the following operations. Each *developer* in D builds through reproducible builds procedures the update source code src, obtaining the update binaries bin:

$$\mathsf{bin} \leftarrow \mathsf{DeterministicBuild}(\mathsf{src}) \tag{21}$$

Developers agree on a shared deterministic encryption key k by using an authenticated group key agreement [21]:

$$k \leftarrow \mathsf{KeyAgree}(1^\mu, \{pk_d\}_{d \in D}) \tag{22}$$

where 1^μ is the security parameter. Each participating *developer* encrypts the update binaries bin through deterministic encryption with k and produces eb:

$$eb \leftarrow \mathsf{Encrypt}_{\mathsf{DET}}(k, \mathsf{bin}) \tag{23}$$

Then, each participating *developer* uses a secure hash function $\mathsf{H}(\cdot)$ to compute the digests h_k, h_{eb}, h_{src} and h_{bin}:

$$h_k \leftarrow \mathsf{H}(k) \qquad h_{eb} \leftarrow \mathsf{H}(eb) \tag{24}$$

$$h_{\mathsf{src}} \leftarrow \mathsf{H}(\mathsf{src}) \qquad h_{\mathsf{bin}} \leftarrow \mathsf{H}(\mathsf{bin}) \tag{25}$$

Each *developer* in D encrypts the deterministic encryption key k through MA-CP-ABE encryption by using the *multi-authority policy* $\mathbb{A}$ received by *admins*, and computes its digest h_{ek_d} with a secure hash function $\mathsf{H}(\cdot)$:

$$ek_d \leftarrow \mathsf{Encrypt}_{\mathsf{ABE}}(k, \mathbb{A}, \{pk_r\}_{r \in \mathcal{R}})\ \forall d \in D \tag{26}$$

$$h_{ek_d} \leftarrow \mathsf{H}(ek_d)\ \forall d \in D \tag{27}$$

One designated *developer* determines the timestamp t of the current update and computes the *update validation metadata* uvm and *binaries metadata* bm:

$$\mathsf{bm} := \langle h_{\mathsf{bin}}, h_{eb}, h_k, \mathsf{DKMA}, t, \langle v, \Delta t, \sigma_{\Delta tv} \rangle \rangle \tag{28}$$

$$\mathsf{uvm} := \langle h_{\mathsf{src}}, \mathsf{bm} \rangle \tag{29}$$

The developer sends both of them to all other participating *developers*. Each developer verifies their correctness by:

- recomputing h_{src} and h_{bin}, and by verifying that they match the corresponding values in uvm and bm;
- verifying signature $\sigma_{\Delta tv}$;
- verifying that digests h_k and h_{eb} are equal to the digests computed in Equation 24;
- verifying that DKMA is valid, as defined in Section 5.2;
- verifying that t is a timestamp indicating a plausible time of creation of tuples bm and uvm.

Developers in D multi-sign uvm, producing AUVM:

$$\mathsf{AUVM} = \langle \mathsf{uvm}, \sigma_{\mathsf{AUVM}} \rangle \tag{30}$$

Finally, participating *developers* gather the digests h_{ek_d} and multi-sign the tuple $\langle \mathsf{bm}, \{h_{ek_d}\} \rangle$, producing ABM:

$$\mathsf{ABM} = \langle \langle \mathsf{bm}, \{h_{ek_d}\} \rangle, \sigma_{\mathsf{ABM}} \rangle \tag{31}$$

We highlight that the digests h_{src}, h_{bin}, h_{eb} and h_k along with signatures σ_{AUVM} and σ_{ABM} are used to guarantee integrity and authenticity of source code, binaries and related cryptographic material to *validators* and *clients*, respectively.

5.5 Update validation

We model the update validation procedure as $\mathsf{ValidateUpdate}(\mathsf{src}, k, \mathsf{AUVM}, \mathsf{ABM}, \mathsf{location})$, where src is the update source, k is the deterministic encryption key, AUVM and ABM are authentication material, location is the address of encrypted binaries. The goal of this procedure is validate source-to-binary correspondence and append ABM and location to the *update* and *location* layers of the multi-layer skipchain, respectively.

A designated *developer* sends AUVM, ABM and $\langle \mathsf{src}, k \rangle$ to *validators* over a confidential and authenticated channel. Each *validator* obtains the *admins* public keys $\{pk_a\}$ and the latest update version value v' from the *admins* and *update* skipchain layers, and verifies that AUVM, ABM, the tuple $\langle \mathsf{src}, k \rangle$ and version value v' are correct and authentic information by executing Algorithm 1. If all checks pass, *validators* multi-sign ABM producing σ_{VABM}:

$$\mathsf{VABM} := \langle \mathsf{ABM}, \sigma_{\mathsf{VABM}} \rangle \tag{32}$$

Then, a designated *validator* sends VAUM to *admins*.

Admins multi-sign the location of the update at version $\mathsf{ABM}.v$ producing *Authenticated Location* AL which we define as follows:

$$\mathsf{location} := \{loc_{eb}, loc_{ek_1}, \ldots, loc_{ek_{|D|}}\} \tag{33}$$

$$\mathsf{AL} := \langle \langle \mathsf{location}, \langle \mathsf{ABM.bm}.v, v_c \rangle \rangle, \sigma_{\mathsf{AL}} \rangle \tag{34}$$

where loc_{eb} is the location of encrypted updates eb, $loc_{ek_1}, \ldots, loc_{ek_{|D|}}$ are the locations of encrypted keys ek_d and v_c is a unique counter value used to denote the number

