LeapChain: Efficient Blockchain Verification for Embedded IoT

Emanuel Regnath
emanuel.regnath@tum.de
Technical University of Munich, Germany

Sebastian Steinhorst
sebastian.steinhorst@tum.de
Technical University of Munich, Germany

ABSTRACT

Blockchain provides decentralized consensus in large, open networks without a trusted authority, making it a promising solution for the Internet of Things (IoT) to distribute verifiable data, such as firmware updates. However, verifying data integrity and consensus on a linearly growing blockchain quickly exceeds memory and processing capabilities of embedded systems.

As a remedy, we propose a generic blockchain extension that enables highly constrained devices to verify the inclusion and integrity of any block within a blockchain. Instead of traversing block by block, we construct a *LeapChain* that reduces verification steps without weakening the integrity guarantees of the blockchain. Applied to Proof-of-Work blockchains, our scheme can be used to verify consensus by proving a certain amount of work on top of a block.

Our analytical and experimental results show that, compared to existing approaches, only LeapChain provides deterministic and tight upper bounds on the memory requirements in the kilobyte range, significantly extending the possibilities of blockchain application on embedded IoT devices.

CCS Concepts: • **Computer systems organization** → *Embedded and cyber-physical systems*; Peer-to-peer architectures;

Keywords: Internet of Things, Embedded, Blockchain, SPV

1 INTRODUCTION

Distributed embedded systems often face the challenge to reach consensus about a global system state. However, traditional consensus protocols, such as PBFT, only work for small network sizes, as nodes need to know all identities and exchange data with each other [1]. With the Internet of Things (IoT) as an emerging future of billions of interconnected devices, this requirement renders these solutions infeasible for scaling [2].

Cryptocurrencies such as Bitcoin [3] reach consensus on global transactions in a large, trustless, and open peer-to-peer network without any central authority. The transactions are stored in a distributed chain of blocks (blockchain), each block securing the order and integrity of previous blocks. While cryptocurrencies focus on transactions of assets, it is possible to store any data in a blockchain [4] and there exist ideas to transfer this technology to the embedded domain, such as for secure peer-to-peer firmware updating and validating [5–7]. For this scenario, a manufacturer would publish a signed hash of the latest firmware on a blockchain, where the signature is verified by all full powered network participants and – if valid – included in the blockchain. An IoT end node could then receive a new firmware from any possible peer and verify its validity and integrity by simply comparing its hash to the hash in the blockchain [4]. This solution is more robust as it avoids a single point of failure, more efficient because the end nodes do not need to perform public key cryptography, and more transparent as any new update is publicly verified on the blockchain.

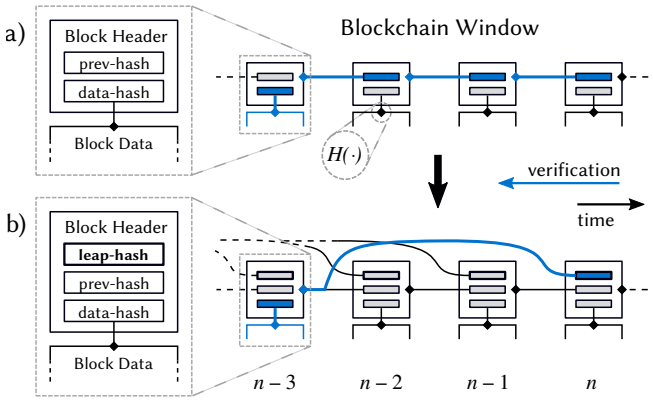

Figure 1: LeapChain Verification. We extend the conventional block structure a) that only connects a block to its direct predecessor, by a leap-hash b) allowing us to traverse the blockchain with a reduced amount of steps to verify the inclusion and integrity of block data.

However, due to the constant growth of the blockchain, downloading and processing the entire chain requires more and more resources over time. Each additional block increases the communication overhead, memory allocation, processing time, and power consumption of each device. For example, to verify the current Bitcoin blockchain, an embedded device would need to download and hash approximately 40 MB of block headers[1], already using simplified verification [8].

For the vast majority of cheap IoT devices, the blockchain length will quickly exceed their resource capabilities, rendering verification impossible. In the case of firmware updates, this might introduce severe vulnerabilities or leave devices unable to continue their intended service. It is therefore necessary to develop a verification approach for the embedded domain, in order to efficiently and safely use blockchain technology in constrained IoT environments.

1.1 Contributions

We propose *LeapChain*, a generic blockchain data structure, applicable to any kind of blockchain technology, and its corresponding algorithms for efficient verification of blocks. The concept reduces verification steps by additional backlinks as illustrated in Figure 1 and enables embedded devices to verify blockchain content using only a few kilobytes of RAM. In particular, we

- introduce a novel interlink pattern using only one additional backlink per block (Section 2),

With the support of the Technische Universität München – Institute for Advanced Study, funded by the German Excellence Initiative and the European Union Seventh Framework Programme under grant agreement n° 291763.

[1] Calculation based on 500 000 block-headers (reached Dec. 2017) of 80 bytes.

- provide methods for constructing a LeapChain that allows to verify the inclusion and integrity of a block, as well as consensus verification for Proof-of-Work blockchains using only a logarithmic amount of blocks (Section 3),
- evaluate upper bounds and show that LeapChain outperforms related approaches (Section 4) in a simulation framework and on embedded hardware (Section 5).

LeapChain guarantees deterministic and tight upper bounds for hardware requirements regarding memory and computation. This enables a safe and efficient embedded design, which is not possible using any existing state-of-the-art approach.

1.2 Blockchain Model and Terminology

For our contribution, we only consider the core components of a blockchain model that are required in the most narrow definition. Because of these minimal requirements, we will be able to demonstrate that our approach is applicable to any kind of existing blockchain implementations. We derive our notation from [9] that is also used by [10] to which we will later compare our approach.

Blockchain. The blockchain is a distributed data structure that stores a system state over time and is shared and replicated by all nodes [4]. Formally, a blockchain is an ordered chain

$$\boldsymbol{C} = \left\{ B_i \,\middle|\, i \in 1, \ldots, n \right\}, \qquad B_i \prec B_{i+1} \tag{1}$$

of n blocks B_i where n is the height of $\boldsymbol{C}$ and i the height or index of B_i. Each block confirms and reinforces the data of its preceding block by including the hash of the previous block in its own block (Figure 1a). For all hashes, we assume a single cryptographic hash function $H(\cdot)$ that outputs a hash h of κ bits:

$$H(\cdot) : \{0,1\}^* \rightarrow \{0,1\}^\kappa. \tag{2}$$

Including the previous hash in each block secures the integrity and ordering of the blocks because any change to the data of an existing block would result in changing hashes of all consecutive blocks. A block is represented by the tuple $B_i = \langle h_\mathrm{p}, h_\mathrm{d} \rangle$ with the prev-hash $h_\mathrm{p} = H(B_{i-1})$. The actual data D_i of a block can be of arbitrary structure and is bound to a block only by its hash $h_\mathrm{d} = H(D_i)$.

