Decentralized Public Key Infrastructure

by (alphabetical by last name) Christopher Allen, Arthur Brock, Vitalik Buterin, Jon Callas, Duke Dorje, Christian Lundkvist, Pavel Kravchenko, Jude Nelson, Drummond Reed, Markus Sabadello, Greg Slepak, Noah Thorp, and Harlan T Wood

Abstract

Today's Internet places control of online identities into the hands of third-parties. Email addresses, usernames, and website domains are borrowed or "rented" through DNS, X.509, and social networks. This results in severe usability and security challenges Internet-wide. This paper describes a possible alternate approach called *decentralized public key infrastructure (DPKI)*, which returns control of online identities to the entities they belong to. By doing so, DPKI addresses many usability and security challenges that plague traditional public key infrastructure (PKI). DPKI has advantages at each stage of the PKI life cycle. It makes permissionless bootstrapping of online identities possible and provides for the simple creation of stronger SSL certificates. In usage, it can help "Johnny" to finally encrypt thanks to its relegation of public key management to secure decentralized datastores. Finally, it includes mechanisms to recover lost or compromised identifiers.

1. Introduction — Why DPKI

Section Contributors Alphabetical By Last Name: Christopher Allen, Christian Lundkvist, Jude Nelson, Drummond Reed, Markus Sabadello, and Greg Slepak

The Internet facilitates communications and transactions between individuals worldwide. This is conducted through the use of identifiers such as email addresses, domains, and usernames. But who controls these identifiers? How are they managed? And how is secure communication facilitated between them?

In the modern day, third-parties such as DNS registrars, ICANN, X.509 Certificate Authorities (CAs), and social media companies are responsible for the creation and management of online identifiers and the secure communication between them. Unfortunately, this design has demonstrated serious usability and security shortcomings.

The Control & Management of Online Identifiers

When DNS and X.509 PKIX were designed, the Internet did not have a way to agree upon the state of a registry (or database) in a reliable and decentralized manner. Therefore, these systems designated trusted third-parties to manage identifiers and public keys. Virtually all Internet software now relies on these authorities. Because of this, website domains do not really belong to the organizations that register them (NOTE: Instead, they belong to third parties like ICANN, domain registrars, Certificate Authorities, and anyone capable of influencing, coercing, or hacking into them.) and, similarly, usernames on websites do not really belong to those users.

These trusted third-parties (sometimes abbreviated TTP), act as corruptible central points of failure, each capable of compromising the integrity and security of the entire Internet. Because control of these identifiers is given to TTPs, the usability of those identifiers is also compromised. These issues with corruptibility and usability cause additional problems:

- Some companies spend significant resources fighting security breaches caused by misbehaving CAs;
- Many websites still do not support HTTPS;
- Truly secure and user friendly communication still remains out of reach of for most netizens. [ref: "Why Johnny Can't Encrypt" http://arxiv.org/abs/1510.08555]

For all these reasons, the foundational precept of DPKI is that *identities belong to the entities they represent*. That requires designing a *decentralized* infrastructure where every identity is controlled not by a trusted third-party, but by its *principal owner*.

The Security of Online Communication

Online communications are secured through the safe delivery of *public keys*. These keys correspond to identities. The entity these identities represent, called the *principal*, uses a corresponding secret *private key* to both decrypt messages sent to them, and to prove they sent a message (by signing it with the private key).

PKI systems are responsible for the secure delivery of public keys. However, the commonly used X.509 PKI, PKIX, undermines both the creation and the secure delivery of these keys.

The Challenges of Third Parties: Finding "The Right Key"

In X.509 PKIX, web services are secured through the creation of the keys signed by CAs. However, the complexity of generating and managing keys and certificates in PKIX has caused web hosting companies to manage the creation and signing of these keys themselves, rather than leaving it to their clients. This creates major security concerns from the outset, as it results in the accumulation of private keys at a central point of failure (the web hosting company), making it possible for anyone with access to that repository of keys to compromise the security of the connections to those websites in a way that is virtually undetectable.

The design of X.509 PKIX also permits any of ~1200 CAs around the world to impersonate any website. This is further complicated by the risk of coercion or compromise of a CA. Because of these dangers, users cannot be certain that their communications are not being compromised by a fraudulent certificate allowing a MITM (Man-in-the-Middle) attack. These attacks are extremely difficult to detect; companies like Google that produce web browsers can sometimes recognize attacks on their own websites, but they cannot prevent attacks on arbitrary websites.

Workarounds have been proposed. HPKP is an IETF standard that lets websites tell visitors to "pin" the public key they receive for a period of time (ignoring any other key). However, such mechanisms are difficult for website administrators to use and therefore might not be used much in practice. HPKP is vulnerable to "Hostile Pinning", and in cases where the pin is legitimate it comes with a risk of breaking websites if key(s) need to be legitimately replaced. Worse still, some implementations of HPKP make it trivial for a third-party to override arbitrary pins without user consent.