Algorithm 1 Metadata validation

function VALIDATE($\{pk_a\}$, AUVM, ABM, src, k, v')
  uvm $\leftarrow$ AUVM.uvm
  bm $\leftarrow$ uvm.bm
  DKMA $\leftarrow$ bm.DMKA
  $\{pk_d\} \leftarrow$ DKMA.$\{pk_d\}$
  Verify($\{pk_d\}$, AUVM.σ_{AUVM}, AUVM)
  Verify($\{pk_d\}$, ABM.σ_{ABM}, $\langle$bm, ABM.$\{h_{ek_d}\}\rangle$)
  Verify($\{pk_a\}$, bm.$\sigma_{\Delta tv}$, $\langle$bm.v, bm.$\Delta t\rangle$)
  Verify($\{pk_a\}$, DKMA.σ_{DKMA}, DKMA)
  $v' \stackrel{?}{<}$ bm.v
  AUVM.$h_{\mathsf{src}} \stackrel{?}{=}$ H(src)
  bin $\leftarrow$ DeterministicBuild(src)
  bm.$h_{\mathsf{bin}} \stackrel{?}{=}$ H(bin)
  bm.$h_k \stackrel{?}{=}$ H(k)
  $eb \leftarrow$ Encrypt$_{\mathsf{DET}}(k$, bin)
  bm.$h_{eb} \stackrel{?}{=}$ H(eb)

of location changes for the same version v which, in this procedure, is initialized to zero. Value v_c is increased in case of updates to the location and controlled by the *witnesses* accordingly. A designated *admin* starts the PBFT-CoSi protocol with *witnesses* by sending VAUM and AL to the *witness* leader. During the *pre-prepare* phase of the PBFT-CoSi protocol, *witnesses* verify multi-signatures σ_{VAUM} and σ_{AL} with the latest *validators* and *admins* public keys specified in the *validators* and *admins* skipchain layers, respectively. If verification succeeds, *witnesses* append VAUM and AL to the *update* and *location* layers, respectively. At the end of PBFT-CoSi *commit* phase, the *witnesses* leader returns to the designated *admin* an attestation of the correct execution of the protocol.

5.6 Client key refresh

We model the client key refresh procedure as ClientKeyRefresh$(cid, P_{cid}, \mathcal{A})$, where cid is the unique identifier of the client, P_{cid} is the set of *original attributes* associated to the client by the *software house* and $\mathcal{A}$ is the latest version of the *ABE attributes matrix* (Section 5.3). The procedure generates a set of *ABE keys* SK_{cid} assigned to the client to decrypt update binaries (Section 5.8).

A *client* first authenticates to the *authentication server* by presenting appropriate credentials that include the client identifier cid. If authentication is successful, the *authentication server* retrieves the *client original attributes* P_{cid} from its own database. By using the *ABE attributes matrix* $\mathcal{A}$, the *authentication server* assigns the set of client ABE attributes $\mathcal{C}$ depending on the *original attributes* P_{cid} associated to the client.

For ease of presentation, we model P_{cid} as a set of indexes to the enumerated set of *original attributes*, as described in the *update policies* procedure (Section 5.3), that is: $P_{cid} \subseteq [|P|]$.

Let us consider a key reliability parameter $f_k \in [0, n_r - t_r - 1]$ that regulates the *clients* tolerance to *ABE servers* key rotations. As an example, a value $f_k = 1$ guarantees that even if one *ABE server* rotates his keys, the *client* is still able to decrypt future updates. The *authentication server* chooses a subset of ABE servers $\bar{\mathcal{R}} \subseteq \mathcal{R}$ for which he releases decryption keys, such that $|\bar{\mathcal{R}}| = (f_k + t_r)$.

The matrix of client *ABE attributes* $\mathcal{C}$ is computed as:

$$\mathcal{C} = \left(\alpha^j_{\delta(r)} : \alpha^j_{\delta(r)} \in \mathcal{A}\right), \forall r \in \bar{\mathcal{R}}, \forall j \in P_{cid} \tag{35}$$

For ease of presentation, we denote as $\mathcal{C}_r$ the row of $\mathcal{C}$ associated to server r. The *authentication server* sends to each *ABE server* $r \in \bar{\mathcal{R}}$ the row $\mathcal{C}_r$ over a secure channel. Each server r computes a set of *ABE keys* $SK_{cid,r}$ as:

$$SK_{cid,r} := \{sk : sk \leftarrow \mathsf{KeyGen}(cid,\ sk_r,\ e_r), \forall e_r \in \mathcal{C}_r\} \tag{36}$$

where we recall that sk_r is server's r secret key. Each *ABE server* $r \in \bar{\mathcal{R}}$ sends the set of keys $SK_{cid,r}$ to the *client* through the *authentication server*. Once all *ABE servers* in $\bar{\mathcal{R}}$ have responded, the *client* can compute the matrix $SK_{cid} = (SK_{cid,r}), \forall r \in \bar{\mathcal{R}}$.

The *authentication server* can adopt multiple strategies to establish the value f_k and the servers of the set $\bar{\mathcal{R}}$. The number of keys the *client* receives depends on the value f_k. The value f_k must lie in the range $[0, n_r - t_r - 1]$ and $|\bar{\mathcal{R}}| = (f_k + t_r)$ because the *client* is able to satisfy policy $\mathbb{A}$ only if *ABE servers* issue at least t_r keys (see Section 5.3). Moreover, $t_{decrypt} \leq n_r$ because $\bar{\mathcal{R}} \subseteq \mathcal{R}$. To one extreme, choosing $f_k = 0$ minimizes the number of keys sent and managed by the *client*, however it forces to refresh his keys when any of the keys belonging to *ABE servers* in $\bar{\mathcal{R}}$ is rotated. On the other extreme, choosing $f_k = n_r - t_r - 1$ maximizes the number of keys sent to the *client*, but the *client* is forced to refresh his keys only when $n_r - t_r - 1$ *ABE servers* keys have been rotated. As long as $f_k < n_r - t_r - 1$, the *authentication server* can choose which *ABE servers* issue the new keys. This may be useful to load balance the key generation process among *ABE servers* in case of bursty key refresh workloads.

