V'CER: Efficient Certificate Validation in Constrained Networks

David Koisser and Patrick Jauernig, *Technical University Darmstadt;* Gene Tsudik, *University of California, Irvine;* Ahmad-Reza Sadeghi, *Technical University Darmstadt*

https://www.usenix.org/conference/usenixsecurity22/presentation/koisser

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

V'CER: Efficient Certificate Validation in Constrained Networks

David Koisser
Technical University Darmstadt
david.koisser@trust.tu-darmstadt.de

Patrick Jauernig
Technical University Darmstadt
patrick.jauernig@trust.tu-darmstadt.de

Gene Tsudik
University of California, Irvine
gene.tsudik@uci.edu

Ahmad-Reza Sadeghi
Technical University Darmstadt
ahmad.sadeghi@trust.tu-darmstadt.de

Abstract

We address the challenging problem of efficient trust establishment in *constrained networks*, i.e., networks that are composed of a large and dynamic set of (possibly heterogeneous) devices with limited bandwidth, connectivity, storage, and computational capabilities. Constrained networks are an integral part of many emerging application domains, from IoT meshes to satellite networks. A particularly difficult challenge is how to enforce timely revocation of compromised or faulty devices. Unfortunately, current solutions and techniques cannot cope with idiosyncrasies of constrained networks, since they mandate frequent real-time communication with centralized entities, storage and maintenance of large amounts of revocation information, and incur considerable bandwidth overhead.

To address the shortcomings of existing solutions, we design V'CER, a secure and efficient scheme for certificate validation that augments and benefits a PKI for constrained networks. V'CER utilizes unique features of Sparse Merkle Trees (SMTs) to perform lightweight revocation checks, while enabling collaborative operations among devices to keep them up-to-date when connectivity to external authorities is limited. V'CER can complement any PKI scheme to increase its flexibility and applicability, while ensuring fast dissemination of validation information independent of the network routing or topology. V'CER requires under 3KB storage per node covering 10^6 certificates. We developed and deployed a prototype of V'CER on an in-orbit satellite and our large-scale simulations demonstrate that V'CER decreases the number of requests for updates from external authorities by over 93%, when nodes are intermittently connected.

1 Introduction

Spurred by new and emerging applications—ranging from IoT to satellite networks—there has been a growing trend of interconnecting large numbers of heterogeneous resource-constrained devices in recent years. In such settings, both devices and networking are constrained: devices have anemic computation and storage abilities, while networking is characterized by limited bandwidth, low transmission range, dynamic topology, and more critically, by intermittent connectivity due to mobility and/or device hibernation. We use the term *constrained networks* to describe such settings.

In particular, satellite networks constitute an emerging class of constrained networks [27]. Due to decreased satellite costs (e.g., $22,000 for a *CubeSat*, including launch [26]) and their increased accessibility (e.g., AWS Ground Station service [2]), the number of operational satellites has doubled to over 4,000 since 2019 [49]. Moreover, the trend towards constellations, i.e., deployment of a network of small satellites (instead of few large ones), will dramatically increase this growth in the years to come. SpaceX's Starlink alone plans to deploy around 42,000 satellites [48].

However, small satellites have many constraints, such as strict power budgets, radiation-resistant hardware components with limited computing and storage capabilities [3], and physical transmission limitations, e.g., due to periodic line-of-sight blockage by planets and other celestial bodies. While satellite networks might seem to be an extreme example of constrained networks, similar problems arise in terrestrial settings. For example, a number of mesh protocols have been designed to handle poor network conditions for low-power Internet of Things (IoT) devices [7, 25, 55, 56]. Also, similar to line-of-sight disruptions in satellite communication, home/office automation devices often hibernate to conserve power. Unattended outdoor IoT devices that use natural sources of power (solar, wind, etc.) tend to hibernate. Moreover, mobile terrestrial devices can go out of range or encounter communication obstacles. All these conditions result in intermittent or unstable connectivity.

Efficient and secure trust establishment in constrained networks is essential. While small, static, homogeneous networks could rely on symmetric cryptography, large heterogeneous networks require scalable asymmetric cryptography. Public Key Infrastructure (PKI) and public-key certificates are common tools deployed for establishing mutual trust between

devices. However, timely certificate revocation of malfunctioning or compromised devices is critical for retaining trust in the whole network. Since satellites also suffer from software bugs [13] and can be subject to attacks [47], timely revocation is very important. In fact, the Internet Engineering Task Force (IETF) already recognized the difficulty of revocation in a protocol slated for satellite networks [20].

There is also a large body of literature on PKI in distributed settings, such as observer-based approaches [5, 30, 33, 43], schemes that enable end-users to distributively check their certificates [37, 53], blockchain-based approaches [1, 10], and PKI that is specifically geared towards networks with delay tolerance [15, 17, 41] as well as mobile ad-hoc networks [12, 36, 51, 52]. Furthermore, recent efforts focus on PKI for IoT [24, 45, 50].

As discussed in Section 9, current techniques have some important shortcomings with regard to revocation checks in constrained networks: First, they make strong assumptions about network connectivity or incur heavy communication overhead for the entire network. For instance, on-demand revocation checking, such as Online Certificate Status Protocol (OCSP) [44], requires a reliable connection and separate request for every certificate validation. However, a reliable connection to a central entity cannot be guaranteed in constrained networks. Second, storage and distribution of explicit revocation information, e.g., using Certificate Revocation Lists (CRLs) [8], consumes high bandwidth and storage, including regular updates. However, devices in a constrained networks can be highly limited in terms of storage and networking abilities, e.g., the popular Z-Wave low-power IoT technology has a bandwidth of 100Kbps at best [54]. Even recent results that significantly reduce storage and update overheads of CRLs [32,46] are specifically designed for (generally reliable) Web-based revocation. Thus, they are poorly suited for an environment where devices frequently miss revocation updates. We discuss this in more detail in Section 8.

In summary, efficient certificate validation in constrained networks is still a challenging open problem that we aim to tackle in this paper.

Goals & Contributions: We present V'CER, a novel certificate validation scheme for constrained networks. V'CER provides lightweight certificate validation directly between devices, with minimal communication overhead, by defining operations that allow nodes to epidemically keep each other's revocation information up-to-date. In a network with 10^6 active certificates, V'CER requires under 3KB of storage per device to allow all devices to mutually authenticate each other, using widely available cryptographic primitives. Furthermore, if devices miss revocation updates, V'CER reduces (by over 93%) the number of devices that need to request fresh validation information from the CA.

Our main contributions include:

- V'CER enables flexible and lightweight revocation checks in PKI schemes, especially for device-to-device trust establishment, thus enabling PKI in constrained networks.
- V'CER defines novel algorithms that utilize the deterministic structure of Sparse Merkle Trees (SMTs), which allows devices to keep each other up-to-date. This eliminates the need for the vast majority of devices to contact the CA when updates were missed.
- V'CER introduces the *Validation Forest* (VF) data structure for efficient exchange of validation information among devices, whenever they come in contact. VF allows for epidemic dissemination of validation information, without the need to consider application-specific network aspects, such as the underlying topology or routing protocols.
- V'CER involves no on-demand requests, while requiring very little storage overhead for devices. At the same time, it offers better security guarantees for constrained networks than prior approaches.
- We evaluate V'CER's proof-of-concept implementation on the European Space Agency's OPS-SAT satellite. We then thoroughly evaluate V'CER in a large-scale simulation modeling a constrained network. We also open-source our prototype to the research community[1].

2 Background

This section provides background on the data structure that is central for V'CER operation.

2.1 Sparse Merkle Trees

First proposed by Merkle [38], an Merkle Hash Tree (MHT) is an accumulator that efficiently represents a set of data elements and allows to construct Proofs of Inclusion for individual elements [38]. It requires a secure hash function, the digest of which is used to label each element contained in the set. Knowing only the root hash of the tree, anyone can verify whether a given element (leaf) is part of the set by using a small set of hashes (co-path) corresponding to all sibling nodes on the path to root, i.e., a Proof of Inclusion (PoI), which is $O(\log n)$ where n is the number of leaves.

An Sparse Merkle Tree (SMT) [34] is a type of MHT which contains all possible hash values, e.g., an SMT for SHA256 has 2^{256} leaves and depth of 255. When inserting a new element into an SMT, the leaf representing a placeholder is replaced by the new element. Because of the deterministic position of elements in the tree, removing elements is easy. When an element is removed, the root hash changes and a previously valid PoI for the removed element becomes invalid.

[1] https://github.com/vcer4pki/VCER

While a complete SMT for a reasonable hash function (e.g., SHA256) is not computable, in practice, most leaves need not be assigned. Specifically, all "empty" leaves can be assigned $\mathtt{H}(\varnothing)$. Going up such a tree, all hashes above any two empty leaves amount to: $\mathtt{H}(\mathtt{H}(\varnothing)\|\mathtt{H}(\varnothing))$, and so on. Therefore, using SHA256 as an example, we can construct all 256 empty branch hashes for all depths in the tree stored in an *EmptyHashesList*, which covers all hashes in the empty parts of the tree. Since most parts of this SMT are empty, we only need to consider assigned leaves to compute the root.

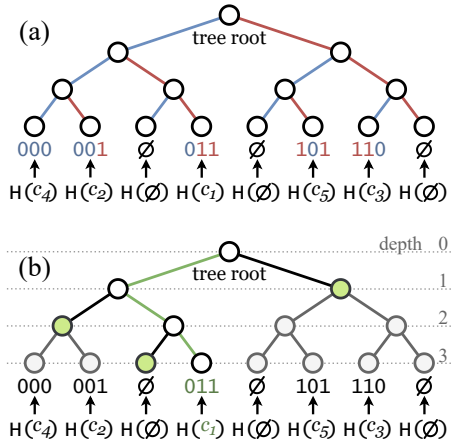

Figure 1: SMT examples: (a) depicts how the position of hashes is determined and (b) shows a PoI and how depth numbering works.

Figure 1 (a) shows a sample SMT with the five elements $c_1, ..., c_5$ for a hash function with a 3-bit digest. Branch colors show how the positioning of the elements works: blue represents an unset bit going left along the branch, and red is a set bit going right. For example, leaf c_5 has the hash digest $\mathtt{H}(c_5) = 101$, which represents its position in the tree. The three hash digests 010, 100 and 111 are not assigned in this example. Figure 1 (b) shows the path for leaf c_1 as green lines in the tree. To construct its PoI, we need its sibling, and all siblings of its ancestors, corresponding to the nodes in green. A PoI for c_1 is verified as follows:

(i) The verifier knows the (signed) root and the PoI: $[\mathtt{H}(\varnothing), \mathtt{H}(\mathtt{H}(c_4)\|\mathtt{H}(c_2)), \mathtt{H}(\mathtt{H}(\mathtt{H}(\varnothing)\|\mathtt{H}(c_5))\|\mathtt{H}(\mathtt{H}(c_3)\|\mathtt{H}(\varnothing)))]$

(ii) The verifier computes the hash of c_1, combines it with the first element in the PoI, and hashes both. Next, the result is combined with the second element in the PoI and hashed. Finally, the last hash is combined with the third element in the PoI and hashed.

(iii) The verifier compares the last hash from (ii) with the root value. If they match, the PoI is valid.

However, note that, since the sibling leaf of c_1 is empty, we can omit it. In a more complex example using SHA256, a PoI would need 256 elements, though most of them will be empty, which can be omitted with the help of the *EmptyHashesList*. This requires an additional *bitmap* the same size as the tree height, e.g., 256 bits for SHA256. The bits in the bitmap at every tree level indicate whether an empty hash or an element of the PoI should be used (see Algorithm 6 in Appendix A for details). Because the digest of a strong cryptographic hash function is a pseudo-random value, the hashes of elements are uniformly distributed. Therefore, assigned leaves are evenly spread over the SMT, implying a PoI size of $\log n$ hashes, *on average*.

