# Bitforest: a Portable and Efficient Blockchain-Based Naming System

Yuhao Dong
University of Waterloo
yd2dong@uwaterloo.ca

Woojung Kim
University of Waterloo
w3kim@uwaterloo.ca

Raouf Boutaba
University of Waterloo
rboutaba@uwaterloo.ca

***Abstract*—Public key infrastructures (PKIs), or more generally secure naming systems, lie at the foundation of the security of any communication system. Without a trustworthy binding between user-facing names, such as domain names, and cryptographic identities, such as public keys, all security guarantees against active attackers come crashing down like a house of cards.**

**Blockchains such as Bitcoin, by offering a decentralized yet secure public ledger, show promise as the root of trust for naming systems with no central trusted parties, greatly increasing their security compared to traditional centralized PKIs. Yet blockchain PKIs such as Namecoin and Blockstack tend to significantly sacrifice scalability and flexibility in pursuit of decentralization, hindering large-scale deployability on the Internet.**

**We propose Bitforest, a secure naming system with an architecture combining a centralized yet only partially trusted name server with efficiently queryable verification data embedded in a novel data structure inside a cryptocurrency blockchain. Bitforest achieves decentralized trust and security as strong as existing blockchain-based naming systems while retaining most of the flexibility and performance of centralized PKIs, allowing fully-validating thin clients to look up and verify name bindings with comparable efficiency to traditional systems. We use both numerical simulation and real-world experiments to evaluate the performance of Bitforest compared with other naming systems, both centralized and blockchain-based, showing that its performance goals are indeed achieved.**

## I. Introduction

Cryptographic protocols such as TLS, which provide secure communications over insecure networks such as the Internet, have gained extremely pervasive deployment. Yet the security of these protocols relies ultimately on one thing: a secure *public key infrastructure*. Application-level names must be somehow bound to cryptographic identities, such as public keys, in a trustworthy way; otherwise, security against active man-in-the-middle attacks cannot be achieved.

Unfortunately, building a secure PKI has proved to be quite difficult. Traditional systems, like the hierarchical PKI used in TLS and S/MIME, attempt to achieve their security properties by introducing trusted third parties, such as certificate authorities (CAs) or key servers. Centralization, however, leads to brittle security, as compromised or incompetent trusted parties can undermine the security of entire namespaces [1], [2].

Blockchains, public append-only ledgers that are unforgeable yet fully decentralized, offer a promising alternative to centralized trust. The first blockchain, Bitcoin [3], was conceived only as a financial cryptocurrency, but its first descendant, Namecoin [4], pioneered the idea of building a secure naming system by encoding name-value pairs inside a blockchain. Several newer blockchain naming system designs, such as Certcoin [5], follow the same general design.

However, though Namecoin-like designs bring significant security improvements, they also face many new challenges. Among other issues, all nodes in the network must synchronize a local copy of the blockchain, so anybody wishing to look up names in a secure fashion faces large, linearly-increasing storage costs. Additionally, without a large userbase, blockchains are vulnerable to attacks which compromise their security guarantees.

Newer blockchain-based PKIs, most recently and successfully Blockstack [6], do attempt to mitigate these issues. However, although Blockstack makes it easier to deploy new features and reduces the amount of data that needs to be replicated to all participants by moving most of the data away from the underlying blockchain, it still fails to eliminate the requirement for verifying large amounts of blockchain data, and continues to be much less flexible in enforcing rules for namespaces compared to centralized solutions.

These issues with existing blockchain-based distributed PKIs motivate us to build a new system, Bitforest, sidestepping the common pitfalls of "pure" blockchain PKIs by using a hybrid architecture combining a public blockchain with a minimally-trusted centralized service. A novel data structure encodes information crucial to the integrity of the PKI in an arbitrary cryptocurrency blockchain, allowing us to inherit the security of an underlying blockchain while retaining much of the properties of traditional PKIs including fast name lookups, low storage requirements, policy flexibility, and performance. We believe that by escaping the apparent hard tradeoff between decentralized, blockchain-anchored trust and high performance, Bitforest makes it significantly more practical to deploy a blockchain-based PKI.