As we aim for a general embedded application, we do not make assumptions about the data of a block and will not provide details about any transaction system used in cryptocurrencies.

Depending on the consensus mechanism, additional information must be stored in each block. Note that this information can be stored in the block data D_i, which could then again be divided into consensus header information and another payload hash h_CP, leading to a hierarchical layer structure. However, using only two hashes as the block header is a minimal, yet sufficient specification to model any kind of blockchain application.

For simplicity, in the remainder of this paper, we refer to the block header as the block itself.

Proof of Work (PoW). Most blockchain implementations secure the construction of the chain by a cryptographic puzzle called Proof of Work (PoW). In a PoW blockchain, the block structure is extended by a nonce field $ctr \in \left[0, 2^{32}\right[$ such that $B_i = \langle h_\mathrm{p}, h_\mathrm{d}, ctr \rangle$.

Nodes with high computational power, called "Miners", verify new data D_{n+1} according to application-specific rules[2], pack them into a new unconfirmed block B_{n+1}, and iteratively try to find a ctr such that

$$B_{n+1} = \left\langle H(B_n),\, H(D_{n+1}),\, ctr \right\rangle \quad \wedge \quad H(B_{n+1}) < T \tag{3}$$

where T is the target value of the hash. Since the output of the hash is unpredictable, the only way to find such a low hash is to brute force it. T can be seen as the difficulty of the PoW puzzle: the lower T, the more tries are necessary on average.

The first miner who finds a valid hash, shares its block with the network and retrieves a reward in return. Each node in the network will validate the new block, append it to its local blockchain, and start the mining race on the next block.

Since a higher number of miners will find a valid hash in shorter time, the average time needed to find a valid hash would decrease. To keep the network stable, the difficulty is adjusted by consensus to keep the average blocktime constant.

PoW is the consensus mechanism that enables a trustless, open network where everyone can participate. A node joining the network could receive several different blockchains but will always select the one representing the most PoW as the common consensus by verifying and summing up the PoW of each block. However, nodes cannot absolutely determine when the consensus is reached. At any time, a longer valid blockchain could appear, replacing blocks of the shorter one. The likelihood that such an event changes a certain block quickly approaches zero with the number of succeeding blocks. The blockchain is secure because of the assumption that an honest majority of processing power will on average generate PoW faster than any dishonest minority [3, 9].

2 OUR LEAPCHAIN APPROACH

The main foundation of this paper is our blockchain extension that inserts additional connections with a special backlink pattern to speed up traversing the chain without weakening its integrity guarantees. Note that to traverse a conventional blockchain backwards, a node needs to iteratively verify the direct predecessor of a block using the prev-hash h_p block by block. As shown in Figure 1, we extend the conventional block structure, such that each block header stores one additional back-linking leap-hash h_l that "points" further back than just the direct predecessor – leaping over several blocks in between.

The memory overhead of this extension is minimal compared to the size of the full blocks of a blockchain. Since this additional leap-hash is part of the header that is hashed, it provides the same integrity mechanism as the prev-hash. Overhead and integrity will be further analyzed in Sections 5.1 and 5.2.

Leap-Width. The distance between the current block B_i and the block B_{i-w} that matches the leap-hash $h_\mathrm{l} = H(B_{i-w})$ is the leap-width w. With our extension, it is possible to traverse back the blockchain *either* step by step using the prev-hash h_p *or* in steps of width w using the leap-hash h_l. The first intuition would be to choose a constant w, which allows to reach any block within approximately $\frac{1}{w}$ of the steps required for the conventional blockchain. However, this would improve the amount of steps only by a linear factor, which would contradict our goal to reach any block with a sub-linear amount of steps. Since the blockchain is continuously growing in height, we need a flexible leap-width $w(i)$ depending on the current height i of a block.

Backlink Pattern. In order to achieve a logarithmic scaling, we use several leap-widths $w(i) \in W$ based on different exponents to

[2] E.g. in Bitcoin the miners validate transactions but they could also validate the signature of a new embedded device firmware.

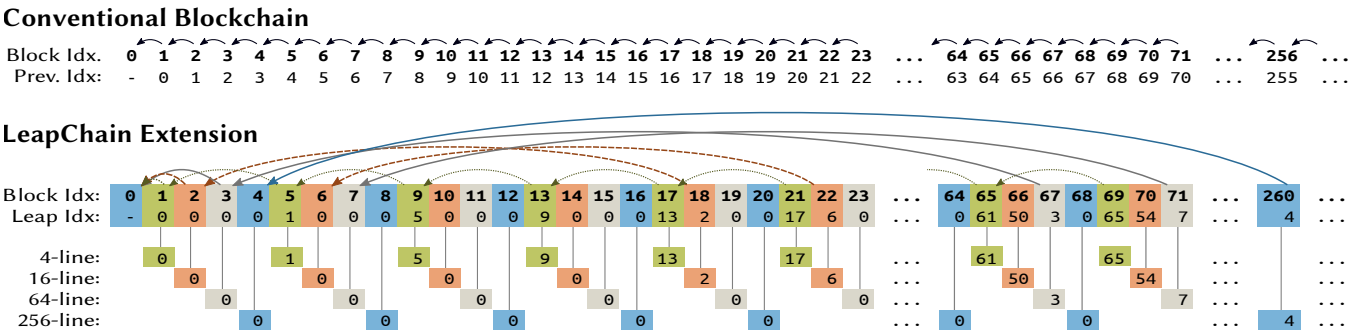

Figure 2: The LeapChain extension and its interlink pattern. In a conventional blockchain only the hash of the previous block is stored, which allows only to verify (jump to) the previous block. With the LeapChain extension, an additional leap hash is stored in each block that allows wider jumps. In this example we use a leap base of $b = 4$ resulting in 4 colored leaplines that allow us to jump back either 4, 16, 64, or 256 blocks. Each leapline can be reached within 4 consecutive blocks.

a constant base b. These leap-widths are calculated from the block height i according to

$$w(i) = \begin{cases} b^b & \text{if } i \bmod b = 0 \\ b^{(i \bmod b)} & \text{otherwise} \end{cases} \in W = \{b^1, b^2, \ldots, b^b\} \quad (4)$$

which ensures that 1) there are exactly $|W| = b$ different leap-widths, 2) all leap-widths have a single common divisor b and 3) each leap-width is b times the previous leap-width.