3 System Model

We consider three types of entities: $\mathcal{CA}$, *Nodes*, and cacher *Nodes*. *Nodes* are devices that need to validate each other's public key certificates. We use the term "validation" to focus on the revocation check, i.e., the chain-of-trust verification of certificates is implied. Although a *Node* can compute hashes and verify signatures, it has limited processing power and storage. A subset of *Nodes* play the role of cacher *Nodes* by storing additional information. Communication is constrained due to low bandwidth, mobility, and intermittent connectivity that can result in frequent network partitions. *Nodes* exchange contact messages whenever they *meet*. $\mathcal{CA}$ is the certificate-issuing authority. Communication with the $\mathcal{CA}$ is particularly restricted. We treat $\mathcal{CA}$ as a single entity, albeit it might be distributed in practice.[2] Each *Node* knows $\mathcal{CA}$'s certificate and trusts $\mathcal{CA}$'s signatures. The number of certificates in the system is denoted by n. For simplicity, we assume that each *Node* has exactly one certificate. Further, we assume a coarse time synchronization among *Nodes* and $\mathcal{CA}$ in the range of hours, e.g., to check certificate expiration.

3.1 Adversary Model

We consider the Dolev-Yao model adversary (*Adv*) that can eavesdrop on, intercept, or inject any number of messages [16]. However, *Adv* is naturally bound to physical constraints of the network, such as not being able to reach a disconnected *Node*. In particular, we assume that *Adv* acts locally, and cannot block all communications in the network at the same time. We assume the Sparse Merkle Tree construction to be secure, i.e., collision-resistant.

3.2 Requirements

As mentioned earlier, designing an efficient distributed certificate validation scheme for constrained networks is challenging, especially since nodes can miss updates, due to connectivity issues. We believe that an ideal scheme must satisfy the following requirements:

R.1: Handle arbitrary delays: Responses from a central entity ($\mathcal{CA}$) might be delayed or lost.

[2] Section 7 discusses the setting with multiple $\mathcal{CA}$s.

R.2: Avoid single points of failure: Although centralized systems are easy to set up, they fail when the central entity loses connectivity. Thus, we must avoid relying solely on central entities.
R.3: Consistency: While there is always current local state, validation decisions must be derived from a common trust anchor to ensure consistency.
R.4: Handle constrained devices: A certificate validation scheme must have minimal impact on nodes' scarce resources.
R.5: Timeliness: Since freshest validation information is crucial for security, an ideal scheme must ensure regular updates and their fastest dissemination.

4 V'CER Overview

V'CER consists of three main components:

(i) Certificate validation via an individual PoI for each *Node*.
(ii) Efficient spreading of fresh validation information using the *Validation Forest* ($\mathcal{VF}$) data structure. $\mathcal{VF}$ is the trust anchor used to validate any PoI.
(iii) Distributed repair whereby up-to-date *Nodes* directly help outdated *Nodes* to recover from missed updates.

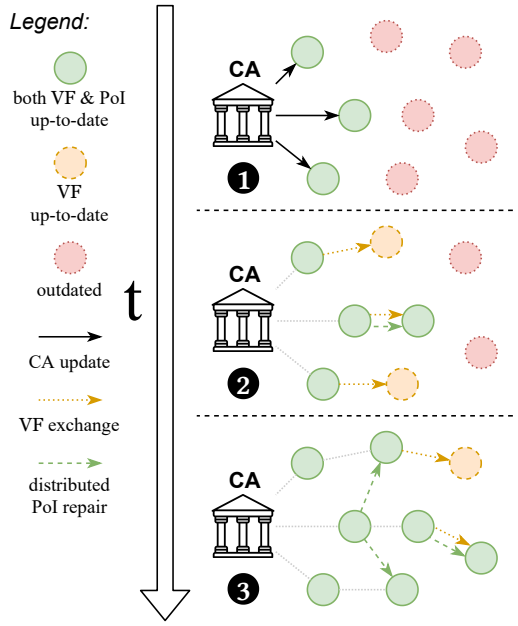

Figure 2: Example of $\mathcal{CA}$ updates being spread in a distributed manner.

Figure 2 is a high-level overview of V'CER operation when there are $\mathcal{CA}$ updates, e.g., some certificates are newly revoked. After $\mathcal{CA}$ updates the validation information aggregated in $\mathcal{VF}$, all *Nodes* become outdated. In step ❶, $\mathcal{CA}$ spreads its update information, with which each *Node* can update both its $\mathcal{VF}$ and its PoI. However, in a constrained network, an update would not reach all *Nodes*. Specifically, the update reaches only the solid green *Nodes* in step ❶, while all the rest miss this update (dotted red *Nodes*), e.g., by not being connected to the network or by simply hibernating.

After some time, in step ❷, some outdated *Nodes* meet up-to-date *Nodes* and start to exchange information. Up-to-date *Nodes* update outdated nodes' $\mathcal{VF}$ (dashed yellow arrows). With a fresh $\mathcal{VF}$, a *Node* can correctly validate other *Nodes*' certificates, e.g., reject newly revoked ones. This illustrates one key feature of V'CER: any *Node* encountering an up-to-date node obtains the latest $\mathcal{VF}$ on contact. Thus, fresh validation information spreads very quickly and epidemically.

An outdated *Node*'s own PoI can become outdated if it misses some updates. Although such a *Node* can still validate certificates of up-to-date nodes correctly, it cannot provide a proof for its own certificate validity to other *Nodes* (dashed yellow). However, nodes with up-to-date PoIs can help nodes with outdated ones to update their PoI via distributed repair (green dashed arrows). For this, we exploit the deterministic structure of SMTs to design operations for distributed repair, as shown in Section 5.4. In step ❷ this succeeds for the node in the center. However, after some time passes in step ❸, nodes continue to encounter others, increasing their chance for distributed repair to succeed. Eventually, most nodes would successfully repair their outdated PoI, with only a few having the need to directly request a fresh one from the $\mathcal{CA}$. We demonstrate this in Section 6.3.2.

5 V'CER Certificate Validation Scheme

This section describes all components of V'CER. For certificate validation, *Nodes* use Proofs of Inclusion to verify that a *Node*'s certificate is valid. In a simplified example, the $\mathcal{CA}$ can build an SMT with all the active certificates' hashes and *Nodes* only need to know the root hash of the SMT to verify any PoI. Each *Node* then stores the PoI for its own certificate, and becomes capable of proving validity of its certificate to others. This is done by computing the PoI's root hash (see Algorithm 6 in Appendix A) and if it matches the root hash given by $\mathcal{CA}$, the certificate is valid. Thus, *Nodes* only need to store the respective tree root and their own PoI.

Figure 3 shows individual operations in V'CER. In step ❶, the $\mathcal{CA}$ constructs the Validation Forest $\mathcal{VF}$, as described in Section 5.1. Instead of using a single SMT, $\mathcal{VF}$ is a data structure used by *Nodes* to efficiently keep their certificate validation information up-to-date. Upon any changes in $\mathcal{VF}$, e.g., revocations, the $\mathcal{CA}$ constructs updates that are processed by the *Nodes* in step ❷, which is described in Section 5.2. As some *Nodes* may miss these updates, in step ❸, *Nodes* exchange information to keep each other updated and identify which parts of $\mathcal{VF}$ are outdated (see Section 5.3). Finally in step ❹, *Nodes* repair each other's PoIs when $\mathcal{CA}$ updates are missed, as presented in Section 5.4. We accompany our description with a running example that uses practical parameters. The evaluation in Section 6.3 is also based on this

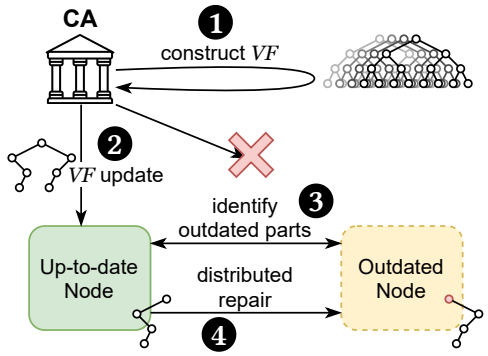

Figure 3: The individual operations of V'CER.

example. In terms of cryptographic primitives, the example uses SHA256 and ECDSA signatures based on the *secp256r1* curve.

5.1 Validation Forest

The Validation Forest $\mathcal{VF}$ is the core data structure for validating certificates in V'CER. As long as a $\mathcal{N}ode$'s $\mathcal{VF}$ is up-to-date, it can correctly validate another $\mathcal{N}ode$'s certificate with its PoI. $\mathcal{VF}$ has three main parts. (i) A number of SMT roots for PoI validation. (ii) Aggregator $\mathcal{A}ggr$ is a small data structure used for efficient exchange between $\mathcal{N}odes$ to keep their $\mathcal{VF}$ up-to-date. (iii) $\mathcal{CA}$'s signature of $\mathcal{A}ggr$ ($\mathcal{A}_s$), which is the trust anchor for all operations in V'CER. add phrasing that VF represents active cert set

The number of trees in $\mathcal{VF}$ is the number of epochs it models. Certificates have a defined maximum lifetime and this lifetime is split into e epochs. A certificate's expiration date is then used to assign it to the respective epoch tree $\mathcal{E}_\mathcal{T}$, represented by the epoch root $\mathcal{E}_r$ stored in $\mathcal{VF}$. Each node stores the $\mathcal{VF}$ including all $\mathcal{E}_r$ and keeps it up-to-date. A $\mathcal{N}ode$'s own proof is valid for a single $\mathcal{E}_r$, which can be inferred from the certificate's expiration date. This stabilizes the individual SMTs, significantly reducing the number of potentially outdated PoIs on any updates by $\mathcal{CA}$. After an epoch has passed, a *forest prune* occurs, i.e., the oldest tree is pruned, as it contains only expired certificates. This means that $\mathcal{CA}$ and $\mathcal{N}odes$ can discard the oldest $\mathcal{E}_\mathcal{T}$ and the corresponding $\mathcal{E}_r$.

For our accompanying example, we define certificates with a maximum lifetime of around one year, split into weeks as epochs. Thus, e =52, meaning that the $\mathcal{VF}$ stores a maximum of 52 $\mathcal{E}_r$, each representing certificates for the corresponding week of expiration. This requires each node to store up to 52 hashes for the tree roots, resulting in ~1.7kB of storage. Assuming that changes occur mostly in the newest epoch, e.g., when new certificates are issued, other epoch trees are left untouched, including their $\mathcal{E}_r$. Thus, all current PoIs depend on them. This amount of overhead fulfills requirement R.4 for low-storage devices. Other certificate lifetime configurations are possible; however, the primary overhead factor is the revocation update frequency, as we show in Section 6.3.2.

5.1.1 Aggregator

The Aggregator $\mathcal{A}ggr$ is used to exchange key information by $\mathcal{N}odes$ about the current $\mathcal{VF}$ state to help keep it up-to-date among each other. It is designed to be lightweight to allow its inclusion in contact messages $\mathcal{N}odes$ exchange when they meet (cf. Section 5.3). $\mathcal{A}ggr$ includes Aggregator root $\mathcal{A}_r$, timestamp $\mathcal{A}_t$, and checksums $\mathcal{A}_c$ to identify outdated tree roots. Furthermore, $\mathcal{A}ggr$ is signed by $\mathcal{CA}$. Aggregator signature $\mathcal{A}_s$ serves as the trust anchor for $\mathcal{N}odes$ to confirm outcomes of all operations (requirement R.3).

$\mathcal{A}_r$ is computed by simply concatenating all tree roots in $\mathcal{VF}$ and hashing resulting information. Thus, any change in any tree would result in a different $\mathcal{A}_r$, and $\mathcal{N}odes$ can use it to check if any parts of their $\mathcal{VF}$ is outdated. To distinguish among multiple $\mathcal{A}ggr$ in terms of freshness, each contains a timestamp $\mathcal{A}_t$.