The Usability of PKI

Even if third-party authorities could be trusted, the current PKI system has major usability problems. A group

from Brigham Young University investigated the usability of Mailvelope, a browser extension that supports GPG-encrypted communication through third-party websites like Gmail. Their research demonstrated a 90% failure rate in secure communication attempts among the participants. Public key management, the study found, was the main reason that users were unable to use the software correctly.

Even TextSecure/Signal — a secure messaging system endorsed by Edward Snowden for its security and ease of use — has usability problems due to its inability to smoothly handle public key changes. If a user deletes and reinstalls the app, their friends are warned that their public key "fingerprint" has changed. This scenario is indistinguishable from a MITM attack, and few users are likely to understand or bother verifying that they received the correct public key.

The Danger of Message Compromise

As a result of conventional PKI's usability challenges, much of Web traffic today is unsigned and unencrypted. This is particularly evident on the major social networks. Because of PKI's complexity, social networks do not encrypt their user's communications in any way, other than relying on PKIX by sending them over HTTPS. Because messages are not signed, there is no way to be sure that a user really said what they said, or whether the text displayed is the result of a database compromise. Similarly, user communication is stored in a manner that anyone with access to those databases can read — compromising user privacy and burdening social networks with large liability risks.

2. DPKI's Answer To The Web's Trust Problems

Section Contributors Alphabetical By Last Name: Drummond Reed, and Greg Slepak

The answer is not to abandon PKI, but to find an alternative: DPKI, a future specification for a *decentralized* public-key infrastructure.

The goal of DPKI is to ensure that, unlike PKIX, no single third-party can compromise the integrity and security of the system as as whole. Trust is decentralized through the use of technologies that make it possible for geographically and politically disparate entities to reach consensus on the state of a shared database. DPKI focuses primarily on decentralized key-value datastores, called *blockchains*, but it is perfectly capable of supporting other technologies that provide similar or superior security properties.

Third-parties, who are called miners (or validators), still exist, but their role is limited to ensuring the security and integrity of the blockchain (or *decentralized ledger*). These third-parties are financially incentivized by a *consensus protocol* to follow the rules of the protocol. Deviation from the protocol results in financial punishment, while consistency with the protocol typically results in financial reward. Bitcoin, devised by Satoshi Nakamoto, is the first such successful protocol. It is based on *proof-of-work*, in which the *energy expenditure* of "miners" is used to secure the database.

A principal can be given direct control and ownership of a globally readable identifier like a website domain by registering the identifier in a blockchain, just like any other type of transaction. Within the key-value datastore (NOTE: In this case "key" refers to a database lookup string, not a public or private key.), the principal uses the identifier as the lookup key.

Simultaneously, blockchains allow for the assignment of arbitrary data such as public keys to these identifiers

and permit those values to be globally readable in a secure manner that is not vulnerable to the MITM attacks that are possible in PKIX. This is done by linking an identifier's lookup value to the latest and most correct public keys for that identifier.

In this design, control of over the identifier is returned to the principal. No longer is it trivial for any one entity to undermine the security of the entire system or to compromise an identifier that is not theirs. This is how DPKI is able to address both the security and the usability problems that plague DNS and X.509 PKIX.

A complete description of blockchains and their consensus protocols is beyond the scope of this paper. However, *§5, "Security of Identifiers and Public-Keys,"* discusses some of their security properties and the Appendix *"Thin Client Details"* describes how the data in these blockchains can be securely accessed from mobile devices that do not themselves have a full copy of the blockchain.

3. DPKI's Threat Model

Section Contributors Alphabetical By Last Name: Jude Nelson

Like conventional PKI systems, DPKI assumes that a persistent active adversary Mal constantly tries to trick one principal Alice into trusting the wrong key for another principal Bob. This can take the form of discovering the wrong identifier for Bob (e.g., finding the wrong account at twitter.com) or caching the wrong key once the identifier is known.

Assume that Mal is a computationally-bounded adversary who is capable of compromising or compelling centralized trusted PKI parties to trick Alice into trusting the wrong key. This has already been proven feasible, as in the case of DigiNotar and in cases wherein state actors force CAs in their jurisdictions to sign invalid keys. In addition, assume that Mal is capable of altering or blocking a bound fraction (less than 100%) of the messages exchanged between Alice and Bob. This is also feasible today, and is evidenced by ISP-level censorship, by request redirection, and by packet-mangling attacks, which are executed in order to disrupt existing file-sharing technologies like BitTorrent, to black-hole packets from known Tor exit nodes, and to block HTTP access to politically-sensitive materials.