Beginning with $i = 1$, we assign each block of height i a leap-hash that belongs to the $w(i)$-th previous block. If $w(i)$ points to a block index $i < 0$ we set the leap-hash to the hash of the genesis block ($i = 0$). This pattern has the following four properties:

- p1) Each block leaps back to a block with the same leap-width forming a continuous *leapline*.
- p2) There are $\frac{w}{b}$ leaplines for each leap-width w.
- p3) Each block belongs to exactly one particular leapline.
- p4) Any leapline can be reached within b consecutive blocks.

If we need to jump b times on one leapline, we can also jump once on the next wider leapline instead, which leads to a logarithmic amount of steps based on the distance.

Example. Consider the case where base $b = 4$, which would result in four leap-widths $\{4^1, 4^2, 4^3, 4^4\} = \{4, 16, 64, 256\}$. The resulting leap pattern is illustrated in Figure 2. All blocks with index $i \bmod 4 = 1$, which are colored in green[3], form a single leapline with leap-width $w = 4$ (see p1). The next leap-width is $w = 16$ (orange) and between two connected orange blocks, we have $4 - 1$ other orange blocks, each belonging to one of the $\frac{16}{4} = 4$ separate leaplines of width $w = 16$ (see p2). For $w = 64$, we have $\frac{64}{4}$ different leaplines and so on. The common base $b = 4$ ensures that all leaplines jump multiple of 4 and thus a block of one leapline will never hit a block of another leapline (see p3). As a result, block 2 can be reached from block 69 in 4 steps: 68, 67, 3, 2. Of course, the maximum leap-width $w_{\max} = 256$ is not sufficient to maintain a logarithmic scaling for large blockchains. In Section 5.1 we will compare different choices of the base b and its implications.

3 VERIFICATION METHODOLOGIES

In this section, we discuss how the leap pattern can be used to verify that a certain block X and its data D_X, which could, e.g., be a firmware hash, are included in a blockchain $\boldsymbol{C}$ and were confirmed by consensus. For this verification, we first need to prove that X is indeed part of $\boldsymbol{C}$ by checking the integrity of the chain and second that the most recent blocks of $\boldsymbol{C}$ reflect the current consensus.

Note that these two verification steps are fundamentally different. For inclusion verification, we just need the hashes that are stored in the header and we will illustrate that, in this case, LeapChain is sufficient to provide a general solution with the same integrity guarantees as the full chain. For consensus verification, however, we need to consider the underlying consensus mechanism which we do not cover in general due to its diversity. We will only illustrate and analyze that LeapChain works for the common Proof-of-Work mechanism with sufficient security (detailed in Section 5.2) compared to the full chain.

In general, we can use the additional backlinks to construct a LeapChain that proves the integrity of the blockchain $\boldsymbol{C}$ between two blocks $X, Y \in \boldsymbol{C}$ with $X \prec Y$. This LeapChain efficiently traverses $\boldsymbol{C}$ from Y to X by a subset of blocks $\mathcal{L} \subseteq X, \ldots, Y$, providing evidence that X is indeed part of the same chain as Y.

In the following, we assume a network running a distributed blockchain application. We distinguish two types of nodes that run the exact same application but differ in the amount of blocks they can afford to store. First, we have a memory-constrained node – called verifier V – that wants to verify that a block X and its data D_X is part of the blockchain $\boldsymbol{C}$. V cannot afford to store $\boldsymbol{C}$ entirely. The second type is a full node – called prover P – that stores the entire blockchain $\boldsymbol{C}$. Verifier V therefore requests a "proof" $\mathcal{L}$ from a prover P in order to verify properties about the entire blockchain that it cannot verify solely from processing its partial local copy of the blockchain. Prover P constructs the proof $\mathcal{L}$ from its full local copy of $\boldsymbol{C}$ and sends it back to V, which will process $\mathcal{L}$ to decide whether the property about $\boldsymbol{C}$ holds or not.

In Section 3.1, we first discuss the case that V knows a valid, more recent block $Y \succ X$ that is part of the current consensus and needs to look up a previous block X that was pruned. This scenario applies to embedded nodes that keep a rolling window of the most recent blocks (suffix) and works for any kind of blockchain.

In Section 3.2 we discuss the case that V only knows the genesis block of the blockchain and needs to verify that block X is part of $\boldsymbol{C}$ *and* that it is accepted consensus of the network. This scenario applies to nodes that initially join a blockchain network and only works for PoW blockchains.

[3]References to the colors of leaplines are only made to support the reader but are not mandatory as leaplines can be identified by block index as well.

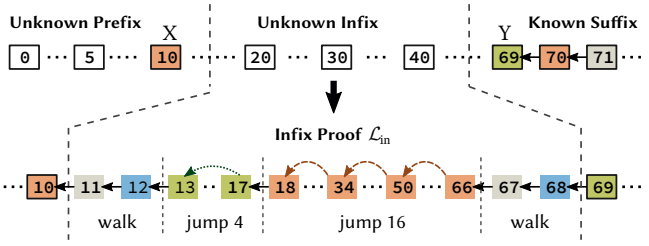

Figure 3: Infix-proof using $b = 4$. $\mathcal{L}_{\text{in}}$ traverses from $Y = 69$ to $X = 10$ (distance $\delta = 59$) using only 10 intermediate blocks. The first leap-width is $w = b^e = 4^2$ because $e = \lfloor \log_4(59) \rfloor = 2$.

3.1 Verification of Inclusion with Infix Proofs

Verifier V can verify that any block X is part of $\boldsymbol{C}$ if it knows a more recent block $Y \in \boldsymbol{C}$ with $X \prec Y$. First, V requests block X including data D_X from the network. Any node that knows D_X and the corresponding block $X = \langle h_{\text{p}}, h_{\text{l}}, h_{\text{d}} \rangle$ may send it to V as a reply. The replying node does not need to know the full chain and does not need to be trusted. To verify that D_X belongs to X, V simply checks $h_{\text{d}} = H(D_X)$. Afterwards, V requests a LeapChain $\mathcal{L}_{\text{in}}$ from P.

The prover P, which knows the complete blockchain $\boldsymbol{C}$ or at least $X \ldots Y \subseteq \boldsymbol{C}$, is able to construct a LeapChain $\mathcal{L}_{\text{in}} \subseteq X \ldots Y$ as an infix proof. This proof $\mathcal{L}_{\text{in}}$ connects the two blocks X and Y using much less blocks than the full subchain $X \ldots Y$ but provides the same integrity. Figure 3 shows an example how $\mathcal{L}$ connects X and Y. As a requirement, each block $L_j \in \mathcal{L}_{\text{in}}$ must keep the same order as in $\boldsymbol{C}$ and only link back to a previous block, such that each backlink is secured by the hash of its corresponding block.