$\mathcal{A}ggr$ also contains the checksums $\mathcal{A}_c$ for all $\mathcal{E}_r$ in $\mathcal{VF}$ to efficiently identify outdated $\mathcal{E}_r$. This avoids having a $\mathcal{N}ode$ send all tree roots to its peer with an outdated $\mathcal{VF}$ every time. $\mathcal{A}_c$ is split into two types: main and aggregated. Each checksum type has a configurable size. Main checksums are applied directly to tree roots representing newest epochs in $\mathcal{VF}$. Aggregated checksums are applied on a number of concatenated tree roots that come after the ones covered by main checksums. The number that is aggregated into a single checksum is configurable in V'CER. Note, while we use the term "checksum", it is sufficient to simply use some bytes of the hash, as the SMT protects against collisions. This way, tree roots that are expected to change more often than others are covered by their own checksum, e.g., newly issued certificates are inserted into the newest tree. In contrast, $\mathcal{E}_\mathcal{T}$ that are expected to be more stable get aggregated checksums.

In our example, we use 2 main checksums and aggregate 10 tree roots per aggregated checksum, resulting in 5 aggregated checksums for a total of 7. For $\mathcal{A}_t$, we use a 4 Bytes-long UNIX-timestamp. Using 2 Bytes per checksum, this results in $\mathcal{A}ggr$ = 50 Bytes ($\mathcal{A}_r$ = 32B, $\mathcal{A}_c$ = 14B, $\mathcal{A}_t$ = 4B). Also, $\mathcal{CA}$'s signature $\mathcal{A}_s$ over $\mathcal{A}ggr$ is 64 Bytes. This amount of overhead fulfills requirement R.4 for low-bandwidth devices.

5.2 CA Updates

When any changes occur, $\mathcal{CA}$ updates the respective $\mathcal{E}_\mathcal{T}$ resulting in changes for both $\mathcal{VF}$ and PoIs. Thus, $\mathcal{CA}$ needs to distribute updates for $\mathcal{N}odes$ to be up-to-date. To keep the $\mathcal{N}odes$' $\mathcal{VF}$ updated, $\mathcal{CA}$ simply needs to distribute new $\mathcal{A}ggr$, including new $\mathcal{A}_s$ and $\mathcal{E}_r$ for affected epochs. Also, if there are no updates for a while, $\mathcal{CA}$ can regularly send out current $\mathcal{A}ggr$ with new $\mathcal{A}_t$. This way, $\mathcal{N}odes$ eventually realize that they are outdated, e.g., even when disconnected for a while.

While this keeps $\mathcal{Nodes}$' validation information updated, any PoI found in epochs affected by an update becomes invalid, and respective $\mathcal{Nodes}$ can no longer provide a valid proof for their certificates. Instead of $\mathcal{CA}$ individually distributing updated PoIs to all affected $\mathcal{Nodes}$, it constructs universal updates for all $\mathcal{Nodes}$. This is done by distributing PoIs for all updated certificates, including revoked ones. Due to the deterministic order of elements in SMT, $\mathcal{Nodes}$ can process the update PoIs affecting their own epoch to update their PoI. Afterwards, the update PoIs are discarded. Furthermore, when an update contains many PoIs, there are likely many redundant PoI elements that can be aggregated to reduce update size.

Algorithm 1 `update_poi_with_poi` function for updating a proof of inclusion regarding an up-to-date proof of inclusion. $int\langle p\rangle$ accesses the p-th bit of int from the right, $|\mathbb{H}|$ is the bit length of the hash digest and $|l|$ is the number of elements in list l. The variables ending in *path* are lists, *hash* and *bitmap* variables are integers that fit $|\mathbb{H}|$.

Input: *my_leaf_hash*, *my_path*, *my_path_bitmap*, *new_leaf_hash*, *new_path*, *new_path_bitmap*
Output: *my_path*, *my_path_bitmap*

1: $xor\_leaves \leftarrow my\_leaf\_hash \oplus new\_leaf\_hash$
2: $target\_pos \leftarrow$ position of left-most set bit in *xor_leaves* from the left
3: $is\_update \leftarrow$ `False`
4: $path\_pos \leftarrow 0$
5: **for** $i \leftarrow 0$ **to** $(target\_pos + 1)$ **do**
6: **if** $my\_path\_bitmap\langle|\mathbb{H}| - 1 - i\rangle = \text{True}$ **then**
7: **if** $i = target\_pos$ **then**
8: $path\_pos \leftarrow path\_pos + 1$
9: $is\_update \leftarrow \text{True}$
10: **else**
11: $path\_pos \leftarrow path\_pos + 1$
12: $my\_path[|my\_path| - path\_pos] \leftarrow new\_path[|new\_path| - path\_pos]$
▷ loop finds update-bit-position and if it affects existing hash in path
13: $update\_hash \leftarrow$ `calc_path_root` $(new\_leaf\_hash, new\_path, new\_path\_bitmap, (target\_pos + 1))$
14: **if** *is_update* **then**
15: $my\_path[|my\_path| - path\_pos] \leftarrow update\_hash$
▷ replace exiting hash in path
16: **else**
17: my_path`.insert(`$(|my\_path| - path\_pos), update\_hash$`)`
18: $my\_path\_bitmap\langle|\mathbb{H}| - 1 - target\_pos\rangle \leftarrow \text{True}$
▷ insert new hash in path & set respective bit in bitmap
19: **return** *my_path*, *my_path_bitmap*

Algorithm 1 shows how $\mathcal{Nodes}$ process update PoIs provided by $\mathcal{CA}$. The operation takes an up-to-date PoI and updates an outdated PoI found in the same epoch. This is then done for all PoIs in the update. The path-bitmaps work as described in Section 2.1. First, the hash of the outdated certificate is XOR-ed with the updated hash and the position of the left-most set bit shows where both PoIs split in the tree. Afterwards, the algorithm checks if the outdated PoI already has an element at the split position, triggering an overwrite of this existing element in the PoI. Otherwise, the outdated PoI needs an additional element at the split position, including a set bit in the bitmap. The hash is computed by calling `calc_path_root` (see Algorithm 6 in Appendix A), the same algorithm used to validate PoIs, except the optional last parameter indicates the need to stop at the specified depth, instead of the root hash.

Before the split position is reached, every PoI element is updated along the way. This effectively allows to blindly apply updates, meaning a $\mathcal{Node}$ does not need to worry about the order of applying updates. As long as the PoIs are up-to-date, Algorithm 1 does not perform any destructive updates. Afterwards, the executing $\mathcal{Node}$ can simply check if the resulting PoI is valid. Under the right circumstances, this may even cover for previously missed updates. We take advantage of this aspect for the distributed repair, described in Section 5.4.1.

Epoch Change. Whenever a new epoch starts, there might be many newly issued certificates. On one hand, the aforementioned forest prune occurs (see Section 5.1), i.e., many certificates will expire. At this point, many $\mathcal{Nodes}$ will get a new certificate. On the other hand, new $\mathcal{Nodes}$ may join the network, likewise with new certificates. In V'CER, the issuing of new certificates should be aggregated until an epoch change occurs to increase efficiency. This way, instead of constructing an update bundling many new PoIs, $\mathcal{CA}$ can distribute all the certificates hashes as the update, i.e., all epoch tree leaves. This reduces the update size for the epoch change and all $\mathcal{Nodes}$ that have been issued a new certificate can construct their PoI themselves. Further, this update only needs to be sent to $\mathcal{Nodes}$ that are affected by the epoch change. The rest of $\mathcal{Nodes}$ only need the new epoch root for their $\mathcal{VF}$. Additionally, in case an epoch does not contain any revoked certificates, the respective $\mathcal{E}_r$ can be set to $\mathtt{H}(\varnothing)$, indicating to the network that any certificates in this epoch are not revoked.

5.3 Aggregator Exchange

When a $\mathcal{Node}$ missed any updates from $\mathcal{CA}$, its $\mathcal{VF}$ will be outdated, and thus it will not have fresh certificate validation information. A key aspect of V'CER is for $\mathcal{Nodes}$ to be able to efficiently keep each other up-to-date. For this, $\mathcal{Nodes}$ exchange their Aggregator $\mathcal{Aggr}$ (cf. Section 5.1.1) with the contact message when they meet. Additionally, $\mathcal{Nodes}$ will add information about which epoch they belong to and if they have any caches ready. Both are important for the distributed repair, explained in Section 5.4, and in our accompanying example 1 Byte is sufficient (6 bits for the epoch and 2 bits as flags for caches).

$\mathcal{Nodes}$ only need an up-to-date $\mathcal{VF}$ to be able to correctly validate PoIs for any certificate. This ensures that the current

validation information spreads as quickly as possible throughout the network, without the need to consider any transmission aspects, such as acknowledgments from each $\mathcal{N}ode$ to $\mathcal{CA}$ to all $\mathcal{N}odes$ are up-to-date. For example, if a certificate was revoked and $\mathcal{CA}$ sends out an update, even a $\mathcal{N}ode$ missing this update will meet an up-to-date node eventually and after the $\mathcal{A}ggr$ exchange, inherently know about the revocation, i.e., reject the revoked certificate's PoI. Thus, this meets the requirement R.2 and R.5 for validating certificates.

Figure 4 ❶ depicts such an exchange in detail. After both exchanged their $\mathcal{A}ggr$, the outdated $\mathcal{N}ode$ on the right will see a new Aggregator root $\mathcal{A}_r$ as well as a newer timestamp $\mathcal{A}_t$, and realize its $\mathcal{VF}$ is outdated. To identify which trees are actually affected by the change, the outdated $\mathcal{N}ode$ will check the parities $\mathcal{A}_c$ and send the up-to-date $\mathcal{N}ode$ the respective outdated checksum identifiers. The up-to-date $\mathcal{N}ode$ will answer with the respective $\mathcal{E}_r$ and add $\mathcal{A}_s$. The outdated $\mathcal{N}ode$ is then able to update all of the tree roots in its $\mathcal{VF}$, re-calculate its $\mathcal{A}ggr$, and finally check if $\mathcal{A}_s$ is valid. If one of the actually changed $\mathcal{E}_r$ is found in the same epoch as the $\mathcal{N}ode$'s own certificate, it can imply that its PoI is outdated. There is also the case that $\mathcal{E}_r$ has changed; yet, the corresponding $\mathcal{A}_c$ resulted in the same as before. In this unlikely case, the outdated $\mathcal{N}ode$ has to request all $\mathcal{E}_r$, i.e., $\mathcal{VF}$.

In our example, let us assume there has been a change in the second newest epoch (covered by a main checksum) and 26th epoch in the middle. Thus, the outdated $\mathcal{N}ode$ will request the second and fifth checksum. The up-to-date $\mathcal{N}ode$ will respond with the 11 respective $\mathcal{E}_r$, which allows the outdated $\mathcal{N}ode$ to update its $\mathcal{VF}$. This entire exchange will require around 470 Bytes of communication overhead ($\mathcal{A}ggr + \mathcal{A}_s + 11 \cdot \mathcal{E}_r$) directly between the two $\mathcal{N}odes$.

5.4 Distributed Repair

While the $\mathcal{A}ggr$ exchange ensures that $\mathcal{N}odes$ update their certificate validation information as fast as possible, missed updates may also lead to a $\mathcal{N}ode$'s own PoI to be outdated. However, a key goal of V'CER is to avoid having the outdated $\mathcal{N}ode$ to contact $\mathcal{CA}$ and request a fresh PoI. Otherwise, many $\mathcal{N}odes$ individually requesting fresh PoIs at a similar time leads to a significant overhead for the entire network. Thus, in this section we will present ways how an up-to-date $\mathcal{N}ode$ can directly help an outdated $\mathcal{N}ode$ to repair its PoI. After an outdated $\mathcal{N}ode$ updated its $\mathcal{A}ggr$, it will realize its PoI is not valid and can start requesting repair information from up-to-date $\mathcal{N}odes$ it meets. In the following, we will present two different approaches for this. One is directly leveraging up-to-date PoIs from other $\mathcal{N}odes$. The other approach introduces a cache, stored and maintained by a share of $\mathcal{N}odes$ in the network for increased efficacy of distributed repairs. With these operations, V'CER meets requirement R.2 for a $\mathcal{N}ode$'s own validation proof.