## II. Design

In this section, we describe Bitforest's design principles, its overall architecture, and how its goals are achieved by its design.

### A. Design principles

Based on the experiences of previous blockchain-based PKIs, we designed Bitforest in accordance with the following three main principles:

**Blockchain portability:** Bitforest is designed to not rely on any particular blockchain. Any cryptocurrency blockchain with its security based on the principle of spending unspent transaction

978-3-903176-14-0 

outputs (UTXOs) at most once (i.e. "UTXO-based blockchains") can be easily plugged in. This allows individual deployments of Bitforest to adapt to whatever blockchain fits the application the best — for example, Bitcoin can be used for applications needing high security with less concern over update throughput, while Dashcoin can be used for scenarios needing high performance.

**Centralized administration:** We want it to be possible to create centrally-managed namespaces, such as directories of employees belonging to a certain organization. This is in contrast to fully distributed blockchain-based PKIs such as Namecoin and Blockstack, which are entirely permissionless and thus are often vulnerable to name-squatting and other abuse — the vast majority of name-value bindings in Namecoin, for example, are in fact spam containing no useful information [7]. Furthermore a centralized provider can be used to increase performance by indexing the blockchain and maintaining in-blockchain data structures, as existing "hybrid" blockchain naming systems, most notably EthIKS [8], demonstrate.

**Decentralized identity retention:** On the other hand, we need to avoid trusting the central administrator when enforcing a bottom-line of security — *identity retention*. That is, even with a malicious administrator, it should not be possible to make changes to name-value bindings without authorization from the owner of the name, or for anyone to be fooled into obtaining an incorrect binding. Robustly decentralized identity retention is in fact perhaps the *raison d'être* of blockchain-based PKIs, as other systems almost always require at least partially trusting some centralized entity.

### B. Architectural overview

Bitforest uses a client-server architecture, where *namespace administrators*, or NAs, administer *namespaces*; *clients* then look up names in a certain namespace by querying information published by a particular NA. Instead of imposing a global infrastructure of NAs, Bitforest allows developers to set up application-specific NA infrastructures, similar to how traditional PKIs may have application-specific root CAs.

In Bitforest, names in a certain namespace are mapped by NAs to unique *indices* in a deterministic, verifiable way. Clients then use an *index tree*, a novel insert-only dictionary data structure embedded inside the blockchain, to map these indices to an append-only list of *operation hashes* — secure hashes of each element of the *operation log*, an NA-provided history of all values ever bound to the name. Each entry in the operation log also contains signatures by a cryptographic identity declared in the previous entry; in effect, updates to the name describe who is authorized to append further updates. The last entry in the operation log defines the current binding of the name.

We double-spending prevention, available in all UTXO-based blockchains, to secure the insert-only property of Bitforest's index tree, an idea pioneered by Catena [9], a blockchain-embedded append-only log operating on a similar principle. This allows Bitforest to achieve strong identity retention backed by a fundamental security guarantee of its underlying blockchain while retaining efficient and secure lookup of names without prohibitive bootstrapping cost. Furthermore, storing only indices and hashes in the in-

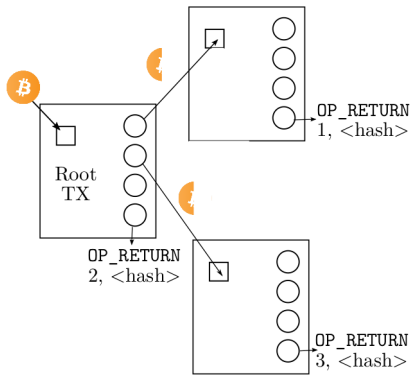



Fig. 1: Example of an index tree with indices 1,2,3

blockchain data structure gives freedom to the NA adminsitrator to enforce non-cryptographic constraints, such as authentication and access control. Updates and queries must involve the NA, as only the NA can compute indices from names or provide the actual operations whose hashes are stored in the index tree.