In light of Mal's powers, two design principles for DPKI become apparent:

1. As has already been suggested, each principal must be in complete control of their current identifier/public-key binding. If only the principal can make changes to their identifier, then Mal is compelled to attack each principal she wishes to compromise. This is in contrast to traditional PKI, where Mal only needs to compromise one CA to trick many principals.

2. The system must make all-or-nothing forward progress: either every principal must witness every other principal's updates to their identifier/public-key bindings, or else no one may observe any updates. This is required to protect against Mal's possible network-level attacks by alerting the entire network to her presence if she censors updates to certain principals. This makes targeted attacks against certain users or key-pairs extremely costly, because it ensures that the only way Mal can attack anyone is to attack everyone at once.

As already suggested, DPKI achieves these design principles through use of secure decentralized key-value datastores to host the bindings between identifiers and public-key hashes. See *§5, "Security of Identifiers and*

Public-Keys" for details.

4. Registration and Identifiers

Section Contributors Alphabetical By Last Name: Christopher Allen, Christian Lundkvist, Jude Nelson, Drummond Reed, Markus Sabadello, Greg Slepak

As described in the previous sections, the core of DPKI are decentralized key-value datastores that can serve as identifier registries, allowing a principal's public keys to become securely associated with their identifier. As long as this registration remains valid and the principal is able to maintain control of their private key, no third-party can take ownership of that identifier without resorting to direct coercion of the principal.

DPKI does not specify what types of identifiers should be used and recognizes that different approaches are possible (e.g., usernames or UUIDs), which may differ in terms of ease-of-use, permanence, uniqueness, security, and other properties.

For DPKI to use a decentralized key-value store it must have the following properties:

- **Permissionless Writes**. Any principal can broadcast a message provided that it is well-formed. Other peers in the system do not require admission control. This implies a decentralized consensus mechanism.
- **Fork Choice Rule.** Given two histories of updates, any principal can determine which one is the "most secure" through inspection.

These needs can be met through blockchains such as Namecoin, Ethereum, and potentially even Bitcoin (through technologies such as Blockstore).

The Requirements of DPKI Registration

The way identifier registration is handled in DPKI is different from DNS. Although registrars may exist in DPKI, they must adhere to several requirements born out of DPKI's goal to ensure that *identities belong to the entities they represent:*

1. Private keys must be generated in a decentralized manner that ensures they remain under the principal's control (e.g., via open source client software on the principal's device). This means that registration services generating keypairs on a server on behalf of principals are ***explicitly prohibited***. To do otherwise would be to recreate the issues mentioned in *§1 "Introduction — Why DPKI"*.
2. Software must ensure that principals are always in control of their identifiers and the corresponding keys. Principals can extend control of their identifier to third-parties (e.g., for recovery purposes), but this must ***always*** be an explicit, informed decision on their part, and never a default, implicit, or misleading behavior of software. Private keys must ***never*** be stored or transmitted in an insecure manner.
3. Software must ensure, to greatest degree possible, that no mechanism exists that would allow a single entity to deprive a principal of their identifier without their consent. This implies:
    1. Once a namespace is created within a blockchain (e.g., via a smart contract on Ethereum), it

cannot be destroyed. Likewise, namespaces cannot contain blacklisting mechanisms that would allow anyone to invalidate identifiers that do not belong to them.

2. The rules for registering and renewing identifiers must be transparent, and they must be expressed in simple terms to users in a way that would be difficult to overlook or misunderstand (e.g., first-come-first-serve, auction). In particular, if registration is subject to an expiration policy, the principal **must** be explicitly warned that this could result in the principal losing control of the identifier.

3. Once set, namespace rules cannot be altered to introduce any new restrictions for renewing or updating identifiers, since otherwise it would be possible to take control of identifiers away from principals without their consent. Likewise, client software for renewing or updating identifiers cannot be modified to introduce new restrictions for updating or renewing an identifier.

4. By default, software for managing identifiers ***must ensure*** that all network communications for creating, updating, renewing, or deleting identifiers is sent via a decentralized, peer-to-peer mechanism. This, again, is to ensure that a single entity (like a registrar) cannot prevent identifiers from being updated or renewed.

We recommend that DPKI infrastructure also strive to ensure the existence of:

- At least one class of identifiers that do not expire once properly registered.
- At least one class of neutral registration policies available to all members of the public, as well as to any service provider that wishes to offer registration services.

DPKI should not discriminate against any party that wishes to use it, and registries should be considered a commons; their design and operation guided by principles of openness, neutrality, and inclusion (NOTE: Ostrom, Elinor (1990). Governing the Commons: The Evolution of Institutions for Collective Action. Cambridge, UK: Cambridge University Press. ISBN 9780521405997. or Allen, Christopher (2015). A Revised "Ostrom's Design Principles for Collective Governance of the Commons" http://www.lifewithalacrity.com/2015/11/a-revised-ostroms-design-principles-for-collective-governance-of-the-commons-.html).