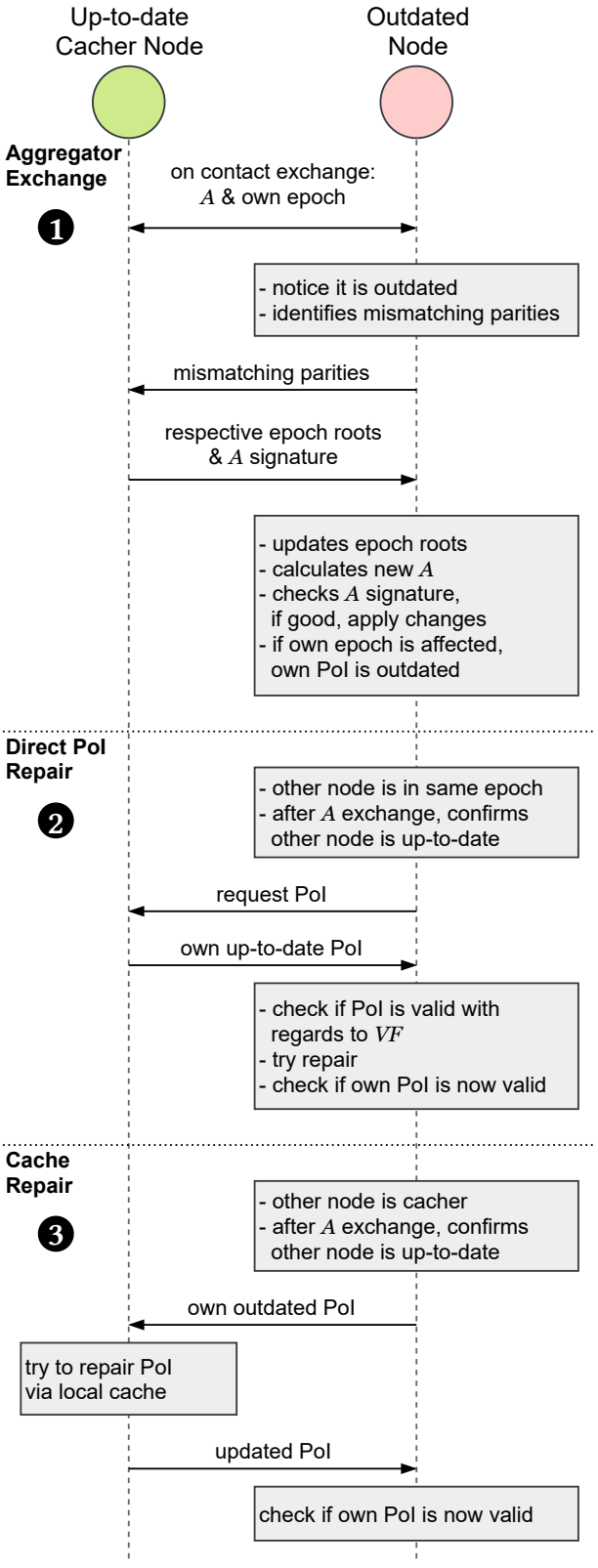

Figure 4: Exchange of $\mathcal{N}odes$ for distributed repair for the three main operations.

5.4.1 Direct PoI Repair

With the direct PoI repair strategy, an outdated $\mathcal{N}ode$ can collect up-to-date PoIs from other $\mathcal{N}odes$ it meets to potentially repair its outdated PoI. Figure 4 ❷ depicts how such an exchange proceeds. After the initial $\mathcal{A}ggr$ exchange on contact, the outdated $\mathcal{N}ode$ knows if the other is up-to-date and if its certificate is found in the same epoch. If both are true, the outdated $\mathcal{N}ode$ will request the other's PoI, check if it is actually valid regarding the current $\mathcal{E}_r$, and if so, use it for repair.

The outdated $\mathcal{N}ode$ leverages Algorithm 1 for this, as mentioned in Section 5.2. The operation replaces all applicable elements in the outdated PoI with the elements in the up-to-date PoI. If the position of the up-to-date $\mathcal{N}ode$'s leaf is favorable, it will update one or more elements in the outdated PoI. In an unfavorable case, elements will simply remain unchanged. This process can be repeated with other up-to-date PoIs in the same epoch, until the $\mathcal{N}ode$'s own PoI is valid regarding the current $\mathcal{E}_r$. This gets more difficult the more updates a $\mathcal{N}ode$ missed for its epoch, which we evaluate in Section 6.2. As Algorithm 1 works by blindly replacing elements in the PoI, the outdated $\mathcal{N}ode$ must first verify the PoI used for the repair is actually valid for the current $\mathcal{VF}$.

5.4.2 Level-Cache Repair

For the Level-Cache (*LC*) strategy, a share of $\mathcal{N}odes$ with larger storage capacity, called *cacher*, may additionally keep all hashes of each $\mathcal{E}_\mathcal{I}$ on a specified depth, i.e., the cache level (*clvl*). This results in a storage overhead of 2^{clvl} hashes per epoch. As mentioned in Section 2.1, the leaves of an $\mathcal{E}_\mathcal{I}$ are uniformly distributed. Due to this fact, the higher depth levels of the SMT are likely to be assigned before the lower ones. If enough elements were inserted, the first few depth levels of a SMT will form a fully filled sub-tree. Further, most updates in its PoI from the perspective of one $\mathcal{N}ode$ will likely be in this sub-tree.

To clarify this phenomenon, consider the example of a SMT with one leaf that is only zeros. Any new leaf that is inserted will go a different path at one point in the tree, which creates the need for an additional element in the PoIs for both leaves. The chances that any new leaf will branch off to the second half of the tree, i.e., the leaf having a set bit on the most-left position, are 50%. The likelihood of branching off on the second depth level is 25%, and halves for any further depth level. Thus, when inserting many leaves, the first depth levels in the SMT will likely branch off first. This implicitly means, that given an up-to-date *LC*, outdated $\mathcal{N}odes$ are likely able to repair their own PoI with it. The exact probabilities of this are discussed in Section 6.2. Furthermore, $\mathcal{N}odes$ may use *LC*s with different *clvl*s, e.g, more resourceful $\mathcal{N}odes$ may keep a larger cache and more limited $\mathcal{N}odes$ a smaller one, if any at all. To avoid having to send the entire *LC* to an outdated $\mathcal{N}ode$, it will instead send its outdated PoI to the cacher $\mathcal{N}ode$, as shown in Figure 4 ❸.

In our example, we chose to equip a share of $\mathcal{N}odes$ with a *LC* for *clvl* = 7. This means that these cacher $\mathcal{N}odes$ will store an additional 4KB per epoch, and thus 208KB in total. In Section 7, we will discuss possible alternative strategies; yet, for our purposes we set all cachers to store a *LC* with the same *clvl* for all epochs.

Algorithm 2 `update_lvl_cache_with_poi` for updating the level-cache regarding a new proof of inclusion

Input: *LC*, *clvl*, *new_leaf_hash*, *new_path*, *new_path_bitmap*
Output: *LC*

1: $delete\_bits \leftarrow 2^{(|\mathbb{H}| - clvl)} - 1$
2: $part\_no \leftarrow new\_leaf\_hash \;\&\; \sim delete\_bits$
3: $part\_no \leftarrow part\_no \gg (|\mathbb{H}| - clvl)$
▷ take *clvl* left-most bits that define position in *LC*
4: *update_hash* ← `calc_path_root` (*new_leaf_hash*, *new_path*, *new_path_bitmap*, *clvl*)
▷ calculate path's root, but stop at depth *clvl*
5: $LC[part\_no] \leftarrow update\_hash$
6: **return** *LC*

However, the cacher needs to keep its *LC* up-to-date as well. The construction of the *LC* is described in Appendix A.2.4. The cacher uses Algorithm 2 to process $\mathcal{CA}$ update PoIs for one epoch. First, it extracts the correct position in *LC* to be updated by the new PoI. Afterwards, it uses the PoI root hash calculation, with the only difference that it stops at *clvl* and inserts the resulting hash at the respective position of *LC*. When the cacher itself misses $\mathcal{CA}$ updates, it may also meet other cacher $\mathcal{N}odes$ and request up-to-date *LC*s from them.

Algorithm 3 `update_poi_with_lvl_cache` for updating a proof of inclusion with a given level-cache

Input: *my_leaf_hash*, *my_path*, *LC*, *clvl*
Output: *my_path*

1: $delete\_bitmap \leftarrow 2^{(|\mathbb{H}| - clvl)} - 1$
2: $part\_no \leftarrow my\_leaf\_hash \;\&\; \sim delete\_bitmap$
3: $part\_no \leftarrow part\_no \gg (|\mathbb{H}| - clvl)$
4: $part\_neg \leftarrow \sim part\_no$
▷ negated bits of *part_no* define its hash-neighbor in *LC*
5: **for** $i \leftarrow 0$ **to** *clvl* **do**
6: *new_hash* ← `calc_pos_in_LC` (*part_neg*, $(i+1)$, *LC*)
▷ constructs a hash at position *part_neg* at depth $(i+1)$ with the *LC*
7: $my\_path[|my\_path| - 1 - i] \leftarrow new\_hash$
8: $part\_neg\langle clvl - 1 - i\rangle \leftarrow \sim part\_neg\langle clvl - 1 - i\rangle$
▷ flip bit to get hash-neighbor for next depth
9: **return** *my_path*

To update a PoI with a *LC*, Algorithm 3 is used. Like in Algorithm 2, the first three lines construct the position of the targeted cache element in *LC*. Then, for all *clvl*, the respective neighborhood hash of the outdated PoI is constructed with the help of `calc_pos_in_LC`. This function simply takes a position and depth, which are used to construct a hash further up the tree in *LC*. The resulting hash is then set in the correct position in the outdated PoI. This process is repeated for all

depth levels in *LC*, as the lower depth PoI elements can be updated as well. This operation blindly replaces PoI elements and may not entirely repair the leaf's PoI. However, if it cannot entirely repair the PoI it will still repair elements that *LC* provides and at least assists in the overall repair process of the outdated $\mathcal{N}ode$. In Section 6.2 we show that *LC* can repair outdated PoIs with high probability, given appropriately chosen parameters.

6 Evaluation

In this section, we evaluate V'CER regarding multiple aspects. First, we evaluate its security, followed by an analysis on the success probabilities of the distributed repair approaches we introduced. Finally, we will consider V'CER's performance regarding run-time overhead and large-scale networks.

6.1 Security

As V'CER provides the means to validate certificates, the adversary $\mathcal{A}dv$ aims to convince $\mathcal{N}odes$ that (i) a revoked or forged certificate is valid, or (ii) a valid certificate is revoked. In the following, we will explore different strategies $\mathcal{A}dv$ may use to achieve this goal and explain how V'CER prevents their success.

Manipulating Updates. To achieve either (i) or (ii), $\mathcal{A}dv$ can try to counterfeit updates by disseminating a new $\mathcal{E}_r$ that reflects the false state of the targeted certificate, i.e., a valid PoI. However, a false $\mathcal{E}_r$ in $\mathcal{VF}$ would also result in a different $\mathcal{A}_r$. As $\mathcal{A}ggr$ is signed by $\mathcal{CA}$ via $\mathcal{A}_s$, $\mathcal{N}odes$ with an up-to-date $\mathcal{A}ggr$ will discard the counterfeit update as invalid.