5.7 Check for updates

The goal of this procedure, which is started by a *client*, is to efficiently update the *client's* copy of the multi-layer skipchain so that the *client* can determine whether new software updates are available. We model the *check for updates* procedure as CheckForUpdates(τ_t), where τ_t is the last authenticated timestamp belonging to the *time* skipchain layer that is known by the *client* and t denotes the t^{th} execution of *check for updates*. The procedure returns the skipblocks between the current latest skipblock of each layer of the multi-layer skipchain maintained by *witnesses*, and the skipblocks pointed by τ_t. Moreover, it returns the latest authenticated timestamp that the *client* uses in the next invocation of *check for updates*.

The *client* requests the latest timestamp τ'_{t+1} to at least t_w *witnesses* over an authenticated channel. If all timestamps are equal, then the *client* sends $\langle \tau_t, \tau'_{t+1} \rangle$ to any *witness* over an authenticated channel. The *witness* determines the *client's* last skipblock for each skipchain layer by following the hash pointer to the parent skipchain of each layer, starting from the *location* skipblock pointed by τ_t. For each layer, the *witness* determines the shortest chain of multi-signed forward pointers between the *client*'s last skipblock and the current latest skipblock. The *witness* sends to the *client* the

required skipblocks of the *admins* and *witnesses* layers, the multi-signature chains and the latest skipblock of all other layers. The *client* validates the multi-signature chains by using the public keys contained in the *witnesses* skipblocks, validates the *witnesses* skipblocks with the public keys contained in the *admins* skipblocks, and finally validates the timestamp τ'_{t+1} obtained in the beginning by using the latest set of *witnesses* public keys and by checking that the timestamp respects the latest *freshness tolerance* value Δt (see Section 4.2). If skipchain validation succeeds, the *client* sets $\tau_{t+1} = \tau'_{t+1}$ for the following invocation of the same *check for updates* procedure. The *client* checks the latest *attributes* skipblock and executes the *client key refresh* procedure if he needs to refresh his keys (see Section 5.6). If a new *update* skipblock is present, then the *client* executes the *download and decrypt* procedure described in Section 5.8.

Moreover, we observe that the network cost of transferring the multi-signature chains to *clients* is logarithmic in the amount of skipblocks of each layer between τ_t and τ_{t+1} [7].

5.8 Download and decrypt

The goal of the *download and decrypt* procedure is to let the *client* download, authenticate and decrypt a new software update after he determines its availability through the *check for updates* procedure (Section 5.7). We model the procedure as $\mathsf{DownloadAndDecrypt}(\mathsf{VABM}, \mathsf{AL}, SK_{cid})$, where VABM and AL are authenticated data structures obtained from the *check for updates* procedure, and SK_{cid} is the *client* ABE keys obtained from the *client key refresh* procedure.

We represent the procedure in Algorithm 2. In lines 2 through 10 the *client* downloads and decrypts ek_i, which is the deterministic encryption key encrypted by *developer* i with ABE encryption (Equation 26). If the digest of the decrypted key is not equal to the digest included in the VABM data structure, he tries to download and decrypt the deterministic key encrypted by developer $i+1$ for all possible developers. When the *client* finds a valid key, he can start the decryption procedure of the update binaries. To this aim, he downloads the encrypted binaries from the *distribution infrastructure* (line 12), validates their authenticity (line 13) and decrypts them (line 14). We recall that the authenticity of digest h_{eb} is guaranteed by σ_{VABM} included in VABM and validated during the *check for updates* procedure. Finally, the *client* can install the update binaries bin and complete the update procedure.

5.9 Key management

The *key management* operations are accomplished by *admins*, who collectively act as a root certification authority, and thus are the only role responsible for rotating the public keys of other roles. Each key management operation produces a new authenticated set of public keys for a specific role. With the only exception of the *developers* role, the new set is sent to *witnesses*, who verify its authenticity, validate its correctness and append it to the corresponding *keys metadata* skipchain layer. In the following we describe key management operations for the different roles: *admins*, *witnesses*, *validators*, *ABE servers* and *developers*.

Admins. *Admins* manage their keys $\{pk_a\}$, authenticated in attestation AKMA, by self-authenticating a new set of public keys $\{pk_a\}'_{a\in[n_a]}$ in a new attestation AKMA′. *Admins* must specify the new validity period $\langle \mathsf{AKMA}'.v_b, \mathsf{AKMA}'.v_e\rangle$, which must not overlap with the validity period $\langle \mathsf{AKMA}.v_b, \mathsf{AKMA}.v_e\rangle$ (Equation 5). *Admins* may also change the multi-signature threshold by specifying $\mathsf{AKMA}'.t_a \neq \mathsf{AKMA}.t_a$. A threshold $\mathsf{AKMA}.t_a$ of *admins* multi-sign AKMA′ and send it to *witnesses*, that append it only if it satisfies the security constraints (non-overlapping validity periods, threshold and validity of signing keys).

Witnesses, validators. *Admins* manage the public keys of *witnesses* and *validators*, authenticated in a key metadata attestation KMA (specifically, WKMA, VKMA for the two roles, see Section 5.2), by computing a new attestation KMA′. *Admins* must specify the new validity period $\langle \mathsf{KMA}'.v_b, \mathsf{KMA}'.v_e\rangle$ which must not overlap with the validity period $\langle \mathsf{KMA}.v_b, \mathsf{KMA}.v_e\rangle$. A threshold $\mathsf{KMA}.t_a$ of *admins* multi-signs the new KMA′ and send it to *witnesses*, that append it only if it satisfies the due security constraints.