Proof Construction. The infix proof $\mathcal{L}_{\text{in}}$ will be constructed based on the distance δ between the known block Y at height i_{h} and the target block X at height i_{t}. The block indices of $\mathcal{L}_{\text{in}}$ are determined by Algorithm 1. First, we initialize the LeapChain with $\mathcal{L}_{\text{in}} = \{Y\}$ (line 2). We iteratively calculate the exponent e of the largest leapline that fits into the remaining distance `dist` (l. 3-5), how many steps we need to walk to this leapline (l. 9), and how often we can jump on this leapline (l. 16), which is expressed by

$$\texttt{leapcount(dist, } e\texttt{)} = \left\lfloor \texttt{dist} \,/\, b^e \right\rfloor . \tag{5}$$

We append all involved blocks to $\mathcal{L}_{\text{in}}$ (l. 10-12, 17-19) and start the next iteration until we reach block X. The last iteration is likely to be $e = 0$ and $w = 1$ which means we have a "walking" phase at the end. Since both hashes h_{l} and h_{p} are included in the blocks, this phase can be seen as an iteration of leaps with $w = 1$ and does not need to be considered a special case. At the end, we prune the blocks X and Y from the proof (l. 22) because they are already known by the verifier and do not need to be transmitted.

The resulting infix proof $\mathcal{L}_{\text{in}}$ proves the integrity of $\boldsymbol{C}$ between any two blocks $X \prec Y$ and thus also proves consensus for X if Y is known to be part of the consensus. If we do not know whether Y is part of the consensus, we need to verify this with a suffix proof.

3.2 Verification of Consensus via Suffix Proofs

We will now illustrate that LeapChain is not only suitable for verifying the inclusion of blocks but can also improve the efficiency for verifying the consensus property. We will illustrate this only for the common Proof-of-Work mechanism but we are confident that our concept could be adapted to other mechanisms as well.

```
1  function leap_chain_idx(i_h, i_t):
2    leapch = list(i_h)
3    while leapch.last() > i_t do
4      dist = leapch.last() - i_t
5      e = ⌊log_b(dist)⌋
6
7      # walk to leap line
8      if (e > 0):
9        steps = (leapch.last() − e) mod b
10       foreach i in [ 1, steps ] do
11         leapch.append(leapch.last() - 1)
12       done
13     endif
14
15     # jump down leapline or directly walk (e = 0)
16     leaps = leapcount(dist, e)
17     foreach j in [ 1, leaps ] do
18       leapch.append(leapch.last() - b^e)
19     done
20   done
21
22   return leapch[1:-1]  # prune i_h and i_t
```

Algorithm 1: Pseudocode for LeapChain construction.

If we consider a PoW blockchain, V can request two LeapChains in order to verify that a block X is part of $\boldsymbol{C}$ *and* that X is common consensus with a certain probability. In this case, V does not need to know any recent block $Y \succ X$ of the blockchain but only the hash $H(B_0)$ of the genesis block B_0 or $H(B_C)$ of another commonly known checkpoint block B_C where $B_0 \preceq B_C \prec X$.

The PoW consensus requires that each block hash needs to be below the target value T, which is determined by a certain difficulty. This difficulty is usually calculated from previous blocks using timestamps. Since we will leap blocks, V cannot determine the difficulty for the blocks of a LeapChain because it does not know the difficulty of the intermediate blocks. However, this problem could be solved by including the current difficulty in the block data such that miners can extract and verify the difficulty from each single block.

In the following, we choose another option and assume a constant difficulty, which means that each block hash needs to be below the same target value T that will never change. Even though practical blockchain applications based on PoW with a flexible number of Miners require a variable difficulty, we use this simplification because it is used by [9], the main theoretical framework for PoW, and thus also used by [8, 10], to which we later want to compare our work using similar assumptions. As pointed out by [9] and [8], analyzing a constant difficulty is sufficient, because accounting for variable difficulty can be easily achieved by counting blocks proportional to their difficulty. We will further discuss this matter in Section 5.2.

We store the additional fields, required for the PoW consensus, in the block data, such that the size of our block header does not change and we do not need to consider any differences between our minimal model for a general blockchain and the special PoW case.

First, V verifies that X is connected to B_C and thus part of the same blockchain as B_C by requesting an infix proof $\mathcal{L}_{\text{in}}$ from a prover P using the method described in Section 3.1.

However, till now $\mathcal{L}_{\text{in}}$ is not sufficiently secure to proof that X is part of the current consensus. As shown in Figure 4, an adversary could reuse the existing prefix $B_0 \ldots B_{i-1}$ and only mine a block X' that includes some fake data and valid backlinks to the existing prefix. Therefore, V requests a second proof $\mathcal{L}_{\text{su}}$ that puts a certain amount of cumulative PoW on top of X as evidence that X was confirmed by several succeeding blocks and is indeed part of the consensus.

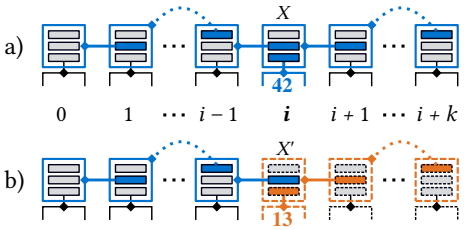

Figure 4: a) Valid prefix and suffix proofs of the block X (blue) with a data value of 42. b) In case no more recent block $Y > X$ is known, an adversary could try to provide proofs for a fake block X' with a different value 13. The adversary could reuse the prefix but needs to mine a suffix proof from block i to $i + k$ (dashed orange).

Since each block needs to satisfy the same target value T, each block contributes the same amount of required PoW. We therefore construct a proof $\mathcal{L}_{\text{su}}$ of certain length m confirming block X by m succeeding blocks which inherit m times the PoW of a single block.

An adversary would now need to mine a fake block X' and all m blocks of $\mathcal{L}_{\text{su}}$ to convince V that X' is part of the consensus, which is infeasible.

In order to calculate $\mathcal{L}_{\text{su}}$, we adjust the `leapcount` function of Algorithm 1 such that we only leap blocks if there are enough blocks left that will increase the cumulative PoW. Therefore, after each block we add to $\mathcal{L}_{\text{su}}$, we need to have $m - |\mathcal{L}_{\text{su}}|$ blocks left that we could "walk" block by block to produce the required length. More specifically, the amount of leaps on each leapline b^e with $e > 0$ should satisfy

$$\texttt{dist} - \texttt{leaps} \cdot b^e > (m - |\mathcal{L}_{\text{su}}|) - \texttt{leaps} \tag{6}$$

which can be solved for `leaps` and leads to

$$\texttt{leapcount}(\texttt{dist},\ e) = \begin{cases} \left\lfloor \frac{\texttt{dist} - m + |\texttt{leapch}|}{b^e - 1} \right\rfloor & e > 0 \\ \texttt{dist} & e = 0 \end{cases} \tag{7}$$

for Algorithm 1. If $m > \delta$, the algorithm now adds all δ indices to `leapch` and at least m indices otherwise.