Blocking Updates. $\mathcal{A}dv$ can isolate a $\mathcal{N}ode$ preventing $\mathcal{CA}$ updates to reach it or delay update messages such that they receive them after the validation. This way, a recently compromised certificate would still be validated regarding the outdated $\mathcal{VF}$, making $\mathcal{A}dv$ achieve (i). This requires $\mathcal{A}dv$ to block any contact of the outdated $\mathcal{N}ode$ with up-to-date $\mathcal{N}odes$, to prevent them to update their $\mathcal{VF}$. This gets increasingly more difficult with increased network connectivity and more opportunities to meet nodes. Nevertheless, as $\mathcal{CA}$ regularly sends out an updated $\mathcal{A}ggr$, even isolated $\mathcal{N}odes$ will eventually consider their current $\mathcal{VF}$ to be outdated, and thus reject any PoIs. This limits the vulnerability window for $\mathcal{A}dv$, similar to Certificate Revocation Lists (CRLs) or certificates with a limited lifetime, as considered in related works (cf. Section 9). However, we consider completely isolating $\mathcal{N}odes$ as non-trivial, and thus V'CER provides better security than schemes relying on a more centralized distribution of updates, as $\mathcal{A}dv$ needs to prevent a $\mathcal{N}ode$ to communicate with more entities.

Denial-of-Service. Instead of preventing updates from reaching an individual $\mathcal{N}ode$, a powerful $\mathcal{A}dv$ could also perform a DoS attack on the entire network to prevent it from receiving updates to achieve (i). Expecting a regular update of $\mathcal{A}ggr$ by $\mathcal{CA}$, all $\mathcal{N}odes$ will eventually consider their $\mathcal{VF}$ to be stale. Again, the vulnerability window is similar to other revocation checks.

Destructive Repair. To achieve (ii), $\mathcal{A}dv$ can target a $\mathcal{N}ode$ with an outdated PoI and send it false repair information to prevent it from obtaining a valid proof for its certificate after an applicable $\mathcal{CA}$ update. As described in Section 5.4 an outdated $\mathcal{N}ode$ ensures the correctness of repair information regarding its up-to-date $\mathcal{VF}$ before applying the repair. This way, any invalid repair information is detected. Further, meeting any benign $\mathcal{N}odes$ may result in a repaired PoI anyway and for $\mathcal{A}dv$ to prevent this, requires it to isolate the $\mathcal{N}ode$. Finally, a $\mathcal{N}ode$ with an outdated PoI will give up on repairs eventually and request its valid PoI directly from $\mathcal{CA}$.

6.2 Distributed Repair Analysis

In this section, we evaluate the effectiveness of distributed repair approaches. Note that the following results are for a single SMT; yet, $\mathcal{VF}$ will consist of a tree per epoch. For example, assuming individual PoI updates are evenly distributed among all epochs, the number of total missed updates needs to be divided by the number of epochs. For this, we ran simulations on a pre-generated SMT with 100,000 random leaves and averaged the results over 10,000 runs.

Direct PoI Repair Analysis We now present simulation results regarding the repair of an outdated PoI by using random up-to-date PoIs, as described in Section 5.4.1. From the perspective of an outdated $\mathcal{N}ode$, the key driver for success is the distance of the missed updated leaves from the $\mathcal{N}ode$'s own leaf in the SMT. The closer any missed updated leaf is, the fewer PoIs overall can help with the repair. The more updates are missed, the higher the probability that one of them is unfavorable for an outdated $\mathcal{N}ode$.

Figure 5 shows the simulation results for this strategy with an increasingly higher number of missed updates. *first* depicts the percentage of runs which failed to repair the outdated PoI on the first try and *first 10* for the first ten tries. After trying 100 PoIs, the run gave up and consider the attempt failed. Thus, *fails* shows the percentage of runs that stopped after 100 tries. Finally, *avg. try* sketches the average number of PoIs it took to successfully repair the outdated PoI.

Level-Cache Repair Analysis From a theoretical perspective, each *LC* divides each $\mathcal{E}_T$ into 2^{clvl} parts. An outdated $\mathcal{N}ode$ should be able to successfully use an up-to-date *LC* to repair its PoI, if no updates happened in the same part as its

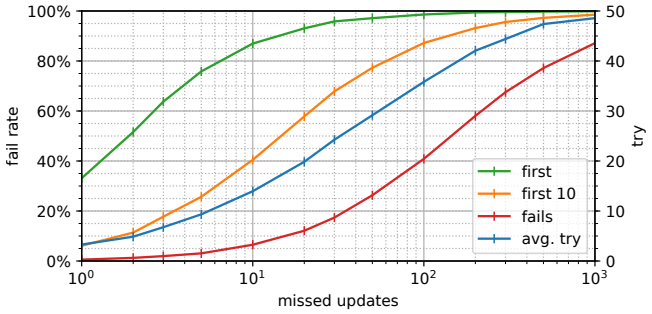

Figure 5: Simulation results for the random PoI repair for increasingly higher number of missed updates.

certificate is located in. This model is shown in Figure 6. We assume that each missed update has a $1/(2^{clvl})$ chance to be in the same part as an outdated $\mathcal{Node}$, as the distribution of leaf positions is uniform (cf. Section 2.1). The differently colored lines show how many missed updates a *LC* can handle, with a probability less than the target percentage. The dotted line shows the storage overhead for each cache level. Our simulations confirmed the theoretical suggestion without significant deviations.

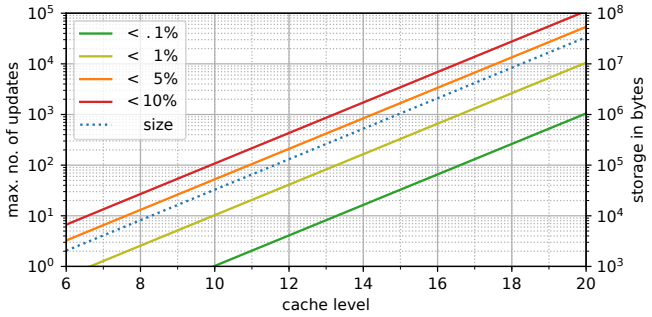

Figure 6: Theoretical number of missable updates for target fail probabilities regarding *LC*-based repair.

6.3 Performance Evaluation

In this section, we evaluate the performance of V'CER. First, in terms of run-time for low-power devices and second, the effectiveness for large-scale applications.

6.3.1 In-Orbit Setup

We implemented a prototype of V'CER to evaluate run-time performance of all basic operations and distributed repair. The prototype is written in Python. While Python is not optimal in terms of performance on constrained devices, we believe that it suffices for a prototype. Further, we used Python's `hashlib` for hash computations, which internally calls the OpenSSL native library [42]. Hash computations constitute the majority of computational overhead in V'CER. The prototype is available on GitHub[3].

To show the feasibility for one of the most challenging constrained networks, satellite networks, we deployed the prototype on ESA's OPS-SAT satellite, an in-orbit platform that is open for any party to register and upload experiments [19]. For this, we adjusted our prototype's code to run as a NanoSat MO Framework app [23], a straightforward way to run code on OPS-SAT, as it provides crucial services, such as data en-/decoding for transmissions. For this, we had to convert our prototype to Java, a requirement to use the MO Framework. The Java prototype uses the internal `BigInteger` class for the bit-operations and the internal `MessageDigest` class for the hash calculations. On the ground, we ran $\mathcal{CA}$ and deployed two additional devices, which were connected to ESA's satellite dish via a SSH tunnel to transmit data from and to the satellite. OPS-SAT uses the S-Band frequencies as the main link for data communication, with up to 256 kbit/s for communication from the satellite and 1 Mbit/s to the satellite [19]. Due to OPS-SAT's polar orbit and a single satellite dish in central Europe, a rough estimate for the amount of communication opportunities is less than 6 passes per day, each with less than 10 minutes of varying bandwidth capacity. This allowed us to successfully test V'CER's feasibility on a deployed satellite with constrained communication, including revocation checks, uploading $\mathcal{CA}$ updates, and distributed repairs between devices.

In the following, we present performance measurements over 1000 executions for the most complex aspects of V'CER. We use SHA-256 as a hash algorithm and *secp256r1* ECDSA for signatures. OPS-SAT runs on a Altera Cyclone V SoC with 800MHz and 1GB of RAM, while on the ground one device runs on a Raspberry Pi 3B+ with 1.4 GHz and 4GB of RAM, the other on a Raspberry Pi Zero W with 1GHz and 512MB of RAM. Note that OPS-SAT also runs other crucial systems in parallel, such as the Attitude Determination and Control System (ADCS). Furthermore, while OPS-SAT is quite powerful compared to other deployed satellites, upcoming satellite hardware will be more powerful, e.g., see the DAHLIA project [14] or the RAD5500 [4]. Results in Table 1 indicate that even with a much weaker (in terms of performance and RAM) system the execution of the individual steps is still fast, keeping most execution times in the range of approximately less than 40ms. An exception is processing 20 update PoIs, which is quite a high number for a single epoch and expected to be rare. This makes V'CER practical even for low-power devices, and thus fulfills requirement R.4 for low-performance devices.

[3] https://github.com/vcer4pki/VCER

Table 1: Run-time measurements of V'CER, @A refers to the Raspberry Pi 3B+, @B refers to the Rasbperry Pi Zero W, @C refers to the OPS-SAT satellite.

Operation	@A [ms]	@B [ms]	@C [ms]
Signature check & $\mathcal{A}ggr$ exchange	2.915	6.869	31.008
PoI authentication	3.776	19.847	228.083
Processing 20 update PoIs	76.929	407.744	4544.915
Single PoI repair	7.574	39.377	452.260
LC (*clvl* = 7) repair	6.191	29.474	264.278

6.3.2 Large-Scale Performance

To evaluate V'CER in terms of scalability, we implemented a large-scale network simulation in Python, directly leveraging the prototype described in Section 6.3.1, running on a Intel Xeon CPU E5-2650 v3 @ 2.30GHz with 10 cores and 256GB of memory. The simulation runs are executed with an increasing number of $\mathcal{N}odes$, from 10 000 up to a million, over the course of 4 simulated weeks. In line with related work, we simulate the aspects that affect V'CER's efficiency, i.e., communication overhead over the entire network. For instance, delays for individual communications between nodes will not significantly affect the system. As with the accompanying example in Section 5, we use the SHA256 hash function and *secp256r1* ECDSA for signatures. Further, $\mathcal{VF}$ is split into 52 epochs, representing 52 weeks of a lifetime of one year per certificate, and 7 parities consisting of 2 Bytes for each $\mathcal{A}_c$.

Each simulated week, an epoch change occurs, which executes the forest prune (cf. Section 5.1) and issues 0.1% new certificates (~5% yearly). On each day in the simulation, 0.028% of the active certificates are revoked (~10% yearly), which are re-issued the next day in the newest epoch. For example, as IoT devices are notorious for having security bugs, we assume a relatively large share of yearly revocations. Our share of revocations is based on the estimate of certificates affected by the Heartbleed bug in the Internet [35], one of the biggest recorded revocation event. Splitting the yearly revocations in days is in-line with related work [46], as even in the case of Heartbleed, revocations occurred gradually [35].

Every time $\mathcal{CA}$ distributes an update, a random share of $\mathcal{N}odes$ do not receive the update, i.e., become outdated. We simulated different missing shares of 10%, 30%, and 50% in separate runs. 10% of $\mathcal{N}odes$ are cachers that additionally store a *LC* with *clvl* = 7; yet, also can miss updates. For outdated $\mathcal{N}odes$ to repair their own validation information, each $\mathcal{N}ode$ encounters 5 random $\mathcal{N}odes$ each hour. During those encounters, both $\mathcal{N}odes$ exchange $\mathcal{A}ggr$, update their $\mathcal{VF}$ if applicable as well as notice outdated validation information, and if only one $\mathcal{N}ode$ is outdated try the distributed repair. After an outdated $\mathcal{N}ode$ has met 30 up-to-date $\mathcal{N}odes$ and was not able to repair its validation information, it will give up and request it directly from $\mathcal{CA}$.