ABE servers. *Admins* manage the public keys of *ABE servers*, authenticated in a key metadata attestation RKMA (see Section 5.2), by computing a new attestation RKMA′ as outlined in Algorithm 3. If *admins* rotate *ABE servers* public keys, *admins* must also execute the *attribute derivation* phase of the *update policies* procedure to update the *authenticated attributes matrix* $\mathcal{A}$ (Section 5.3). If *admins* change the value of t_r to t'_r after removing or adding new *ABE servers*, *admins* must also execute the *policy translation* phase of the *update policies* procedure to compute a new *multi-authority policy* $\mathbb{A}$ that complies to the new t'_r. *Admins* can rotate *ABE servers* keys to promptly recover from the compromise of up to $t_r - 1$ *ABE servers* without executing $\mathsf{Setup}(1^\lambda)$ again, by executing Algorithm 3. In line 2 *admins* update the set of *ABE servers* by excluding the set of compromised *ABE servers* $\mathcal{R}_c \subset \mathcal{R}$ and including the set of new *ABE servers* $\mathcal{R}'$ (where $|\mathcal{R}_c| = |\mathcal{R}'|$). In line 3 each new *ABE server* in $\mathcal{R}'$ generates his signing key pair. In lines 4 and 5 *admins* authenticate the new set of *ABE servers* public keys. In lines 6 through 8 *admins* update and authenticate the new attribute matrix $\mathcal{A}'$ in a new attestation AAM′ and finally in lines 9 and 10 *admins* append the new authenticated data structures AAM′ and RKMA′ to the appropriate skipchain layers.

Developers. *Admins* authenticate *developers* keys with the DKMA attestation at every execution of the *update authentication* procedure (see Equation 20). For this reason, *developers*

Algorithm 2 Download and decrypt

```
1:  function DownloadDecrypt(VABM, AL, SK_cid)
2:      i ← 0
3:      repeat
4:          i ← i + 1
5:          loc_ek_i ← AL.location.loc_ek_i
6:          ek_i ← Download(loc_ek_i)
7:          H(ek_i) ?= VABM.ABM.h_ek_i
8:          k ← Decrypt_ABE(SK_cid, ek_i)
9:          h ← VABM.ABM.bm.h_k
10:     until i ?= |D| || H(k) ?= h
11:     loc_eb ← AL.location.loc_eb
12:     eb ← Download(loc_eb)
13:     H(eb) ?= VABM.ABM.bm.h_eb
14:     bin ← Decrypt_DET(k, eb)
```

Algorithm 3 Rotate ABE servers

function ROTATEABESERVERS($\{sk_a\}$, $\mathcal{R}_c$, $\mathcal{R}'$, v)
  $\mathcal{R} = (\mathcal{R} \setminus \mathcal{R}_c) \cup \mathcal{R}'$
  $\langle sk_r, pk_r \rangle \leftarrow \mathsf{AuthSetup}(r)\ \forall r \in \mathcal{R}'$
  $\sigma_{\mathsf{RKMA}'} \leftarrow \mathsf{MultiSign}(\{sk_a\},\ \langle \{pk_r\}_{r\in\mathcal{R}}, t_r, v'_b, v'_e \rangle)$
  $\mathsf{RKMA}' = \langle \langle \{pk_r\}_{r\in\mathcal{R}}, t_r, v'_b, v'_e \rangle, t_a, \sigma_{\mathsf{RKMA}'} \rangle$
  $\mathcal{A}' \leftarrow \mathsf{UpdatePolicy}(\mathbb{P}, v, t_r, \mathcal{R})$
  $\sigma_{\mathsf{AAM}'} \leftarrow \mathsf{MultiSign}(\{sk_a\},\ \langle \mathcal{A}', v \rangle)$
  $\mathsf{AAM}' = \langle \langle \mathcal{A}', v \rangle, \sigma_{\mathsf{AAM}'} \rangle$
  $\mathsf{Append}(\mathsf{L_r}, \mathsf{RKMA}')$
  $\mathsf{Append}(\mathsf{L_{at}}, \mathsf{AAM}')$

can freely rotate keys between software releases, and no skipchain layer is dedicated to tracking their evolution.

We highlight that key management operations that alter the number of actors within a specific role might impact the security and the performance of the system. Removing an actor might require the remaining actors within the same role to guarantee higher level of availability or to operate higher workloads to comply to the required multi-signatures threshold. At the same time, lowering the threshold would guarantee the same level of performance but decrease the security level of the system. As an example, reducing the number *validators* without also reducing the corresponding multi-signature threshold $\mathsf{VKMA}.t_v$ implies that a higher percentage of *validators* must be available during the *update validation* procedure. However, reducing $\mathsf{VKMA}.t_v$ implies tolerating a lower amount of malicious *validators*. We note that the number of *witnesses* is $n_w = 3(t_w - 1) + 1$ due to the use of PBFT protocol, therefore *witnesses* must not be less than 4.

6 SECURITY ANALYSIS

Survivability is guaranteed by the adoption of multi-signatures coupled with validity thresholds and by the accurate definition of access control policies, as described in Section 5.3. This design choice allows the compromise of up to a threshold of actors (admins, developers, ABE servers, witnesses, verifiers) still guaranteeing that an adversary cannot forge any authenticated cryptographic material produced in the procedures of our proposal. The key rotation procedure (Section 5.9) ensures *recoverability*. In fact, if a threshold of admins, witnesses, verifiers, ABE servers or developers keys are compromised, then admins can recover the system to a safe state by rotating the compromised keys as soon as the incident is detected.