### C. Basic structure of the index tree

Bitforest's index tree is essentially an insert-only binary search tree (BST) consisting of transactions in the underlying blockchain, mapping indices, 256-bit keys corresponding uniquely to names (see II-E), to 256-bit cryptographic hashes of every entry in an operation log. Every node in the index tree is a transaction, where:

- The first input, for all nodes except the root node, spends either the 1st or 2nd output of the node's parent in the index tree.
- The first four outputs are, in order,
  - A spendable output to be spent by the left child
  - A spendable output to be spent by the right child
  - A spendable output to be spent by the *update chain*
  - An unspendable output (for example, using `OP_RETURN` in Bitcoin) storing an index and the 256-bit hash of the first operation of the name associated with the index

(The "update chain" that uses the third output of each node stores subsequent updates to the name, and is discussed in II-D) All the transactions in the index tree represent nodes in a BST, sorted by index, with smaller indices to the left and larger indices to the right; children are related to their parents by spending a particular output in their parent. Money flows down the tree from the root transaction and is used to cover transaction costs; additional funds can be introduced at any point through transaction inputs other than the first one. Fig. 1 gives an illustration of a small index tree. Inserting a new index-hash pair to the tree is done in a straightforward manner — a new node is broadcast to the blockchain, spending the output that would connect it to the tree in such a way that searching for the index would find the new node. The NA maintains control over inserts to the index tree by sending the spendable outputs to addresses whose private keys only the NA knows.

An important property is that every node can have at most one left node, and at most one right node, confirmed in the blockchain. This is due to double-spending prevention — each transaction output can be spent by one and only one subsequent

transaction — and ensures that existing nodes and links in the index tree cannot be overwritten once created as long as the blockchain's guarantees hold. Thus, the BST represented by the index tree can only be added to, and indices, once bound to a particular hash, can never be rebound to anything else.

More crucially, for any index in the index tree, we can generate a short *proof of existence* that the index is bound to a particular 256-bit hash. This proof consists of the transactions that form the path, traced by applying the standard BST search algorithm for the index, leading from the root transaction to a transaction that includes the given index. Such a proof does not take up much space: for a tree with randomly-distributed indices, the length of a proof of existence is expected to be $\Theta(\log n)$. As an example, in the index tree in Fig. 1, the root hash and its left child constitute a proof that the index 1 is bound to a certain hash.

Any Bitforest client, with hardcoded knowledge of the root transaction, can validate a proof of existence in an index tree for an index $x$, by checking the following properties:

1) The transactions indeed exist in the blockchain.
2) The first transaction in the proof is indeed the already-known root transaction.
3) The fourth output of the last transaction binds a hash to $x$.
4) Each transaction in the proof $t_i$ spends the expected output of the previous transaction $t_{i-1}$. That is, if $x$ is smaller than the index in the fourth output of $t_{i-1}$, $t_i$ spends its first output while if $x$ is bigger it spends the second output; in case $x$ is equal to the index of a transaction not at the end of the path, the proof is declared invalid.

The last property is, essentially, to check that the proof of existence really ends at the "canonical" transaction for a certain index — the transaction on which a step-by-step search from the root would terminate. Malicious NAs may attempt to insert non-canonical transactions into the tree which refer to the same index, violating the BST invariant, but paths from the root transaction to these transactions would not satisfy the last property, preventing the NA from being able to show a proof of existence for them, safeguarding the insert-only property of the index tree.

Checking these properties takes very little time and storage space. Verifying that the transactions are confirmed in the blockchain can be done efficiently and securely by a lightweight thin client in almost any UTXO-based cryptocurrency blockchain. For example, for Bitcoin, this only requires the client to synchronize the Bitcoin block headers — a very small amount of data compared to the entire blockchain — using Simple Payment Verification (SPV) [3]. The remaining three steps are done locally by the client and complete essentially instantly.

### D. Updating an index tree

In the previous section, we sketched the general structure of the "tree" part of the index tree, which provides an insert-only, efficiently queryable mapping from numerical indices to 256-byte hashes embedded in the blockchain. Yet in Bitforest, names are not mapped to a fixed immutable piece of data, but rather to an operation log that can always be appended to. How is the gap between the two abstractions bridged?