Figure 7 shows the measurements of two failure percentages. The *failed repairs* shows the share of outdated $\mathcal{N}odes$, which were not able to distributively repair their validation information and instead, required a direct request to $\mathcal{CA}$. Even in the drastic case of 50% $\mathcal{N}odes$ missing updates with 1 million $\mathcal{N}odes$ total, more than 93% of the outdated $\mathcal{N}odes$ are able to collaboratively repair their PoI. Therefore, V'CER fulfills requirement R.1 and R.5 for a $\mathcal{N}ode$'s own validation proof. Outdated $\mathcal{N}odes$ need to meet 8.9, 9.3, and 10.1 up-to-date $\mathcal{N}odes$ on average until the distributed repair is successful, for 10%, 30%, and 50% of $\mathcal{N}odes$ missing updates respectively. Assuming only one of the two $\mathcal{N}odes$ in an encounter needs an up-to-date PoI, e.g., to establish a secure channel without mutual authentication, we also measured the percentage of encounters in which both $\mathcal{N}odes$ are outdated. With 1 million $\mathcal{N}odes$ and 50% $\mathcal{N}odes$ missing updates, there are less than 5% of encounters, in which both do not have an up-to-date PoI.

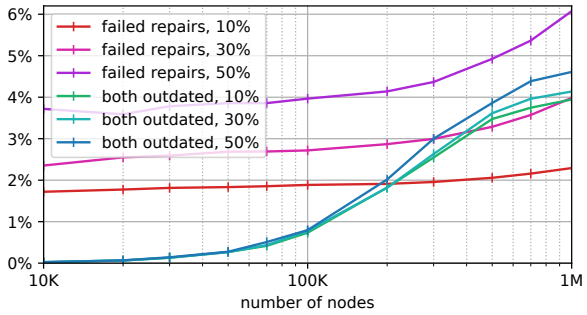

Figure 7: Simulation results for percentage of $\mathcal{N}odes$ failing to repair their PoI, when they missed $\mathcal{CA}$ updates, and the percentage of all encounters where both $\mathcal{N}odes$ are outdated; both for different missing shares

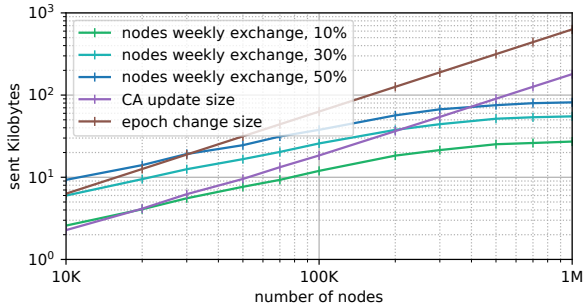

Figure 8: Simulation results for sent out Kilobytes for daily $\mathcal{CA}$ updates, weekly epoch change update size, and average weekly exchanged communication between $\mathcal{N}odes$ for different missing shares.

Figure 8 depicts three key communication overhead measurements. The graphs for *nodes weekly exchange* represents the average amount of exchanged Kilobytes directly between $\mathcal{N}odes$, per node per week. This includes the $\mathcal{A}ggr$ exchange, $\mathcal{VF}$ repair, and distributed repair of outdated PoIs. With 1 mil-

lion $\mathcal{N}odes$ and 10% of $\mathcal{N}odes$ missing updates, $\mathcal{N}odes$ directly exchange 27.2KB per week and 81.6KB with 50%. The *CA update size* is the average update size sent out daily by $\mathcal{CA}$. This contains both the revocations as well as the re-issuance the day after revocation (0.056% of total number of $\mathcal{N}odes$). This metric is independent of the missing share and for 1 million $\mathcal{N}odes$, the daily $\mathcal{CA}$ update is 179.3KB on average. Finally, the *epoch change size* shows the size of the weekly update sent out by $\mathcal{CA}$. Independent of the missing share as well, the epoch change is 629.8KB. Note, as mentioned in Section 5.2, that the epoch change update does not need to be distributed to all of the network and only needs to be sent to $\mathcal{N}odes$ affected by epoch change.

7 Discussion

This section discusses various practical aspects of V'CER.

PKI integration. Since V'CER only deals with certificate validation, it needs a Public Key Infrastructure (PKI) as the basis for deployment. To the best of our knowledge, V'CER can be used alongside any PKI scheme, similar to OCSP Stapling, by having the prover provide validation evidence (see Section 8). Ideally, the authentication PoI is directly sent along with the certificate during the handshake protocol to reduce communication overhead. Furthermore, to allow audit of individual changes, $\mathcal{CA}$ can keep a log of all individual operations, such as revocation of a certificate. External observers need only to store the resulting $\mathcal{A}ggr$ for each $\mathcal{CA}$ update, allowing them to verify $\mathcal{CA}$ logs after the fact.

Multiple Certificate Authorities. While we assume a single Certificate Authority (CA) throughout the paper, in some real-world use-cases there multiple CAs can be involved. This can be accommodated by employing a committee that acts as a single $\mathcal{CA}$ via consensus among the members, for all operations. For example, a signature scheme to allow for such a strategy are BLS signatures [9], which can model the necessary threshold-based signatures to create $\mathcal{A}_s$, verifiable by $\mathcal{N}odes$. Alternatively, each member can maintain its own $\mathcal{VF}$ and be responsible for certifying their distinct group of nodes in the network. In such a case, a node from one vendor can simply collect and keep the most recent $\mathcal{VF}$ of other vendors to authenticate their nodes. $\mathcal{N}odes$ would only have to deal with PoI updates regarding their vendor's $\mathcal{VF}$, while only keeping the other $\mathcal{VF}$ up-to-date.

Caching Certificates. If a $\mathcal{N}ode$ regularly communicates with another $\mathcal{N}ode$, the other $\mathcal{N}ode$'s certificate and PoI can be cached to avoid subsequent redundant checks. A $\mathcal{N}ode$ can keep cached PoIs up-to-date, just like it does for its own PoI. This way, there is a good chance that cached certificates are ready for use, even after many updates. If any cached PoIs get revoked, they can be discarded. Furthermore, since a $\mathcal{N}ode$ can provide many up-to-date PoIs to an outdated node, this strategy can vastly improve the direct PoI repair approach, as discussed in Section 5.4.1).

Dynamic Cache Sizes. While we assumed uniform use of *LC*, i.e., all cacher $\mathcal{N}odes$ use the same *clvl* across all epochs in $\mathcal{VF}$, it may be advantageous in some scenarios to use a dynamic cache size. On the one hand, cacher $\mathcal{N}odes$ with bigger storage can also keep bigger caches, allowing them to succeed more often when executing distributed repairs. On the other hand, the same cacher may use different cache sizes for different epochs. For example, if most updates are expected within newer epochs, a cacher can use a bigger cache for these than for almost expired epochs. Note, a bigger *LC* can easily construct a smaller one, e.g., when transitioning to a smaller cache on an epoch change.

8 Comparison to OCSP and CRL schemes

This section focuses on comparing V'CER to the most commonly found types of schemes found in the revocation space, Online Certificate Status Protocol (OCSP), CRLs, and variants that improve on CRLs.

OCSP Stapling. In OCSP, the verifier of a certificate directly requests a confirmation from $\mathcal{CA}$ that the certificate is not revoked. Due to the on-demand nature of OCSP, it is not directly applicable to our system model (cf. Section 3), as it requires a fast and reliable connection to $\mathcal{CA}$ to properly function. Nevertheless, there is also OCSP Stapling. Here, the proving party requests the OCSP for its own certificate, stores it, and directly delivers it to the verifier. Each OCSP has a validity period that enforces a fresh OCSP eventually. A single OCSP request is around 4KB in size [35]. However, if we assume the same validity period as in our accompanying example of a single day, this means all $\mathcal{N}odes$ in the network each needs to request a new OCSP daily. For example, even with 100 000 $\mathcal{N}odes$, this would already amount to around 390MB of communication overhead across the network, compared to our 18.4KB via the universal $\mathcal{CA}$ update.

Traditional CRLs. With the CRL approach, $\mathcal{N}odes$ need to maintain and store the entire CRL. However, these lists can grow very large. For example, in the Internet the CRLs size for the median certificate is 51KB, ranging up to 76MB for a single CRL [35]. Yet, even assuming an optimal CRL, i.e., storing only strictly necessary entries, would already require over 3.6MB of storage to cover for the whole year with 1 million $\mathcal{N}odes$ (as in our evaluation)—assuming an average of 38 Bytes per entry [35]. In contrast, $\mathcal{N}odes$ in V'CER only need to store ~3KB to achieve the same.

Enhanced CRLs. While traditional CRLs can be very large, there are numerous works to significantly reduce the storage and update overhead. To the best of our knowledge, the most notable works in this space recently are CRLite [32] and Let's Revoke [46]. Both aim to reduce storage and update sizes of revocation information for the Internet. CRLite uses a cascade of decreasingly smaller bloom filters to aggregate revocation information. For 1 million certificates, this requires ~112.5KB of storage [32]. Delta updates model the difference between the old and the new bloom filters on all levels, which can be applied in a XOR-like fashion. Modeling a change of 0.056% in the contained certificates, e.g., on daily revocations plus re-issuance (as in our evaluation), requires around 10% to 50% of the original filter size (according to the measurements presented in Figure 6 in [32]). Let's Revoke [46] uses a bitvector per expiration day per CA, which flags if consecutively numbered certificates have been revoked. As these bitvectors are expected to have many zeroes, they can further be compressed to save storage space. With 1 million certificates and 10% of them revoked, this requires ~70KB of storage (~125KB when not compressed) [46]. For delta updates, bitvectors are constructed that can be applied to the original bitvectors via simple bit operations. A compressed update modeling 0.056% of certificate changes is ~2KB in size [46]. Note, these schemes are specifically designed for the Web's PKI and are not concerned with constrained networks.

Both approaches require nodes that missed updates to directly contact any authority or their delegates (e.g., *aggregators* in [32]) to keep up-to-date. In this case, outdated nodes need to individually request the specific range of updates they missed. On the one hand, this requires careful placement of delegates regarding the targeted topology to ensure coverage, which is difficult in many constrained networks, e.g., due to the nature of a dynamic topology. On the other hand, all affected nodes would need to individually request missed updates. For example, a 2KB delta update in Let's Revoke [46] for 1 million nodes with 10% of nodes missing a *single* update would surmount to 195MB of communication overhead across the network.

In contrast, V'CER only requires less than 3KB of storage on each node and uses universal updates that can be applied at any time, e.g., even though a node missed an update, it may receive a subsequent update in the meantime, which may repair its proof without further requests. Nodes in V'CER can also help each other to distributively repair their proof, eliminating the need for individual requests to the CA for the principal share of nodes (as shown in Section 6.3.2). Further, even if the distributed repair fails for nodes, with the up-to-date and quickly disseminated $\mathcal{VF}$ they can still correctly validate other certificates. Finally, when nodes *do* need to request their fresh PoI from the CA, it only requires less than 1KB of communication overhead each.

9 Related Work

In the following, we will examine the related work in the field of certificate validation. Aside the ones mentioned throughout this section, there are also works that focus on constructing PKIs for Internet of Things (IoT); yet, for revocation checks, they rely on CRLs [50], on-demand checks [24] that we both discussed in Section 8, or are blockchain-based [28,45], which we examine later in this section. Otherwise, recent works are dominated by observer-based approaches among CAs, end-user-based approaches, or blockchain-based approaches. Finally, we compare preceding works focusing on efficient validation for untrusted validation directories with V'CER.

Observer-based approaches. These works focus on restricting maliciously acting CAs by monitoring them for suspicious behavior. Most prominent in this area is Certificate Transparency (CT) [33], already adopted by many CAs and browsers for the Internet. In this approach, an append-only Merkle Tree is used to log all issued certificates by a CA as hash leaves. On issuing, the CA publishes the new certificates along with a *consistency proof*, which proves that the previous Merkle Tree is contained in the now extended tree. Observers, such as other CAs, check if the new certificates contain any unjustly issued ones, e.g., for domains that the issuing CA is not responsible for. The consistency proof ensures all certificates were published and correctly appended to the CA's Merkle Tree. On the end-user side, aside from validating the certificate directly with the issuing CA, the client additionally requests the PoI of the certificate from multiple observers. Falsely issued certificates then become apparent; yet, this does not cover revocations. An informal report by Laurie and Kasper hints at extending CT with Sparse Merkle Trees to provide *Revocation Transparency* [34].