Authenticity of software updates is protected by admins and developers multi-signatures, who digitally sign software updates data and metadata during the update authentication procedure (Section 5.4). The authenticity of developers and admins digital signatures is in turn guaranteed by the multi-layer skipchain that allows admins to manage keys and act as a certification authority.

An adversary can try to break authenticity in several ways: by compromising admins to issue rogue keys, by compromising developers to authenticate malicious source code, by compromising validators to approve malicious update binaries or witnesses to equivocate or fork the multi-layer skipchain. However, *authenticity* does not suffer from single points of failure as the validity of a role's multi-signature is determined by an admin-defined threshold on the number of signers, which is authenticated through attestations published on the multi-layer skipchain (Section 5.2). As a result, an adversary is unable to break authenticity of software updates because we assume it is not able to compromise more than a threshold of actors for each role.

Confidentiality. We evaluate confidentiality guarantees by distinguishing the confidentiality of software updates binaries and of source code. The former is protected against an adversary who intercepts the binaries, either by compromising the distribution infrastructure or by intercepting them while being sent to and downloaded from the distribution infrastructure. The adversary can try to break the confidentiality of intercepted software update binaries by violating ABE servers to recover the decryption key k. As a result, the confidentiality of software update binaries does not suffer from single points of failure because, as described in Section 5.3, the adversary must violate at least t_r ABE servers to be able to obtain the decryption key.

The confidentiality of software update source code could be violated by corrupting a developer or a validator during the validation phase. Concerning this issue, we consider the typical approach of the literature assuming that the confidentiality of software updates source code is based on weakest-link security. Attacks by one developer could be even minimized by adopting software management techniques that segment source code and prevent one developer to access the whole code and/or by detailed logging and forensics mechanisms, but the integration of similar solutions is out of the scope of this paper.

Freshness and timeliness. The multi-layer skipchain ensures *freshness* of software updates. The adoption of PBFT along with non-equivocation mechanisms guarantees to clients a consistent view of the skipchain state and, more specifically, of its latest skipblocks. As described in Section 5.7, non-equivocation is obtained by querying at least t_w witnesses so that at least an honest witness is queried. The honest witness guarantees to detect equivocation attacks if he returns a response which is inconsistent with the responses of other possible malicious witnesses. Moreover, the *time* skipchain layer allows clients to detect freeze attacks by an adversary who controls the communication channel of the client and presents a stale view of the skipchain.

The *timely* and scalable distribution of software updates is made possible by the design of the software update encryption mechanism that produces a single ABE ciphertext for all clients, thus allowing us to leverage existing distribution infrastructures, such as Content Delivery Networks.

Transparency is guaranteed by validators through the update validation procedure (Section 5.5). Assuming that validators use a trusted compiler, they can detect attacks against source-to-binary correspondence in which an adversary induces developers into signing backdoored software update binaries that do not correspond to the original update source code.

Operation	Role	Procedure	Costs
ABE encryption	Developer	(1)	$p_T + x\,(H + 3\,p_1 + p_2 + 2\,p_T)$
ABE decryption	Client	(6)	$x\,(3\,e + H + p_T)$
ABE keygen	ABE server	(4)	$a(2\,H + p_1 + 3\,p_2)$

(1) Auth. update, (4) Key refresh, (6) Download & decrypt

TABLE 2
Computational costs

Data	Role	Proc.	Type	Costs
ABE ciphertext	Developer	(1)	$\odot$	$G_T + x\,(2\,G_1 + G_2 + G_T)$
	Client	(6)	$\odot$	
	Dist.Inf	(3)	$\odot\otimes$	$\lvert D\rvert\,(G_T + x\,(2\,G_1 + G_2 + G_T))$
ABE client keys	ABE server	(4)	$\odot$	$a\,(G_1 + G_2)$
	Client	(4)	$\odot\otimes$	$a\,(t_r + f_k)\,(G_1 + G_2)$

(1) Auth. update, (3) Publish, (4) Key refresh, (6) Download & decrypt
$\odot$: network costs, $\otimes$: storage costs

TABLE 3
Storage and network costs

7 Costs analysis and performance evaluation

We evaluate the overhead introduced by the proposed framework and demonstrate that it achieves practical performance. First, we analyze the computational, network and storage costs of our contributions: MA-CP-ABE extension and distributed update authentication protocol. Then, we evaluate the framework performance by considering timings of actual state-of-the-art cryptographic libraries at multiple security levels and scenarios of increasing complexity.

7.1 Costs analysis

We analyze the computational costs of MA-CP-ABE in terms of relevant primitive cryptographic operations and complexity of access control policy formulas by referring to Table 2. We denote the pairing operation as e, exponentiation in $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ as p_1, p_2 and p_T respectively, and hash to $\mathbb{G}_2$ operation as H. Moreover, we denote as x the number of rows of the linear secret sharing scheme (LSSS) matrix used by the ABE scheme, that is equal to the number of logic gates in the translated access control policy $\mathbb{A}$ plus one [22]. As we discussed in Section 5.3, the number of logical gates in $\mathbb{A}$ can be computed by knowing the number of gates in the original access control policy $\mathbb{P}$, that we denote as γ, the number of ABE servers n_r, and the ABE server threshold t_r. The value of x can be computed as following:

$$x = \gamma + (\gamma + 1) \cdot [t_r \cdot \binom{n_r}{t_r} - 1] + 1 \tag{37}$$

We observe that the encryption and decryption costs are linear in x, which increases as a binomial function of n_r and t_r. We recall that the value n_r represents the number of heterogeneous ABE servers that do not share common-mode failures [23]. Thus, it is unlikely that n_r exceeds a few units. We also observe that decryption is typically more expensive than encryption due to the three pairing operations $(3 \cdot e)$ and to the possibility of using optimizations for fixed bases in point scalar multiplication operations in the encryption operation [24] (only the point scalar multiplication p_2 is computed on a variable base). Finally, the key generation phase depends on the amount of *original attributes* granted to the client, that we denote as a (see Section 5.6). This represent the worst case of a client requesting keys for all attributes, such as at setup time. In other procedures, such as policy updates (see Section 5.3), the client may request keys for a subset of attributes. If some load balancing strategy is applied, then the procedure in each key generation would involve just a subset of the ABE servers (see Section 5.6).