The answer is the *update chain*, a Catena transaction chain [9] spending the 3rd output of a tree node transaction. A Catena chain is the simplest grow-only blockchain data structure based on exploiting double-spend prevention, and it provides an append-only linked list of log entries. In Bitforest's case, the update chain is used to store hashes of operation log entries other than the first one; the first operation's hash is stored inline in the tree node to reduce the number of transactions needed to register a new name.

By appending to the growing update chain, the NA can append more entries to the operation log; the underlying blockchain's double spend prevention forbids the NA from either rolling back the log to an earlier state or overwriting existing log entries, providing the abstraction we want: a mapping of keys to append-only operation logs.

The proofs of existence discussed in the previous subsection can easily be extended to the entire operation log: we simply include all the transactions forming the update chain in the proof. Clients then verify that each transaction in the update chain does indeed spend the previous entry's first output, and that the first transaction in the update chain spends the third output of the tree node recording the correct index. The size of the proof is now $\Theta(\ell + \log n)$, where $\ell$ is the length of the update chain, and $n$ is the number of names in the namespace. In practice, $\ell$ is unlikely to be large; names bindings in PKIs are typically changed only due to infrequent events such as name transfer or key revocation that may not even happen for the majority of names.

### E. Mapping names to indices

We have discussed how the index tree securely maps numerical indices to cryptographic hashes of operation logs. But how are these "indices" related to the names in the namespace? After all, the whole point of a Zooko's-triangle-violating PKI is to provide human-readable names; requiring clients to look up random numeric indices defeats the purpose.

One naive solution is to map names to indices with a cryptographically secure hash function, such as SHA-256. However, this direct approach is unworkable, as it allows anybody with access to the blockchain to check the existence of names in the namespace without involving the NA, facilitating *name enumeration* and severely weakening policy enforcement in the area of access control over listing names. In addition, malicious name registrants would be able to greatly degrade the performance of the service by registering names with indices that when inserted in order would grossly unbalance the index tree, causing the size of some proofs to be exorbitantly large.

The solution to both of these problems is to compute indices using a *verifiable random function* (VRF) [10], which is a random function that requires a private key to compute, but can then be publicly verified. One example of a VRF is VXEdDSA [11], which Bitforest uses in practice. Given such a function $\mathrm{VRF}()$, the index $x$ for a name $n$ is computed as:

$$x = \mathrm{HMAC}(\mathrm{VRF}_{K_{\mathrm{VRF}}}(n), n)$$

where $\mathrm{HMAC}()$ is an HMAC using SHA-256, and $K_{\mathrm{VRF}}$ is a public key belonging to the NA already known to clients of that NA.

Using a VRF solves both of the problems mentioned above. Firstly, name lookups now must go through the NA, as only the NA can compute the index using a VRF and walk the index tree to generate a proof of existence for that index. This eliminates the policy enforcement and name enumeration issues — both CONIKS [12] and EthIKS [8] use a similar construction to prevent name enumeration. Secondly, we obtain randomly distributed indices that are verifiable after the fact, but unpredictable by anybody other than the NA before a name is registered and placed in the index tree. Thus, malicious registrants cannot precompute the indices of names and register them in a pathological order that would unbalance the tree.

Now we finally have all the details to describe a full response by the NA to a client's query for a name $n$:

- A VRF output $\mathrm{VRF}_{K_{\mathrm{VRF}}}(n)$, which the client verifies and from which it derives the index $x$
- A proof of existence for the operation log bound to $x$, which the client checks according to the procedure given in II-C, containing a cryptographic hash of every element in the operation log
- The operation log itself, which the client checks does indeed hash to the values given in the proof of existence

### F. Operation logs and identity retention

The index tree, combined with a VRF-based function mapping, now gives us a way of securely obtaining an append-only history of any name, which can only be appended to through the NA — the operation log, consisting of many individual *operations*. Each operation in the operation log must be signed by a cryptographic identity declared in the previous operation, preventing any changes to a name's binding unauthorized by the owner of the name. This subsection will discuss the details of how the operation log works.