Enhanced Certificate Transparency [43] aims to extend the CT approach to also handle revocations efficiently. The authors argue that search in the Revocation Transparency proposal [34] remains linear in the number of issued certificates. Aside an append-only Merkle Tree as in CT, the paper introduces an additional tree that is ordered by the subject identities, allowing for logarithmic look-up of revocations. Nevertheless, to ensure consistency, each observer still needs to verify all certificates published and their inclusion into the respective trees. The paper mentions its approach can be extended in a distributed manner, so users require less trust into the CAs. However, this would require random monitoring of CAs by all users as well as gossiping the observed information to identify inconsistencies. Further work focuses on improving resilience of these approaches against colluding CAs, such as AKI [30] or ARPKI [5].

While these observer-based approaches use similar cryptographic structures to V'CER, i.e., hash trees and PoIs, they aim at the orthogonal goal of limiting malicious behavior of CAs by giving them the means to efficiently monitor each

other. V'CER, on the other hand, provides efficient certificate validation for end users among each other. Further, V'CER is not reliant on having reliable connectivity to trusted parties for validation.

End-user-based approaches. These approaches aim to modify the observer-based schemes to allow end-users to monitor for inconsistencies regarding their certificates themselves. In CONIKS [37], a Prefix Merkle Tree is used, along with randomly generated user IDs, to protect the privacy of users. Here, all users need to constantly check all issuing parties for certificates of their own domain, to recognize abuse. When the issuing CA updates its tree in any way, it also needs to re-issue all of its users' PoIs so they are correct with respect to the new tree root. Revocation is handled by on-demand requests to the respective CA. DTKI [53] aims to provide users full data ownership for their certificates by introducing a Merkle Tree log per individual domain. Each log is maintained in a decentralized database, based on consensus of multiple independent entities. Users need to gossip all tree roots among each other to prevent problems on network partitioning. Additionally, DTKI revocation is also done with on-demand requests.

These approaches provide users capabilities to monitor their own domains. However, they are very communication heavy and rely on on-demand requests to check for revocation, making them unsuitable for constrained networks. In contrast, V'CER requires minimal communication by only distributing a number of hashes, i.e., $\mathcal{VF}$. Further, it allows nodes to collaboratively repair their individual validation proof, mostly without the need to contact any authorities.

Blockchain-based approaches. The works in this area aim to shift trust from the CAs to the blockchain. One of the first proposals in this area is Certcoin [21], which builds a decentralized PKI based on Namecoin [29]. The idea is to simply store all certificate updates, including revocation, on the blockchain. A certificate owner may send the PoI for the block containing the certificate for validation, similar to Bitcoin's Simplified Payment Verification [6]. This requires a user to monitor and store all block headers, instead of the full blockchain. Analogously, Blockstack [1] directly builds on Bitcoin for improved security and separates different abstraction layers for simplified access. For certificate validation, a user contacts multiple full nodes (i.e., storing the entire blockchain) and checks if the responses are consistent with each other. EthIKS [10] simply puts CONIKS [37] on top of Ethereum [11] and uses smart contracts for global monitoring and validation of certificates.

All these approaches share the notion of having, at least indirect, access to the blockchain. This is difficult to guarantee in constrained networks or requires significant storage overhead on limited devices. On the contrary, V'CER only requires minimal storage overhead, without the need to be constantly connected to any specific nodes.

PKI for dynamic networks. There are approaches specifically aiming to construct PKIs for constrained networks. There are many works focusing on Mobile Ad-Hoc Networks [12,36,51,52] and Delay-Tolerant Networks [15,17,41]. Note, the latter is considered the state of the art for satellite networks. They generally aim to distribute the role of the CA among the network of nodes. This usually requires a lot of coordination between all nodes. However, the key aspect in the context of this work is how revocation is handled. For this, the approaches either rely on traditional CRLs [51, 52], individual revocation information aggregated by exchanging them among the network [12, 15, 36], or by simply limiting the lifetime of certificates [12, 41].

To the best of our knowledge, all schemes in these areas either share similar problems of CRLs, require communication heavy coordination between nodes, or introduce an additional vulnerability window. In V'CER the vulnerability window can be kept to a minimum, as it relies on distributing only $\mathcal{VF}$. Otherwise, it minimizes the communication overhead required, making it well suited for constrained networks.

Efficient Validation Directories. There have been several works aiming to provide efficient certificate validation [18, 22, 31, 39, 40]. They use authenticated data structures providing efficient validation proofs, to allow the use of untrusted directories capable of answering validation requests, while removing the need for distributing entire CRLs. The reduced communication overhead enables shorter validity periods of revocation information, e.g., daily or even hourly. The data structures are either based on trees for revocation [31, 40], trees covering both revocation and validation [22], hash-chains containing all validity periods [39], or a combination of both hash-chains and trees [18].

These approaches aim at supplying directories with fresh validation information by regular updates from the CAs, e.g., by distributing tree updates [22, 31, 40]. In contrast, V'CER supplies all nodes with fresh validation information by only distributing $\mathcal{VF}$. Nodes can additionally collaborate without any special directories to keep their proofs up-to-date.

10 Conclusion

In this work, we presented V'CER, a novel certificate validation scheme, which is designed to work in a distributed and efficient manner. V'CER can be used to augment any PKI scheme, enabling it to work even in constrained networks. This is achieved by introducing data structures and operations that allow for fast dissemination of validation information as well as a collaborative way for nodes to keep up-to-date. We have demonstrated the efficacy and efficiency of V'CER with large-scale simulations modeling a constrained network.

Acknowledgments

The authors of the Technical University of Darmstadt were supported by the European Space Operations Centre with the Networking/Partnering Initiative and the European Union's Horizon 2020 Research and Innovation program under Grant Agreement No. 952697 (ASSURED). Gene Tsudik's work was supported in part by NSF awards SATC-1956393 and CICI-1840197, as well as a subcontract from Peraton Labs.

References

[1] Muneeb Ali, Jude Nelson, Ryan Shea, and Michael J Freedman. Blockstack: A global naming and storage system secured by blockchains. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 181–194, 2016.

[2] Amazon. AWS Ground Station. https://aws.amazon.com/ground-station, 2022.

[3] Ars Technica. Space-grade CPUs: How do you send more computing power into space? https://arstechnica.com/science/2019/11/space-grade-cpus-how-do-you-send-more-computing-power-into-space/, 2019.

[4] BAE Systems. RAD5545™ SoC based single board computer. https://www.baesystems.com/en/our-company/our-businesses/electronic-systems/product-sites/space-products-and-processing/radiation-hardened-electronics-produ, 2021.

[5] David Basin, Cas Cremers, Tiffany Hyun-Jin Kim, Adrian Perrig, Ralf Sasse, and Pawel Szalachowski. Arpki: Attack resilient public-key infrastructure. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 382–393, 2014.

[6] bitcoin.org. SPV, Simplified Payment Verification. https://developer.bitcoin.org/devguide/block_chain.html#transaction-data, 2022.

[7] Bluetooth SIG. Bluetooth Mesh Networking. https://www.bluetooth.com/learn-about-bluetooth/recent-enhancements/mesh/, 2022.

[8] Sharon Boeyen, Stefan Santesson, Tim Polk, Russ Housley, Stephen Farrell, and Dave Cooper. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008.

[9] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

[10] Joseph Bonneau. Ethiks: Using ethereum to audit a coniks key transparency log. In *International Conference on Financial Cryptography and Data Security*, pages 95–105. Springer, 2016.

[11] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://ethereum.org/en/whitepaper/, 2013.

[12] Srdjan Capkun, Levente Buttyán, and J-P Hubaux. Self-organized public-key management for mobile ad hoc networks. *IEEE Transactions on mobile computing*, 2(1):52–64, 2003.

[13] CSO Online. 8 Famous Software Bugs in Space. https://www.csoonline.com/article/3404528/8-famous-software-bugs-in-space.html, 2013.

[14] Dahlia Consortium. Dahlia Project. https://dahlia-h2020.eu/, 2022.

[15] Chris I Djamaludin, Ernest Foo, S Camtepe, and Peter Corke. Revocation and update of trust in autonomous delay tolerant networks. *Computers & Security*, 60:15–36, 2016.

[16] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983.

[17] Karim El Defrawy, John Solis, and Gene Tsudik. Leveraging social contacts for message confidentiality in delay tolerant networks. In *2009 33rd annual IEEE international computer software and applications conference*, volume 1, pages 271–279, 2009.

[18] Farid F Elwailly, Craig Gentry, and Zulfikar Ramzan. Quasimodo: Efficient certificate validation and revocation. In *International Workshop on Public Key Cryptography*, pages 375–388, 2004.

[19] European Space Agency. OPS-SAT. https://www.esa.int/Enabling_Support/Operations/OPS-SAT, 2022.

[20] Stephen Farrell, Howard Weiss, Susan Symington, and Peter Lovell. Bundle Security Protocol Specification. RFC 6257, May 2011.

[21] Conner Fromknecht, Dragos Velicanu, and Sophia Yakoubov. A decentralized public key infrastructure with identity retention. *IACR Cryptology ePrint Archive*, 2014.

[22] Irene Gassko, Peter S Gemmell, and Philip MacKenzie. Efficient and fresh certification. In *International Workshop on Public Key Cryptography*, pages 342–353. Springer, 2000.

[23] GitHub. NanoSat MO Framework. https://github.com/esa/nmf-mission-ops-sat, 2022.

[24] Joel Höglund, Samuel Lindemer, Martin Furuhed, and Shahid Raza. Pki4iot: Towards public key infrastructure for the internet of things. *Computers & Security*, 89:101658, 2020.

[25] IEEE. 802.11s standard. https://standards.ieee.org/standard/802_11s-2011.html, 2011.

[26] Interorbital Systems. IOS CubeSat Kits. https://www.interorbital.com/Cubesat%20Kits.php, 2022.

[27] IPNSIG. InterPlanetary Networking Special Interest Group. https://ipnsig.org/about/, 2022.

[28] Wenbo Jiang, Hongwei Li, Guowen Xu, Mi Wen, Guishan Dong, and Xiaodong Lin. Ptas: Privacy-preserving thin-client authentication scheme in blockchain-based pki. *Future Generation Computer Systems*, 96:185–195, 2019.

[29] Harry A Kalodner, Miles Carlsten, Paul Ellenbogen, Joseph Bonneau, and Arvind Narayanan. An empirical study of namecoin and lessons for decentralized namespace design. In *WEIS*. Citeseer, 2015.

[30] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perrig, Collin Jackson, and Virgil Gligor. Accountable key infrastructure (aki) a proposal for a public-key validation infrastructure. In *Proceedings of the 22nd international conference on World Wide Web*, pages 679–690, 2013.

[31] Paul C Kocher. On certificate revocation and validation. In *International Conference on Financial Cryptography*, pages 172–177. Springer, 1998.

[32] James Larisch, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. Crlite: A scalable system for pushing all tls revocations to all browsers. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 539–556. IEEE, 2017.

[33] Ben Laurie. Certificate transparency. *Communications of the ACM*, 57(10):40–46, 2014.

[34] Ben Laurie and Emilia Kasper. Revocation transparency. *Google Research, September*, 2012.

[35] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. An end-to-end measurement of certificate revocation in the web's pki. In *Proceedings of the 2015 Internet Measurement Conference*, pages 183–196, 2015.

[36] Haiyun Luo, Petros Zerfos, Jiejun Kong, Songwu Lu, and Lixia Zhang. Self-securing ad hoc wireless networks. In *ISCC*, volume 2, pages 548–555, 2002.