We discuss network and storage overhead introduced by ABE cryptographic material by referring to Table 3. The first column (*data*) describes the type of cryptographic material. The second and third columns (*role* and *procedure*) describe which actors are affected by these costs and in which procedures of the framework, respectively. The fourth column (*type*) indicates whether the material affects storage or network costs for each actor. The last column includes the costs with regard to the type of material, where we denote the size of an element belonging to the groups $\mathbb{G}_1$, $\mathbb{G}_2$ and $\mathbb{G}_T$ as G_1, G_2 and G_T, the number of developers that participate in the update authentication procedure as $|D|$, the key reliability parameter as f_k, and the number of original attributes granted to a client as a. The size of the ABE ciphertext grows linearly as a function of x, thus increasing as a binomial function of n_r and t_r for ABE encryption and decryption costs. Moreover, the size of the keys received and maintained by each client depends on the values a, t_r and f_k because each of $(t_r + f_k)$ ABE servers send a keys of size $(G_1 + G_2)$ (see Section 5.6).

As described in Section 5.4, our distributed update authentication protocol includes the costs due our MA-CP-ABE extension, and costs due to distributed key agreement and deterministic encryption. We observe that any deterministic symmetric encryption scheme adds only a minor computational overhead with respect to probabilistic symmetric encryption schemes [25], that in turn are negligible with regard to asymmetric encryption schemes. Moreover, they do not add relevant network overhead. Thus, any role that is able to operate MA-CP-ABE is also able to support deterministic encryption. We analyze the costs of the distributed key agreement by considering an instantiation based on authenticated group key agreement protocol [21]. Each developer executes $2|D|+1$ group exponentiations and $3|D|$ signatures, where $t_d \leq |D| \leq n_d$ is the number of developers that participate in the key agreement protocol. Moreover, the network costs of each developer consist of $3|D|$ group elements and $3|D|$ signatures. If we consider realistic values of $|D|$ being within the tens of developers, the amount of group elements and digital signatures that a developer must compute and transmit introduce feasible computational and network costs.

7.2 Performance evaluation

The performance evaluation of our MA-CP-ABE extension is based on two pairing-friendly curves: BN256 [26] and BN462 [27], which account for security levels of about 100 and 128 bits. In Table 4 we report timings expressed in clock cycles and group element sizes expressed in bits. We obtained timings by using the implementations included in the MCL library v1.10 [28] compiled for Intel i7-8665U processor. Moreover, we analytically computed element sizes by using the curves parameters. For BN256 $G_1 = 256$ and

Curve	λ	Timings [Clock cycles]					Size [bit]		
		p_1	p_2	p_T	e	H	G_1	G_2	G_T
BN256	100	97 k	210 k	332 k	638 k	131 k	256	3063	3063
BN462	128	719 k	1.6 M	1.9 M	4.8 M	788 k	462	5535	5535

TABLE 4
Curves parameters and performance

n_r	t_r	γ	x	BN256			BN462		
				Enc	Dec	Size	Enc	Dec	Size
2	2	1	4	1.2 ms	2.0 ms	3.7 kB	7.3 ms	14 ms	6.7 kB
2	2	2	6	1.7 ms	3.0 ms	5.4 kB	11 ms	21 ms	9.7 kB
2	2	5	12	3.3 ms	5.9 ms	10.4 kB	21 ms	43 ms	18.7 kB
2	2	10	22	6.0 ms	11 ms	18.7 kB	38 ms	78 ms	33.7 kB
2	2	50	102	28 ms	51 ms	85.2 kB	177 ms	363 ms	153.8 kB
3	2	1	12	3.3 ms	5.9 ms	10.4 kB	21 ms	43 ms	18.7 kB
3	2	2	18	4.9 ms	8.9 ms	15.4 kB	32 ms	64 ms	27.7 kB
3	2	3	24	6.6 ms	12 ms	20.4 kB	42 ms	85 ms	36.7 kB
3	2	10	66	18 ms	33 ms	55.3 kB	115 ms	235 ms	99.8 kB
3	2	50	306	83 ms	152 ms	255.0 kB	530 ms	1.1 s	460.2 kB
4	2	10	132	36 ms	65 ms	110.2 kB	229 ms	470 ms	198.9 kB
5	3	3	120	33 ms	59 ms	100.2 kB	208 ms	427 ms	180.9 kB

TABLE 5
Evaluation of encryption and decryption times, and of the ciphertext size

$G_2 = G_T = k \cdot G_1 = 3072$ bits, and for BN462 $G_1 = 462$ and $G_2 = G_T = k \cdot G_1 = 5544$ bits, where G_1 is computed by the designers of the curve with regard to the security level and $k = 12$ is the embedding degree of both curves. The sizes are computed by considering compressed elliptic curve coordinates and uncompressed finite field elements. In such a way, the size of G_1 and G_2 is equal to the size of an element of the field over which the curve is defined.