If V receives two competing PoW LeapChains $\mathcal{L}_{\text{su}}1$ and $\mathcal{L}_{\text{su}}2$ with different blocks (see Figure 5), we have two possibilities: 1) take the one with more cumulative PoW or 2) challenge the provers by requesting another specific LeapChain. Both cases together allow us to select an arbitrary level of security.

Challenge-Response. As mentioned by [11] as a preferable property, our scheme allows V to challenge P if the *index* of a block Y is known with $X < Y \leq B_{\text{last}}$. V can then calculate one out of several valid LeapChains and request this specific chain as proof from P, which makes it more difficult for P to create a fake chain. The number of possible LeapChains increases with distance δ, however, determining the exact value is out of scope of this paper.

Note that a block does not need to store its index because a verifier can always determine all indices as long as one block index (e.g. 0 for genesis) of the LeapChain is known. This property results from the deterministic leap-line assignment based on the block index.

4 RELATED WORK

Most of the related work on verification of block inclusion is focused on Simplified Payment Verification (SPV) in cryptocurrencies which was already mentioned in the original Bitcoin whitepaper [3]. A light node that wants to verify that a certain transaction is accepted, only keeps the block headers without block data of the entire blockchain

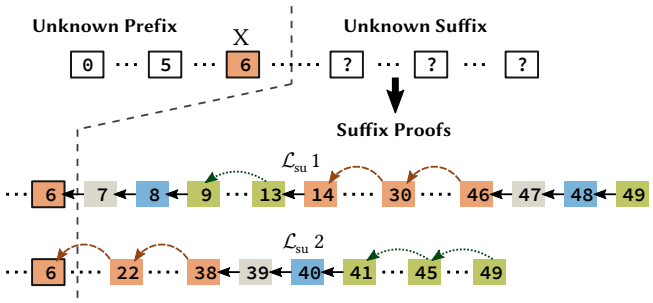

Figure 5: Suffix-proof for consensus verification of block $X = 6$. Here, two valid proofs $\mathcal{L}_{\text{su}}1$ and $\mathcal{L}_{\text{su}}2$ are shown that connect X to a recent block $B_n = 49$, putting 43 additional blocks on top of X. However, $\mathcal{L}_{\text{su}}1$ is longer and proves more cumulative PoW.

and verifies that the transaction belongs to a certain block header. While block headers are much smaller than the full blocks, this "naive SPV" approach scales linear with the length of the blockchain and hence is not feasible on highly constrained devices.

Sidechain SPV. Another idea sketched in [11] suggests that each block creates additional backlinks to *every* previous block using a Merkle tree and including the root hash in the block header. Skipping back is only allowed if the actual Proof-of-Work (PoW) of the current block exceeds the cumulative PoW of all blocks in between, resulting in an average proof length of $\log_2(\delta)$. However, [11] does not evaluate the proof size including the huge amount of additional hashes of the backlinks which would exceed the memory capabilities of embedded nodes.

Skipchains. The approach in [6] proposes a Skipchain $\mathcal{S}_k^l$ where each block stores l backlinks to the k^i-th previous blocks for $0 \leq i \leq l - 1$. If $k < 0$, this corresponds to a probabilistic scheme where the number of skipped blocks equals the number of successful Bernoulli trials with probability k. Considering blocks with high PoW, this could be modeled as a $\mathcal{S}_2^l$ for ideal PoW distribution. However, any chosen l is fixed and finite, resulting either in a very limited logarithmic scaling or a large amount of backlinks, increasing the proof size. In contrast to Skipchain, our LeapChain approach only requires a single backlink to achieve logarithmic scaling via its special pattern, which significantly reduces the memory overhead of each block in a proof.

Proofs of Proof of Work (PoPoW). The most optimized state-of-the-art approach is the PoPoW scheme [8, 10], which we use as a benchmark. In this scheme, each block stores a vector of backlinks only to those blocks that randomly happen to inherit a higher (or "deeper") PoW than required. The approach determines a "depth" μ for each hash $H(B)$, such that $H(B) < 2^{-\mu} \cdot T$ and the j-th element of the vector stores the hash of the nearest previous block that satisfies $j \leq \mu$. This means that on average, the j-th backlink points to every 2^j-th previous block. A verifier can request a proof-chain connected by the backlinks of depth j, such that each block of the proof represents at least 2^j times the minimal PoW required for a block.

While PoPoW relies on probabilistic assumption on how often low hashes occur, LeapChain is fully deterministic. The proof size of PoPoW [8] scales with $\log_2 \log_2 |\boldsymbol{C}| \cdot \log_2(\delta)$, while LeapChain scales independent of the chain length $|\boldsymbol{C}|$ with only $b \cdot \log_b(\delta)$ by using only a single backlink.

b	4	6	8	12	16
$\delta_{\log}$	1029	279945	134 e6	107 e12	295 e18
$w_{\max} = \alpha^{-1}$	256	46656	16.8 e6	8.92 e12	18.4 e18
$\lvert\mathcal{L}_{\text{in}}(\delta_{\log})\rvert$	20	44	76	164	284
$\text{size}(\mathcal{L}_{\text{in}}(\delta_{\log}))$	1.92 kB	4.22 kB	7.30 kB	15.7 kB	27.3 kB

Table 1: LeapChain parameters for several bases b. The maximum size of $\mathcal{L}_{\text{in}}$ applies when using a hash of 32 bytes (e.g. SHA-256).

5 EVALUATION

For our approach, we first introduce upper bounds to resource requirements, which enable to select an appropriate embedded hardware platform. Afterwards, we experimentally compare LeapChain against related work in a simulation illustrating overall performance gains and on embedded hardware to underpin LeapChain's feasibility.

5.1 Analytical Discussion

Maximum Proof Size. As shown in Figure 3, a LeapChain proof $\mathcal{L}_{\text{in}}$ can be divided into 3 parts: 1) an initial walk part to reach the first leapline, 2) a jump part using several leaplines until we 3) walk again to reach the target block. We estimate the maximum size of $\mathcal{L}_{\text{in}}$ based on the worst-case length for each of these 3 parts.