*1) Structure of an operation:* Each operation contains the following fields:

- A random nonce
- An *identity script* representing the cryptographic identity authorized to generate the next operation bound to the name, i.e. the "current owner"
- A collection of cryptographic *signatures*, valid with respect to the identity script declared in the previous operation
- Data associated with the name

The random nonce is used to prevent replay attacks, where the NA replays previous bindings in cases where a previous binding has signatures from the latest owner. It also randomizes the hash of each operation, preventing the operation log hashes in the index tree from leaking any information about the namespace to anybody without access to the NA.

*2) Identity scripts and signatures:* Identity scripts are the entities representing cryptographic identities in Bitforest. They encode a tree structure of key quorums; an identity script is recursively defined as either an Ed25519 [13] public key, or a quorum of $n$ out of $m$ identities. We use a simple stack-based scripting language, inspired by payment scripts in Bitcoin [3], to represent these trees.

We note here that nothing prevents this identity script system from expressing an entirely NA-trusting update policy that gives up identity retention for convenience, relying instead on transparency to implicitly regulate the NA's actions, similar to the behavior of "normal users" in CONIKS [12]. This can be done simply by assigning a keypair controlled by the NA as the "owner" which would allow the NA to append whatever log entry it wants without cryptographic signatures from the user of the name.

### G. Summary

In this section, we presented an architectural overview of Bitforest. A novel data structure, the index tree, allows for the implementation of a securely and efficiently queryable index-value mapping embedded inside a generic cryptocurrency blockchain, where indices can only be inserted and not deleted, and values can only be appended to and not overwritten. This mapping then serves as the basis for a naming system that provides a unique combination of the strong, distributed-trust identity retention guarantees of existing blockchain-based solutions, and the highly flexible policy enforcement found in traditional centralized PKIs.

## III. Implementation and evaluation

In this section, we describe our implementation of Bitforest and evaluate Bitforest against existing naming systems, both by measuring their performance using experiments and by numerical simulation. We also discuss the cost of operating a Bitforest namespace across popular public blockchains.

### A. Implementation

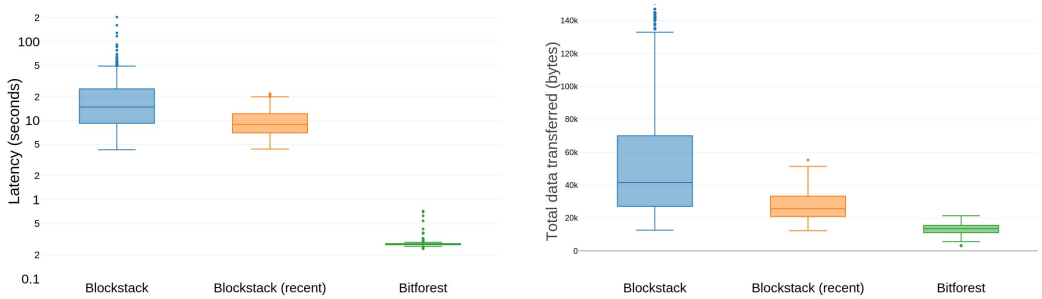
We created a prototype reference implementation of Bitforest in the Java programming language, which we plan on releasing as an open-source library in the future. The Bitforest library allows applications to easily create their own NAs and look up names securely; by default it uses the Bitcoin blockchain to store the index tree.

One particular area of implementation deserving some discussion is the ease of porting Bitforest to different blockchains. Although originally we implemented Bitforest for Bitcoin, in order to evaluate how well Bitforest achieves our goal of blockchain neutrality, we ported it to Litecoin [14], a well-known cryptocurrency "altcoin" with UTXO-based semantics. Implementing the Litecoin version of Bitforest proved to be very easy — based upon spending transaction outputs like the majority of cryptocurrencies, Litecoin allows us to encode the index tree exactly as we have described it.

### B. Lookup performance

In our first quantitative experiment, we evaluate the performance of doing lookups in Bitforest. As a comparison, we also evaluate the performance of secure client-server lookups in Blockstack, the current state-of-the-art in blockchain-neutral blockchain-based PKIs. Unlike most other blockchain-neutral systems, Blockstack has a secure thin-client lookup system — SNV — with at least some decentralized trust, allowing a contest between two systems with comparable security and portability.