[37] Marcela S Melara, Aaron Blankstein, Joseph Bonneau, Edward W Felten, and Michael J Freedman. Coniks: Bringing key transparency to end users. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 383–398, 2015.

[38] Ralph C Merkle. A certified digital signature. In *Conference on the Theory and Application of Cryptology*, pages 218–238, 1989.

[39] Silvio Micali. Scalable certificate validation and simplified pki management. In *1st Annual PKI research workshop*, volume 15, 2002.

[40] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. *IEEE Journal on selected areas in communications*, 18(4):561–570, 2000.

[41] Rabin Patra, Sonesh Surana, and Sergiu Nedevschi. Hierarchical identity based cryptography for end-to-end security in dtns. In *2008 4th International Conference on Intelligent Computer Communication and Processing*, pages 223–230, 2008.

[42] Python Official Documentation. hashlib - Secure hashes and message digests. https://docs.python.org/3/library/hashlib.html, 2022.

[43] Mark Dermot Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS Symposium 2014*, pages 1–14, 2014.

[44] Stefan Santesson, Michael Myers, Rich Ankney, Ambarish Malpani, Slava Galperin, and Dr. Carlisle Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960, June 2013.

[45] Ankush Singla and Elisa Bertino. Blockchain-based pki solutions for iot. In *2018 IEEE 4th International Conference on Collaboration and Internet Computing (CIC)*, pages 9–15. IEEE, 2018.

[46] Trevor Smith, Luke Dickinson, and Kent Seamons. Let's revoke: Scalable global certificate revocation. In *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

[47] Space.com. Hackers Interfered With 2 US Government Satellites. https://www.space.com/13423-hackers-government-satellites.html, 2011.

[48] Space.com. Starlink: SpaceX's satellite internet project. https://www.space.com/spacex-starlink-satellites.html, 2022.

[49] Union of Concerned Scientists. UCS Satellite Database. https://www.ucsusa.org/resources/satellite-database, 2021.

[50] Minmei Wang, Chen Qian, Xin Li, and Shouqian Shi. Collaborative validation of public-key certificates for iot by distributed caching. In *IEEE INFOCOM Conference on Computer Communications*, pages 847–855, 2019.

[51] Bing Wu, Jie Wu, Eduardo B Fernandez, and Spyros Magliveras. Secure and efficient key management in mobile ad hoc networks. In *19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[52] Seung Yi and Robin Kravets. Moca: Mobile certificate authority for wireless ad hoc networks. Technical report, 2004.

[53] Jiangshan Yu, Vincent Cheval, and Mark Ryan. Dtki: A new formalized pki with verifiable trusted parties. *The Computer Journal*, 59(11):1695–1713, 2016.

[54] Z-Wave Alliance. About Z-Wave Technology. https://z-wavealliance.org/about_z-wave_technology/, 2022.

[55] Z-Wave Alliance. Z-Wave Specifications. https://z-wavealliance.org/z-wave-specifications/, 2022.

[56] ZigBeeAlliance. Zigbee specification. https://zigbeealliance.org/wp-content/uploads/2019/11/docs-05-3474-21-0csg-zigbee-specification.pdf, 2015.

A In-depth Operation Description

This section shows the in-depth working of the basic operations for epoch trees.

A.1 Preliminaries and Notation

Table 2 summarizes our notation for the operations. When addressing a position in the SMT the term *depth* or *depth level* is used, where a higher number depth means closer to the leaves and a lower number depth means closer to the root. This is also illustrated in Figure 1 (b). When describing a for-loop in the algorithms, note that "**for** $i \leftarrow 0$ **to** x **do**" means i gets assigned $[0,x)$ in the course of the loop. For brevity, we skip the checks for empty hashes in *EmptyHashesList* and simply check for $\varnothing$.

A.2 CA Operations

This section describes how V'CER does basic operations. We use an incremental approach using the Look-Up-Table

Table 2: Variables and operation definitions.

Variables	
LUT	Look-Up-Table storing hashes of all non-empty branches and leaves of SMT
$\lvert\mathbb{H}\rvert$	Bit length of a hash digest and depth of SMT
$\lvert l\rvert$	Number of elements in list l
LC	Level-Cache-List sorted by position of the cache elements in SMT
$clvl$	Depth of LC
Operations	
$int\langle p\rangle$	Access bit at position p of int from the right
$l[p]$	Access element at position p in list l
$LUT[p,d]$	Get hash at position p at depth d, the last d bits of p will be ignored, e.g., $d=0$ returns SMT_r, $d=\lvert\mathbb{H}\rvert$ returns a leaf
$\sim x$	Flip bit(s) of x

LUT, instead of constructing the tree in one go for a set of leaves [34]. This avoids having to reconstruct the entire epoch tree on an update to it. As V'CER works by regularly updating the respective epoch trees, this helps to significantly reduce the calculation overhead on the $\mathcal{CA}$. This comes at the cost of having to store the LUT.

A.2.1 Add Leaf to SMT

This operation is executed by the $\mathcal{CA}$ and adds one leaf hash to an epoch tree while updating the LUT for subsequent operations. Figure 1 (b) illustrates how it works for c_1. Starting from the leaf, the operation will go up the tree, look-up the respective neighbor in the LUT (see green nodes), calculate and set the intermediate nodes along the leaf's path in the LUT (see green line), and repeat this process until reaching the root. For the initial epoch tree construction or processing multiple updates, this operation is called multiple times individually for each leaf.

Algorithm 4 describes the operation in detail. First, it sets the hash to be added on the respective leaf position in line 1. It traverses the tree from bottom to top, i.e., starting at the right-most bit in the leaf going successively left. In line 4 we construct the position of the neighbor at the current depth by flipping the bit representing this depth. Depending on which side the leaf and neighbor is, in lines 5 to 10, we look-up the left and right hash to concatenate them in correct order, hash them, and set the result at the respective position in the LUT (line 11). This process will be repeated for all depths until the root is reached. For revocation, the same operation is used regarding the leaf to be removed, except in line 1 we set the leaf to $\mathtt{H}(\varnothing)$ instead.

Algorithm 4 `add_leaf` function for adding a new leaf hash and recalculating the tree while updating the LUT

Input: *leaf_hash*, *LUT*
Output: $\mathcal{E}_r$, *LUT*

1: $LUT[leaf\_hash, |\mathbb{H}|] \leftarrow leaf\_hash$
2: **for** $i \leftarrow 0$ **to** $|\mathbb{H}|$ **do**
3: $\quad neighbor \leftarrow leaf\_hash$
4: $\quad neighbor\langle |\mathbb{H}| - i\rangle \leftarrow \sim neighbor\langle |\mathbb{H}| - i\rangle$
5: $\quad$ **if** $leaf\_hash\langle |\mathbb{H}| - i\rangle$ = True **then**
6: $\quad\quad left\_hash \leftarrow LUT[neighbor, (|\mathbb{H}| - i)]$
7: $\quad\quad right\_hash \leftarrow LUT[leaf\_hash, (|\mathbb{H}| - i)]$
8: $\quad$ **else**
9: $\quad\quad left\_hash \leftarrow LUT[leaf\_hash, (|\mathbb{H}| - i)]$
10: $\quad\quad right\_hash \leftarrow LUT[neighbor, (|\mathbb{H}| - i)]$
11: $\quad LUT[leaf\_hash, (|\mathbb{H}| - i - 1)] \leftarrow \mathtt{H}(left\_hash \,||\, right\_hash)$
12: **return** $LUT[\varnothing, 0]$, *LUT*

A.2.2 PoI Construction

This operation is executed by the $\mathcal{CA}$ to construct a PoI for the given leaf hash. The *LUT*, populated by the `add_leaf` operation, is used for the PoI construction. Only necessary hashes will be put in the PoI by excluding empty hashes. However, this additionally requires a $|\mathbb{H}|$-bit sized path bitmap for verification, to know which depth each element in the PoI represents.

Algorithm 5 `calc_poi` function for calculating the proof of inclusion for a leaf

Input: *leaf_hash*, *LUT*
Output: *path*, *path_bitmap*

1: $path[] \leftarrow \varnothing$
2: $path\_bitmap \leftarrow 0$
3: **for** $i \leftarrow 0$ **to** $|\mathbb{H}|$ **do**
4: $\quad neighbor \leftarrow leaf\_hash$
5: $\quad neighbor\langle |\mathbb{H}| - i\rangle \leftarrow \sim neighbor\langle |\mathbb{H}| - i\rangle$
6: $\quad neighbor\_hash \leftarrow LUT[neighbor, (|\mathbb{H}| - i)]$
7: $\quad$ **if** $neighbor\_hash \neq \varnothing$ **then**
8: $\quad\quad path\_bitmap\langle i\rangle \leftarrow$ True
9: $\quad\quad path$`.append(`$neighbor\_hash$`)`
10: **return** *path*, *path_bitmap*

The operation is shown in Algorithm 5. It works similar to `add_leaf`, by starting from the bottom of the tree and working its way up. For each depth, the neighbor position for the given leaf is calculated and looked-up in line 6. If the looked-up hash is not empty, it will be appended to the PoI and the bit at the current depth is set in the path bitmap. For a revoked leaf the operation does not require any changes.

A.2.3 PoI Verification

For this operation, we calculate the root resulting from a PoI. Given the leaf to be verified, its PoI, and respective path bitmap, the operation can be executed by anyone. If this root matches the expected epoch root, the certificate represented by the leaf is valid. Per default, the operation will calculate the root, yet, an optional parameter can be passed to stop at the specified depth-level.

Algorithm 6 `calc_path_root` function for calculating the root of a proof of inclusion, optionally only until a given depth

Input: *leaf_hash*, *path*, *path_bitmap*, $[lvl \leftarrow 0]$
Output: *root_hash*

1: $result \leftarrow leaf\_hash$
2: **for** $i \leftarrow 0$ **to** $(|\mathbb{H}| - lvl)$ **do**
3: $\quad$ **if** $path\_bitmap\langle i\rangle$ = True **then**
4: $\quad\quad neighbor \leftarrow path[0]$
5: $\quad\quad path$`.popfront()`
6: $\quad$ **else**
7: $\quad\quad neighbor \leftarrow \varnothing$
8: $\quad$ **if** $leaf\_hash\langle i\rangle$ = True **then**
9: $\quad\quad result \leftarrow \mathtt{H}(neighbor \,||\, result)$
10: $\quad$ **else**
11: $\quad\quad result \leftarrow \mathtt{H}(result \,||\, neighbor)$
12: **return** *result*

Algorithm 6 gives a detailed description of the operation. The algorithm starts by setting the given leaf in the *result* variable and then works its way up the tree from the bottom. In the lines 3 to 7 it checks if there is an element in the given PoI for the current depth and if so, it extracts this PoI element. Otherwise, an empty hash is assumed instead. Then both hashes are hashed together in the respective order and set to the *result* variable. This process is repeated for all depths until we reach the root, or the given depth level, if specified.

A.2.4 Level-Cache Construction

The construction of a *LC* by the $\mathcal{CA}$ is shown in Algorithm 7. Note that the elements of *LC* are sorted left-to-right by their position in the epoch tree. Thus, we can simply go through all positions in the *LC*, bit-shift it to the very left regarding the digest size, and use this to simply look-up the value at the depth *clvl*. The $\mathcal{CA}$ can repeat this for all epochs and then distribute the resulting *LC*. If some $\mathcal{Nodes}$ missed this distribution, $\mathcal{Nodes}$ that received the *LC* may share it with others, or construct *LC*s with a smaller *clvl* if requested.

Algorithm 7 `construct_lvl_cache` for constructing a level-cache with a specified depth

Input: *clvl*, *LUT*
Output: *LC*

1: $LC[] \leftarrow \varnothing$
2: **for** $i \leftarrow 0$ **to** 2^{clvl} **do**
3: $\quad position \leftarrow i \ll (|\mathbb{H}| - clvl)$
4: $\quad LC$`.append(`$LUT[position, clvl]$`)`
5: **return** *LC*