1) The desired leapline can be reached in a maximum of $b-1$ steps, because the pattern repeats after b blocks. 2) Each leapline is used $b-1$ times in the worst case, because if we need to jump b times, we could jump once using the next higher leapline. The highest exponent of a leapline we need to consider is $e = \lfloor \log_b(\delta) \rfloor$. Thus, each of the e leaplines adds $b-1$ blocks and between the leaplines we need one additional step block to reach the next lower leapline. This results in $\lfloor \log_b(\delta) \rfloor \cdot (b-1+1) - 1$ maximum blocks for the jump part. 3) When using all leaplines, the last leap-width is $w = b$. Thus, we need at most $b-1$ blocks to walk to the target block, but since the target block hash is already included in the second last block, we only need $b-2$ blocks. Combining these results, the proof length is bounded by

$$|\mathcal{L}_{\text{in}}(\delta)| \;\leq\; b \cdot \lfloor \log_b(\delta) \rfloor + 2b - 4, \quad \delta \leq \delta_{\log} \tag{8}$$

with the corresponding proof size of $\text{sizeof}(\mathcal{L}_{\text{in}}) = \text{sizeof}(B) \cdot |\mathcal{L}_{\text{in}}|$. When all b leaplines are used, we reach $\left|\mathcal{L}_{\text{in}}(\delta_{\log})\right| \leq b^2 + 2b - 4$ at the maximum logarithmic distance $\delta_{\log}$.

Logarithmic Distance. Since the maximum leap-width $w_{\max} = b^b$ is a finite constant, the proof length will only scale logarithmic until a distance

$$\delta_{\log} = b^{b+1} + 2b - 3 \tag{9}$$

and scale linearly with a very low slope $\alpha = b^{-b}$ afterwards. $\delta_{\log}$ is derived from the worst cases for each leapline (see previous paragraph) with the difference that the distance includes the target block, resulting in $b-1$ instead of $b-2$ for the last walking phase.

Table 1 provides several parameter sets to illustrate which maximum distance $\delta_{\log}$ can be verified in $\log_b$ scaling and the corresponding bound of steps $\left|\mathcal{L}_{\text{in}}(\delta_{\log})\right|$. Note that these are upper bounds and in practice a more efficient proof can be found for $\delta \approx \delta_{\log}$. After $\delta_{\log}$ is reached, $|\mathcal{L}_{\text{in}}|$ increases linearly by 1 every $\alpha^{-1} = b^b$ additional blocks.

Overhead. Since we store a single additional hash in each block, the memory overhead over a conventional blockchain corresponds to the size of the hash generated by the used hash function. For Bitcoin, which uses SHA-256 and a full block size of 1 MB, the memory overhead for nodes storing the full chain would be 32 B/1 MB = 0.0032%. Considering only block headers, the overhead would be 32 B/80 B = 40%, which is compensated as soon as $\delta \geq 1.40 \cdot |\mathcal{L}_{\text{in}}|$.

The computational overhead for full nodes is negligible as each leap-hash belongs to a block for which the hash is already known.

5.2 Security

We analyze security based on the difficulty for an adversary to provide a fake proof. Our verification method relies on two different mechanisms, the infix proof for proving the inclusion of a block in the blockchain and a suffix proof for proving that a block is accepted by consensus. The security of the infix proof relies on the integrity of the chain of hashes, while the security of the suffix proof relies on the security of the underlying consensus mechanism, which we will discuss for Proof of Work.

Integrity Guarantees. The security of an infix proof $\mathcal{L}_{\text{in}}$ relies solely on the preimage resistance of the hash function H. In order to change any block B_i that existed before a valid and known block B_n, an adversary would need to successfully run a preimage attack on the hash function H. Note that a preimage attack is much more difficult than a collision attack, which only requires to find any two identical hash values and not a specific one. Since the hash of B_i is included in the next block B_{i+1}, the adversary would need to find an alternative block B_i' with the exact same hash as B_i, thus satisfying $H(B_i) = H(B_i)$. For an ideal hash function of κ bits, this requires 2^κ tries on average, which for 256 bit is infeasible. For every block in $\mathcal{L}_{\text{in}}$ either the prev-hash or the leap-hash stores the hash of the previous block and both hash values are stored in the block header which also gets hashed to obtain the current block hash. The working principle of the prev-hash and the leap-hash is the same and thus the integrity of every block, whose hash is included by one of the two hash fields within the infix proof, is guaranteed. As a result, any valid infix proof provides the same security as the full chain regarding its integrity guarantees.

Consensus Guarantees. In the case of a PoW blockchain, the security of the suffix proof $\mathcal{L}_{\text{su}}$ relies on its cumulative PoW that an adversary would need to spend to construct a fake proof inheriting the same PoW. The cumulative PoW is expressed as multiple of the *minimum required* PoW (= 1 PoW) to mine a single valid block with $H(B_i) < T$. In our scenario, T is assumed to be constant and therefore every block inherits the same PoW on average. As a result, the cumulative PoW of $\mathcal{L}_{\text{su}}$ can be estimated as the length $|\mathcal{L}_{\text{su}}|$ times the average PoW of any block $B_i \in \boldsymbol{C}$.

Since we place the required *nonce* field in the block-data, a miner would need to calculate two hashes – the data-hash and the block header-hash – for each attempt to solve the PoW, which will increase the computational effort. However, the purpose of PoW is to perform a certain amount of work involving billions of hash operations, so the double hashing can simply be adjusted by the difficulty. As already mentioned, our approach can be easily adapted to variable difficulty by storing the difficulty together with the nonce in the block data. As shown by [9] and [8], the consensus is then determined by counting the PoW of a block proportional to its difficulty.

Although $\mathcal{L}_{\text{su}}$ provides in principle less security compared to the full chain due to a shorter chain length, LeapChain provides a sufficiently high and more constant security. First, the distribution of the consensus guarantees in the full chain is not constant but

increases from the most recent block to the genesis block, making recent blocks less secure than older blocks. Second, the overall security of every block infinitely increases with each new block that is appended to the blockchain, securing already sufficiently secured blocks by an increasingly superfluous amount of cumulative PoW on top of them. Therefore, we use the fact that LeapChain is more efficient than the full chain, in the sense that it allows to freely choose a flexible security parameter $m = |\mathcal{L}_{\text{su}}|$, in order to ensure a constant security level for the suffix proof. For the m most recent blocks, LeapChain provides the same security as the full chain because, in this case, the suffix proof $\mathcal{L}_{\text{su}}$ equals the suffix of the full chain. For blocks that are older than the m most recent blocks in the full chain, the parameter m can be chosen between the shortest possible proof length $|\mathcal{L}_{\text{in}}|$ and the distance δ.