Both a Blockstack full node and a Bitforest NA are installed on a server, and a client with around 90 ms of network latency to the server is used to benchmark the two systems. We create a Bitforest NA — using the Bitcoin "testnet" [3] to avoid




(a) Lookup latency

(b) Data transferred

Fig. 2: Lookup performance of Bitforest compared to Blockstack

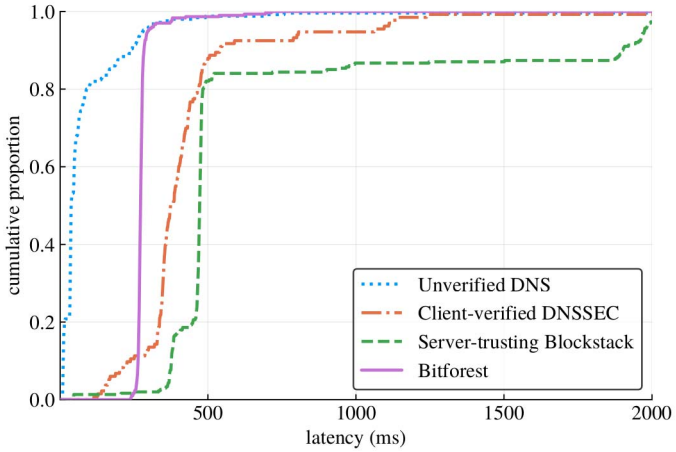



Fig. 3: Bitforest lookup latency, compared with centralized-trust systems

exorbitant transaction costs — and insert 73,000 names, the approximate amount of names in Blockstack. Then, we query our Bitforest NA with 300 random dummy names previously placed in the namespace, while for Blockstack we use SNV to verify 300 random existing name records in the operational Blockstack network; in both cases, we sample the namespace without replacement. Total latency and bytes transferred are then measured by tracing network packets using Wireshark, avoiding inaccurate measurements due to application startup latency.

Fig. 2 summarizes the results of this experiment. Note that Blockstack lookups grow significantly slower as we query names registered in older and older blockchain blocks, as the SNV process requires the client to iteratively "walk" across old blocks until it hits the one where the name is registered; thus, we have a separate metric for Blockstack names registered after the start of 2017 called "Blockstack (recent)".

We see that in both metrics, especially latency, Bitforest performs much better than Blockstack. The particularly lopsided difference in latency measurements likely stems from Blockstack's SNV implementation, which requires the client to incrementally walk backwards along Blockstack's virtualchain, taking up a large amount of network round trips to complete the procedure; on the other hand, Bitforest NAs give full proofs of existence for their index trees in a single request-response cycle. In particular, doing SNV proofs for old Blockstack names can take up to minutes.

Even considering this implementation flaw, though, Bitforest still proves significantly more efficient — this is illustrated by the comparison of data transferred, a metric less affected by issues with the implementation. The advantages of Bitforest's very simple and efficient index tree, as opposed to Blockstack's virtualchain, clearly show, allowing Bitforest queries to transfer only around 10 KB of data, while Blockstack SNV lookups use up several times or even an order of magnitude more bandwidth. Quite evidently, Bitforest's lookup procedure is far more performant than Blockstack's state-of-the-art blockchain-based secure thin-client lookup.

Furthermore, Bitforest offers acceptable speed even compared to systems with centralized trust. Fig. 3 compares lookup latency between Bitforest and three systems lacking distributed trust: unsecured DNS, DNSSEC-secured DNS with all signature validation done by the client, and Blockstack's default server-trusting mode. For DNS, we look up a list of US government DNSSEC-enabled domains [15] using the `dig` tool [16] and its `+sigchase` option, while for Bitforest and Blockstack we use the same set of random names from the first experiment. We see that though DNS is very fast due to its aggressive caching, compared with client-verified DNSSEC and server-trusting Blockstack, both of which are less amenable to caching by the ISP, Bitforest offers excellent performance, even though it has fully-verifying clients and distributed trust.