To estimate the performance of the approach in realistic scenarios, in Tables 5 and 6 we propose results based on a set of parameters that are representative for real-world scenarios. In both tables, we compute timings from the clock cycles reported in Table 4 and formulas reported in Tables 2 and 3, and in Equation (37) by considering a modern x86_64 CPU operating at $4.8GHz$. The considered parameters influence the system performance: n_r, t_r, γ, a and f_k. In Table 5 we show encryption and decryption times as well as ciphertext size which depend on parameters n_r, t_r and γ. Moreover, in Table 6 we report key generation times and the sizes of decryption keys for ABE servers and clients, which depend on parameters a, t_r and f_k.

Results in Table 5 highlight that decryption, run by clients, is the most expensive operation and typically costs twice the encryption, run by developers. Our proposal is practical in realistic scenarios where the number of ABE servers n_r and the amount of tolerable malicious servers $t_r - 1$ is of few units. For example, when $n_r = 3$, $t_r = 2$ and γ is within tens of logical gates, timings are acceptable (if $\gamma \in [1, 10]$ then decryption takes between 60ms to 200ms). A possible instance of the original access policy with $\gamma = 3$ is $\mathbb{P} = \mathsf{A} \wedge (\mathsf{D} \vee (\mathsf{B} \wedge \mathsf{C}))$.

Table 5 also reports the space overhead of a single ABE ciphertext in columns "Size", which corresponds to the network cost for clients during the download and decrypt procedure, and for developers during the authenticate update procedure (see Table 3). As highlighted in Table 3, this value must be multiplied by $|D|$ so to compute the network and storage costs of the distribution infrastructure bears during the publish procedure. In realistic scenarios where the number of ABE servers n_r and the amount of tolerable malicious ABE servers t_r are within a few units, and the number of logical gates γ of the original access policy is within tens of gates, the overhead does not exceed 200KB. When software updates are in the order of hundreds of kilobytes, the space overhead of one ABE ciphertext has a size comparable to that of the update. This overhead becomes negligible in scenarios where software updates tend to be in the order of ten megabytes or more.

a	$(t_r + f_k)$	BN256			BN462		
		Keygen	Server	Client	Keygen	Server	Client
2	2	413 us	6.7 kB	13.3 kB	3.0 ms	12.0 kB	24.0 kB
	3			20.0 kB			36.0 kB
	6			39.9 kB			72.1 kB
3	3	619 us	10.0 kB	30.0 kB	4.5 ms	18.0 kB	54.1 kB
30	4	6.2 ms	99.8 kB	399.4 kB	45 ms	180.2 kB	720.7 kB

TABLE 6
Evaluation of key generation timings and key sizes

Our performance evaluation shows that the timings for generating ABE keys are acceptable even in cases where a client satisfies several tens of attributes, as evidenced in Table 6. Curve BN256 allows to compute a single ABE key in 206.5 microseconds, allowing to generate 4842 ABE keys per second. Curve BN462 allows to compute a single ABE key in 1.5 milliseconds, allowing to generate 666 ABE keys per second. We note that these throughput values consider the maximum achievable throughput of a single machine with a single core executing the key generation procedure. Higher throughput values can be obtained by horizontally and vertically scaling ABE servers or by choosing a key reliability value f_k such that $0 \leq f_k < n_r - t_r - 1$, so that the authentication server can apply load balancing strategies to share the load between ABE servers, as we discussed in Section 5.6. The network costs for ABE servers and the network and storage costs for clients in scenarios where a client satisfies a few attributes, tend to be several tens of kilobytes. In extreme scenarios where a client satisfies several tens of attributes of an unrealistically complex access policy, the space overhead of ABE keys is several hundreds of kilobytes.

Our analysis shows that the framework is practical for a number of ABE servers in the order of several units, and a number of developers and logical gates in access control policies in the order of tens. In particular, the proposed framework is better suited for scenarios in which software updates tend to be in the order of megabytes. As a requirement to support the framework, clients must have enough memory and storage capacity to maintain ABE decryption keys and decrypt ciphertexts, that are typically of tens of kilobytes each.

8 Conclusions

We propose an original framework that allows the secure and survivable distribution of confidential software updates. This framework is based on multi-authority attribute-based encryption, and extends its key generation procedure with an original technique to guarantee survivability. It is based on a distributed infrastructure with no single points of failure which is able to guarantee availability and security even in the presence of partial compromises. We demonstrate the practicality of the proposal through a performance

evaluation of our original key generation technique and of the encryption scheme in the context of secure software updates. The results show that the proposed framework can achieve practical performance at 128-bit security level on modern computers in realistic settings. This framework paves the way to the design of secure and robust business-oriented architectures for the distribution of confidential software updates. Our proposal highlights even some interesting open problems, such as the protection of source code confidentiality with no weakest link security assumption, and the possibility of enabling software transparency without relying on third parties. These issues may be addressed in future work that may well be integrated into the proposed framework.

REFERENCES

[1] Kaspersky Lab, "Operation ShadowHammer: new supply chain attack threatens hundreds of thousands of users worldwide," https://www.kaspersky.com/about/press-releases/2019_operation-shadowhammer-new-supply-chain-attack, 2019, Accessed Jun. 2020.

[2] Microsoft Defender ATP Research Team, "Windows Defender ATP thwarts Operation WilySupply software supply chain cyberattack," https://www.microsoft.com/security/blog/2017/05/04/windows-defender-atp-thwarts-operation-wilysupply-software-supply-chain-cyberattack/, 2017, Accessed Jun. 2020.

[3] NIST, "Software Supply Chain Attacks," https://csrc.nist.gov/CSRC/media/Projects/Supply-Chain-Risk-Management/documents/ssca/2017-winter/NCSC_Placemat.pdf, 2017, Accessed Jun. 2020.