As an example, we assume an attacker P' with 10% of all hashing power that wants to convince a verifier V of a fake block X' and we set $m = 50$. If the latest block is within the first 50 blocks after X', the attacker would need to mine all blocks from X' to the latest block faster than the honest majority and the likelihood of success can be expressed by the equation from [3]:

$$\text{Succ}(z, q) = 1 - \sum_{k=0}^{z} \frac{(\lambda)^k e^{\lambda}}{k!} \left(1 - \left(\frac{q}{1-q}\right)^{z-k}\right) \tag{10}$$

Here, z is the amount of blocks to mine and $\lambda = z\frac{q}{1-q}$ where q equals the hashing power of the attacker. For 50 blocks, this results in $\text{Succ}(50, 0.1) = 7.3 \times 10^{-17}$.

For comparison, most Bitcoin applications require only 6 most recent blocks to trust the consensus as settled [1, 8], which results in an attack success probability of $\text{Succ}(6, 0.1) = 2.4 \times 10^{-4}$. Note that each attack attempt demands high computational effort, so even if this probability seems relatively high, an attacker would need to spend a great amount of money for every single attack attempt. The security of a blockchain in general relies on the fact that attacks cause huge financial damage to an attacker with overwhelming probability. However, by requiring 50 blocks, $\mathcal{L}_{\text{su}}$ exceeds this basic security and approaches the very small success probability of $\text{Succ}(50, 0.1)$. For blocks older than the first 50 blocks an attacker has more time to mine fake blocks but still needs to continuously mine blocks as every suffix proofs contains recent blocks and the blocks chosen for the suffix proof are changing. As an improvement, one could also require that each suffix proof always contains the b most recent blocks in a row to ensure at very least a security of $\text{Succ}(8, 0.1) = 1.7 \times 10^{-5}$.

Even in the unlikely case that an attacker would manage to provide a fake proof, the verifier V which would then receive proofs with different versions of block X, could still challenge the provers, which would require the attacker to find another fake proof within a few seconds, which is infeasible.

Overall, LeapChain offers a controllable, constant, and high security and we will evaluate its embedded performance for an already very high security level of $m = 50$.

5.3 Simulation

We simulated our LeapChain approach in Python on the block headers of randomly generated blockchains using our block structure. We included random data-hashes as the block data is not relevant for our proof construction. For the hash function $H(\cdot)$ we have chosen SHA-256, which outputs a hash of 32 bytes. As shown by [12], the computation time of SHA-256 is linear to its input size. Since hashing is the computationally most intensive operation in the verification, the overall computational effort is proportional to the total amount of data bytes that are hashed.

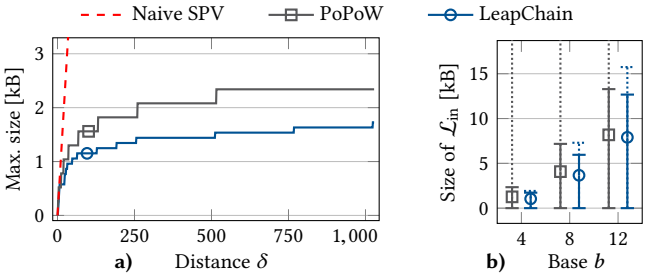

Figure 6: Simulated infix proof sizes when using ideal PoW distribution. a) Maximum proof size for $b = 4$ and $|C| = 1029$ over δ. PoPoW and LeapChain scale logarithmic compared to the naive SPV. b) Average (marked with □/○), min./max. (−), and analytical upper bound (···) of proof size over all $\delta = [1, \delta_{\log}]$ depending on b.

For comparison, we implemented the Proofs-of-Proof-of-Work (PoPoW) scheme [10] with the interlink vector stored directly in the block header using our block structure. We also applied suggested optimizations such as storing the vector in a Merkle tree, which requires only $\log \log |\boldsymbol{C}|$ hashes to be included in the proof for each block. When the prev-hash is used, the interlink vector was omitted to further reduce the proof size. For proof of inclusion, we iteratively selected the interlink with the longest jump that approaches the target block. For the PoW verification, we constructed proofs for all possible interlink depths and then selected the shortest proof that provides the required cumulative PoW.

We measured the following three metrics that are relevant for the verification on an embedded device: 1) the size of a proof, that needs to be transmitted and processed, 2) the computational effort to verify a proof in hashed bytes, and 3) for a suffix proof the security as its cumulative PoW.

Results. For embedded devices, the proof size is most important because it corresponds to the amount of data that needs to be received and processed. Figure 6 shows the measured size and analytical bounds of infix proofs for several bases b. In contrast to the naive SPV, the maximum proof size of PoPoW and LeapChain scales logarithmic, leading to small proof sizes.

For this measurement, we used ideal PoW distribution for PoPoW, where exactly every 2^{μ}-th block has a hash $< T/2^{\mu}$ and is included in the interlink. Even in this best case for PoPoW, LeapChain provides 11% smaller proof sizes on average.

When using a random PoW distribution, as it occurs in real blockchains, the performance gain of LeapChain is even higher for infix proofs (Figure 7a). Since PoPoW relies on probabilistic assumptions about how often low hashes will occur, proof sizes are subject to a high variance, which leads to peaks of large proof sizes. In the absolute worst case, PoPoW would need to included every block leading to the same proof size as the naive SPV.

Regarding the computational effort, LeapChain also significantly outperforms PoPoW. For LeapChain, the hashed data is equal to the proof size but PoPoW requires additional hash operations for looking up the interlink in the Merkle tree.

In the case of suffix proofs (Figure 7b), which require a minimum cumulative PoW, PoPoW comes closer to the size of LeapChain on average but still produces a high amount of peaks that are two to three times larger. Although PoPoW links to blocks that inherit a higher PoW, only the *required* PoW to convince a verifier increases