Finally, we numerically simulate the lookup overhead of Bitforest for extremely large namespaces impractical to create on the Bitcoin testnet in a reasonable period of time. This is to test whether or not proofs of existence stay reasonably small even as the index tree grows deeper as more names are registered. We create index trees of sizes ranging from 100 to 10 million locally without broadcasting any transactions onto the blockchain, and then randomly sample the sizes of proofs of existence. The results of the simulation are plotted in Fig. 4; the solid line indicates the median proof size, while the error bars plot the interval in which 95% of the samples for each namespace size lie. We see that even for very large namespaces, Bitforest's lookup overhead remains small — a 10-million-name index tree, for example, has proofs of existence ranging in size from 10 to 19 KB.

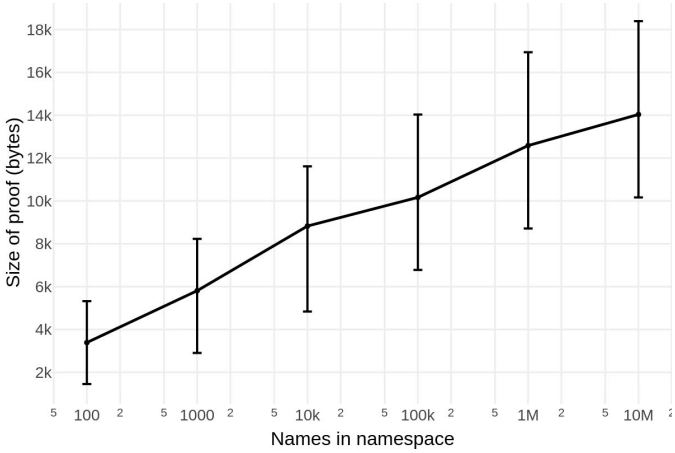



Fig. 4: Bitforest proof-of-inclusion sizes for large namespaces

In conclusion, we see that the price Bitforest pays in lookup performance to achieve much stronger security is quite small, and should not be problematic for the vast majority of applications.

### C. Bootstrapping data

To achieve identity retention as strong as that of Bitforest, previous blockchain-based systems must use full nodes rather than thin clients like Blockstack's SNV, but blockchain full nodes are notorious for requiring very large and linearly-growing amounts of *bootstrapping data*. Classically, all full nodes must download the entire blockchain on first connection and continually synchronize it to local storage, causing large delays in joining the network. Even if optimizations such as pruning [3] and "fastsync", commonly deployed on Blockstack nodes [6], obviate the need to actually store all the blocks seen and speed up initial download, new blocks must still be constantly replicated across all nodes, using up significant amounts of bandwidth.

Bitforest eliminates the need to track an entire blockchain, but thin clients for blockchains often still have a piece of linearly growing bootstrapping data — the Bitcoin SPV blockchain headers for example. How does the cost of keeping up with this data compare to that of downloading and keeping up with a blockchain?

TABLE I: Growth rates of bootstrapping data

| System | Mean monthly growth | Cumulative size |
|---|---|---|
| Bitcoin (Blockstack) | 1.20 GB | 155.6 GB |
| Ethereum | 3.37 GB | 337.6 GB |
| Namecoin | 67.6 MB | 5.26 GB |
| Bitcoin SPV (Bitforest) | 370.0 KB | 40.7 MB |

To answer this question, we use Table I, which shows how fast bootstrapping data grows for various blockchains and Bitcoin SPV, using blockchain data gathered from existing historical records. Note that Blockstack full nodes also need to catch up with the Bitcoin network like a Bitcoin full node, and thus the Bitcoin numbers also apply to Blockstack. Bitforest on Bitcoin has storage overhead corresponding to the numbers we give for Bitcoin SPV — the overhead of storing the Bitcoin block headers.

It is clear that the rate at which blockchains typically grow is quite high, and even blockchains with very low activity, such as Namecoin, still eventually accumulate gigabytes of blocks. On the other hand, although the thin-client access to blockchains needed by Bitforest does have linearly growing bootstrapping data, the growth rate is minuscule compared to that of blockchains and would not be a problem for all but the most tightly constrained embedded environments.