[4] A. Bellissimo, J. Burgess, and K. Fu, "Secure Software Updates: Disappointments and New Challenges," in *HotSec*, 2006.

[5] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "A look in the mirror: Attacks on package managers," in *Proc. 15th ACM Conf. Computer and Communications Security*, 2008.

[6] J. Samuel, N. Mathewson, J. Cappos, and R. Dingledine, "Survivable Key Compromise in Software Update Systems," in *Proc. 17th ACM Conf. Computer and Communications Security*, 2010.

[7] K. Nikitin, E. Kokoris-Kogias, P. Jovanovic, N. Gailly, L. Gasser, I. Khoffi, J. Cappos, and B. Ford, "CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds," in *Proc. 26th USENIX Security Symp.*, 2017.

[8] M. Al-Bassam and S. Meiklejohn, "Contour: A Practical System for Binary Transparency," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, J. Garcia-Alfaro, J. Herrera-Joancomartí, G. Livraga, and R. Rios, Eds. Springer, 2018, vol. 11025.

[9] J. Li, P. L. Reiher, and G. J. Popek, "Resilient Self-Organizing Overlay Networks for Security Update Delivery," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 1, 2004.

[10] H. Johansen, D. Johansen, and R. van Renesse, "Firepatch: Secure and Time-Critical Dissemination of Software Patches," in *Proc. IFIP Int'l Information Security Conf.*, 2007.

[11] M. Ambrosin, C. Busold, M. Conti, A.-R. Sadeghi, and M. Schunter, "Updaticator: Updating Billions of Devices by an Efficient, Scalable and Secure Software Update Distribution Over Untrusted Cache-enabled Networks," in *Computer Security - ESORICS*, 2014.

[12] K. Trishank, B. Akan, A. Sebastien, M. Damon, B. Russ, M. Cameron, L. Sam, W. André, and C. Justin, "Uptane: Securing Software Updates for Automobiles," in *The 14th Escar Europe*, 2016.

[13] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos, "Diplomat: Using Delegations to Protect Community Repositories," in *Proc. 13th USENIX Symp. Networked Systems Design and Implementation*, 2016.

[14] T. K. Kuppusamy, V. Diaz, and J. Cappos, "Mercury: Bandwidth-Effective Prevention of Rollback Attacks Against Community Repositories," in *Proc. USENIX Annual Tech. Conf.*, 2017.

[15] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic Patch-Based Exploit Generation is Possible: Techniques and Implications," in *Proc. IEEE Symp. Security and Privacy*, 2008.

[16] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-Policy Attribute-based Encryption," in *Proc. IEEE Symp. Security and Privacy*, 2007.

[17] Y. Rouselakis and B. Waters, "Efficient Statically-Secure Large-Universe Multi-Authority Attribute-Based Encryption," in *Proc. Int'l Conf. Financial Cryptography and Data Security*, 2015.

[18] A. Beimel, "Secure Schemes for Secret Sharing and Key Distribution," PhD thesis, Technion-Israel Institute of Technology, 1996.

[19] A. Boldyreva, "Threshold Signatures, Multisignatures and Blind Signatures Based on the Gap-Diffie-Hellman-Group Signature Scheme," in *Int'l Work. Public Key Cryptography*, 2003.

[20] D. Boneh, B. Lynn, and H. Shacham, "Short Signatures from the Weil Pairing," in *Proc. Int'l Conf. Theory and Application of Cryptology and Information Security*, 2001.

[21] E. Bresson and D. Catalano, "Constant Round Authenticated Group Key Agreement via Distributed Computation," in *Proc. Int'l Work. Public Key Cryptography*, 2004.

[22] A. Lewko and B. Waters, "Decentralizing Attribute-Based Encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2011.

[23] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, 2004.

[24] N. Pippenger, "On the evaluation of powers and related problems," in *17th Annual Symp. Foundations of Computer Science*. IEEE, 1976.

[25] S. Gueron and Y. Lindell, "GCM-SIV: Full Nonce Misuse-Resistant Authenticated Encryption at Under One Cycle per Byte," in *Proc. 22nd ACM SIGSAC Conf. Computer and Communications Security*, 2015.

[26] M. Naehrig, R. Niederhagen, and P. Schwabe, "New software speed records for cryptographic pairings," in *Int'l Conf. Cryptology and Information Security in Latin America*. Springer, 2010.

[27] Y. Sakemi, T. Kobayashi, T. Saito, and R. S. Wahby, "Pairing-Friendly Curves," Internet Engineering Task Force, Internet-Draft draft-irtf-cfrg-pairing-friendly-curves-05, Jun. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-pairing-friendly-curves-05

[28] S. Mitsunari, "mcl: a portable and fast pairing-based cryptography library," https://github.com/herumi/mcl.

Federico Magnanini received the master's degree in computer engineering from the University of Modena and Reggio Emilia, Italy in 2019. He is working toward the PhD degree at the International Doctorate School in information and communication technologies (ICT) of the University of Modena and Reggio Emilia, Italy. His research interests include information security and distributed ledger technologies.

Luca Ferretti is assistant professor in computer science and engineering at the University of Modena and Reggio Emilia. He received the PhD degree at the International Doctorate School in information and communication technologies (ICT) of the same university. His research interests include information security and applied cryptography.

Michele Colajanni is full professor of computer engineering at the University of Bologna. He received the Master degree in computer science from the University of Pisa, and the Ph.D. degree in computer engineering from the University of Roma. He was assistant professor at the University of Roma, and full professor at the University of Modena since 2000. His research interests include cybersecurity, performance and prediction models, big data on cloud systems.