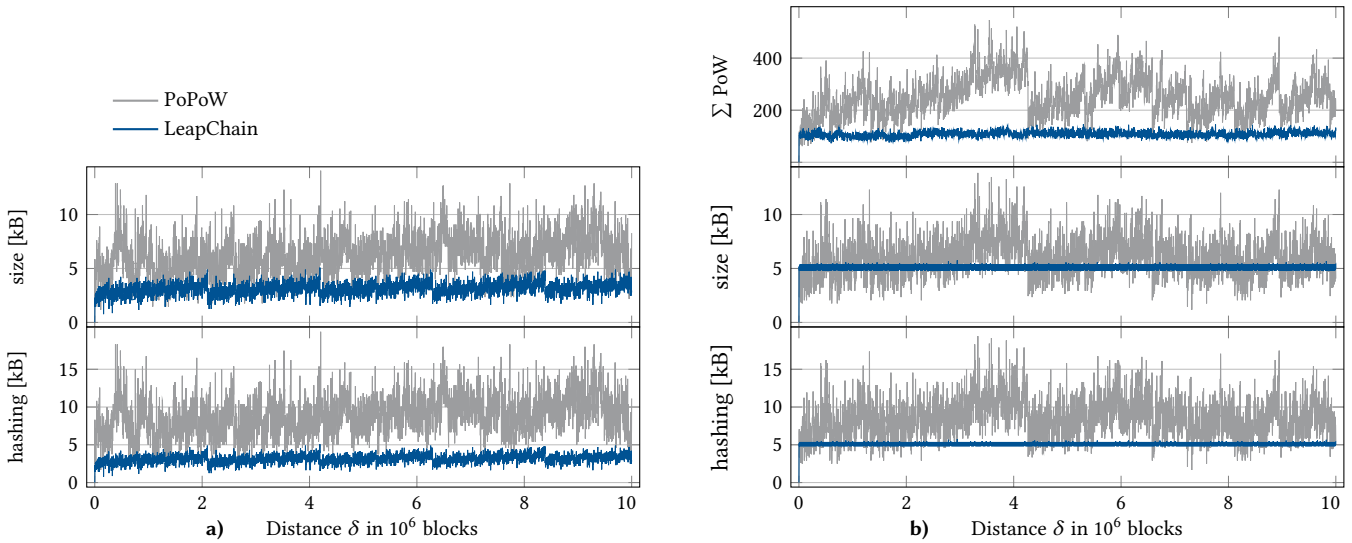

Figure 7: Simulated proofs using $|C| = 10 \times 10^6$, $b = 8$ and random PoW distribution. a) Infix proofs. b) Suffix proofs with required cumulative PoW $\sum \text{PoW} \geq 50$. LeapChain outperforms PoPoW in both scenarios by achieving lower average and maximum values and a smaller variance for proof size and computational effort.

the proof security. A higher cumulative PoW is only of advantage when several competing proofs need to be compared, in which case LeapChain can also challenge the provers. We have chosen a high required cumulative PoW of $\sum \text{PoW} \geq 50$, although Bitcoin already considers a block as part of the consensus if there is a suffix of 6 recent blocks with $\sum \text{PoW} \geq 6$ [1, 8].

Overall, the high variance of PoPoW proofs poses a critical uncertainty for embedded devices which cannot afford to provide large resource reserves [2]. By contrast, LeapChain is fully deterministic and guarantees tight upper bounds of proof size and computational effort, which are close to measured maximum values.

5.4 Running on ESP32 Chipset

In order to compare the approaches on a real embedded IoT platform, we tested them on the ESP32 chipset (2×240 MHz) using MicroPython v1.8.6 with 57 kB available SRAM on the WiPy 2.0 board. For the implementation we used SHA-256 of the `uhashlib`. We tested 3602 infix proofs with $|C| = 10 \times 10^6$ and $\delta \in [1, |C|]$ in steps of $\Delta\delta = 2777$. Our LeapChain approach with $b = 8$ verified all proofs within an average time of 196 ms and a maximum time of 275 ms. PoPoW with ideal PoW distribution (best case) was about 14% slower on average (224 ms) and 29% slower on the longest proof (355 ms), which is in line with our simulation results. Considering its worst-case behavior on random PoW distribution, PoPoW ran out of memory and crashed on the first proof after 133 blocks. Although an optimized C implementation would further reduce hardware requirements, our implementation already demonstrates LeapChain's feasibility on constrained IoT platforms and shows that PoPoW inherits the risk to fail completely.

6 CONCLUSION

Our approach enables embedded IoT devices to verify data integrity and consensus on a blockchain within milliseconds. The LeapChain proof size scales logarithmically with $b \log_b(\delta)$ to the block distance δ while maintaining the same integrity guarantees as the full chain. For consensus verification of PoW blockchains, LeapChain provides a dynamically adjustable security parameter m and already our selection of $m = 50$ significantly exceeds the security requirements of most existing blockchain applications.

Setting $b = 8$, we could verify the inclusion of any block out of 134 million blocks using at most 76 block headers and outperform existing approaches by at least 11% regarding average proof size. While existing approaches could exceed several megabytes of proof size in the worst case, LeapChain guarantees a deterministic and tight upper bound of 7.3 kB, enabling efficient and safe blockchain applications on embedded IoT devices.

REFERENCES

[1] Marko Vukolić. 2016. *The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication.* Springer International Publishing, Cham, 112–125.
[2] Shreyas Sen. 2016. Invited: Context-aware energy-efficient communication for IoT sensor nodes. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, New York, NY, USA, Article 67, 6 pages.
[3] Satoshi Nakomoto. 2008. Bitcoin: A peer-to-peer electronic cash system.
[4] Konstantinos Christidis and Michael Devetsikiotis. 2016. Blockchains and Smart Contracts for the Internet of Things. *IEEE Access* 4, 2292–2303.
[5] Boohyung Lee and Jong-Hyouk Lee. 2017. Blockchain-based secure firmware update for embedded devices in an Internet of Things environment. *The Journal of Supercomputing* 73, 3, 1152–1167.
[6] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. 2017. CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds. In *26th USENIX Security Symposium (USENIX Security 17)*. 1271–1287.
[7] Marco Steger, Ali Dorri, Salil S. Kanhere, Kay Römer, Raja Jurdak, and Michael Karner. 2018. *Secure Wireless Automotive Software Updates Using Blockchains: A Proof of Concept.* Springer International Publishing, Cham, 137–149.
[8] Aggelos Kiayias, Andrew Miller, and Dionysis Zindros. 2017. Non-interactive proofs of proof-of-work. In *Cryptology ePrint Archive*.
[9] Juan A Garay, Aggelos Kiayias, and Nikos Leonardos. 2015. The Bitcoin Backbone Protocol: Analysis and Applications. In *EUROCRYPT (2)*. Springer Berlin Heidelberg, 281–310.
[10] Aggelos Kiayias, Nikolaos Lamprou, and Aikaterini-panagiota Stouka. 2016. Proofs of Proofs of Work with Sublinear Complexity. In *Financial Cryptography and Data Security*. Springer Berlin Heidelberg, Christ Church, Barbados, 61–78.
[11] Adam Back, Matt Corallo, Luke Dashjr, Mark Friedenbach, Gregory Maxwell, Andrew Miller, Andrew Poelstra, Jorge Timón, and Pieter Wuille. 2014. Enabling Blockchain Innovations with Pegged Sidechains. (22 10 2014).
[12] Sanat Ghoshal and Goutam Paul. 2016. *Exploiting Block-Chain Data Structure for Auditorless Auditing on Cloud Data.* Springer International Publishing, 359–371.