### D. Costs of operating an NA

By committing transactions to a blockchain every time names are registered or updated in the namespace, Bitforest NAs typically need to pay blockchain-dependent transaction fees. For high-volume public blockchains, these fees can incur a substantial cost to the operator of the NA — this is especially apparent with blockchains that have both high usage and a poorly-scaling design, such as Bitcoin.

TABLE II: Bitforest transaction costs for various cryptocurrencies. All prices are in US dollars.

| | Bitcoin | Litecoin |
|---|---|---|
| Bootstrapping | $0.032 | $0.022 |
| Creating new name | $0.026 | $0.015 |
| Updating name | $0.013 | $0.011 |

However, since Bitforest is blockchain-neutral, it is possible to choose between blockchains based on the cost of creating transactions. Table II illustrates the costs of Bitforest operations over the two blockchains our prototype supports, based on exchange rates obtained from Coinbase [17], one of the most popular cryptocurrency exchanges, in October 2018. We clearly see that Bitforest operations does incur some transaction fees to the NA, roughly comparable to those of systems such as EthIKS [8], but these costs are quite low, and vary from blockchain to blockchain.

## IV. Conclusion

In this paper, we presented Bitforest, a naming system with blockchain-backed security. Bitforest uses a new architecture informed by the successes and shortcomings of existing blockchain-based systems such as Blockstack and EthIKS, combining a centralized lookup service trusted only for policy enforcement with a novel, blockchain-neutral data structure built from cryptocurrency transactions. This provides an efficiently queryable mapping from names to cryptographic hashes of their respective bindings, while ensuring that the central administrator cannot do anything that violates identity retention without breaking the security guarantees of the underlying cryptocurrency blockchain. Bitforest also allows high flexibility for the administrator in areas such as access control and privacy. Experimental results show that Bitforest performs markedly better than existing blockchain-based systems, with faster lookup than thin clients and dramatically reduced storage overhead compared to full nodes; we also demonstrated that its performance penalty compared to traditional centralized PKIs is small to nonexistent.

## REFERENCES


[1] A. Delignat-Lavaud, M. Abadi, A. Birrell, I. Mironov, T. Wobber, and Y. Xie, "Web pki: Closing the gap between guidelines and practices." in *NDSS*, 2014.

[2] A. Niemann and J. Brendel, "A survey on ca compromises." [Online]. Available: https://www.cdc.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_CDC/Documents/Lehre/SS13/Seminar/CPS/cps2014_submission_8.pdf

[3] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[4] "Namecoin." [Online]. Available: http://namecoin.info

[5] C. Fromknecht, D. Velicanu, and S. Yakoubov, "A decentralized public key infrastructure with identity retention," *Massachusetts Inst. Technol., Cambridge, MA, USA, Tech. Rep*, vol. 6, 2014.

[6] M. Ali, J. Nelson, R. Shea, and M. J. Freedman, "Blockstack: A global naming and storage system secured by blockchains," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016, pp. 181–194.

[7] H. A. Kalodner, M. Carlsten, P. Ellenbogen, J. Bonneau, and A. Narayanan, "An empirical study of namecoin and lessons for decentralized namespace design," in *WEIS*, 2015.

[8] J. Bonneau, "Ethiks: Using ethereum to audit a coniks key transparency log," in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 95–105.

[9] A. Tomescu and S. Devadas, "Catena: Efficient non-equivocation via bitcoin," in *IEEE Symp. on Security and Privacy*, 2017.

[10] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 1999, pp. 120–130.

[11] T. Perrin, "The xeddsa and vxeddsa signature schemes," *Specification. Oct*, 2016.

[12] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "Coniks: Bringing key transparency to end users." in *USENIX Security Symposium*, 2015, pp. 383–398.

[13] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures," *Journal of Cryptographic Engineering*, pp. 1–13, 2012.

[14] C. Lee, "Litecoin," 2011.

[15] "Estimating usg ipv6 & dnssec external service deployment status." [Online]. Available: https://fedv6-deployment.antd.nist.gov/cgi-bin/generate-gov

[16] "dig(1) — linux man page." [Online]. Available: https://linux.die.net/man/1/dig

[17] Coinbase, "Bitcoin, ethereum, and litecoin price." [Online]. Available: https://www.coinbase.com/charts