The Mechanics of DPKI Registration

Registered identifiers are likely to have two types of keys associated with them: the keypair that's used for registering and for updating the data associated with the identifier, and the public keys associated with the identifier (*subkeys*).

It is recommended that the *subkeys* be used by the principal to sign messages. They can be stored directly or indirectly in the datastore:

- **Direct storage** means that the public key itself is stored directly in the DPKI datastore. For most blockchains, this is unlikely since some keys are quite large and most blockchains make storing them impossible or very expensive.
- **Indirect storage** means that a pointer (e.g. a URI) is stored alongside with—or itself containing—the fingerprint for the public key.

5. Security of Identifiers And Public Keys

Section Contributors Alphabetical By Last Name: Vitalik Buterin, Jude Nelson, and Greg Slepak

In DPKI, identifiers are typically lookup keys that map to values that can only be modified by the entity (or entities) with the corresponding private key(s). In such a system, the worst that can happen is:

- An outdated value for a lookup key is sent in response to a lookup.
- The owner of the identifier is not able to update its value due to censorship, and they lose ownership once the identifier expires.

These problems are addressable through the use of thin clients (discussed later) and censorship-circumvention tools.

It is also possible, although extremely unlikely, that a false value is sent for an identifier. This can happen, for example, if a blockchain that is secured by proof-of-work has an adversary capable of overpowering the honest nodes and reversing history beyond the point of registration. However, all participants of the system would be able to detect this attack because it would result in the orphaning of an extremely long chain.

This sort of problem is most likely to arise from a centralization of a blockchain, which is a larger security concern.

Protecting Against Centralization

The degree of decentralization plays a role in the security of a system. Centralized systems are vulnerable to manipulation, censorship, and compromise. They represent a single point of failure that users must trust. When centralized systems go down, they take all their users with them.

While blockchains may start out decentralized, they do not necessarily end up that way. This implies the need for a simple metric that can tell us whether or not a "decentralized datastore" really is still decentralized:

How many doors must you knock on to compromise the users of a system?

We can roughly define a metric for measuring the decentralization of most blockchains by counting the following entities (each of whom act a single point of failure for the entire system when centralized):

- **"Devs"** — The number of parties who have control over the behavior (source code) of the blockchain.
- **"Nodes"** — The number of blockchain replicas, measured by the number of full nodes.
- **"Validators"** — The number of blockchain miners/validator, who are responsible for creating new blocks and authorizing transactions.

Since compromise of any one of those groups leads to compromise of the system, we define the *decentralization of a blockchain* as:

```
Decentralization(Blockchain) = MIN("Devs", “Nodes”, “Validators”)
```

More informally, users can infer the decentralization of a datastore by the Quality of Service (QoS) that it provides. If, for example, users notice that they are suddenly unable to update their identifiers, then this could indicate censorship due to centralization.

A Datastore Agnostic Protocol To Protect Against Centralization

If DPKI were to specify a specific blockchain as its "de facto decentralized datastore", it would put centralization pressures on that blockchain. Worse, using a de facto datastore would could break DPKI if the blockchain became abandoned due to a lack of interest in the chain. Software developers, having coded support for a specific blockchain, would have to expend significant effort to rewrite that software to migrate to a different blockchain. Meanwhile, there could be serious security concerns or QoS issues.

Therefore, the use of an *agnostic protocol* for accessing decentralized datastores is a *fundamental requirement* to ensure the functioning and the decentralization of the DPKI as a whole. Agnostic protocols make it easier for users and developers to migrate should a different datastore better serves their needs. The mere existence of this possibility creates a market of decentralized datastores competing to meet the needs of users.

Securely Accessing Blockchain Data

Most end user devices will not run full nodes because of the resources required, so how do clients access the chain securely?

One solution is to do a blockchain-version of Convergence, wherein a set of "blockchain notaries" tell users the state of a particular object maintained by a blockchain, and the client software checks for unanimous agreement among a set of trusted notaries. However, this route arguably compromises what the key purpose of blockchain technology: removing the need for trusted intermediaries.

Fortunately, there is another technological solution: thin-client protocols. Thin clients download smaller portions of the blockchain, sufficient to provide security guarantees stronger than those provided by trusted intermediaries, but small enough to be used by any modern device. A detailed example of how one possible thin-client protocol works is discussed in the Appendix "Thin Client Details".

For blockchains lacking thin clients, the default should be Convergence-like unanimous consensus based on a random sampling of trusted nodes. These nodes should all see the same chain, so if even one of them disagrees, it is an indication that something is amiss and the event should be reported.

In general, this suggests a modular design, where devices are able to talk to any blockchain and use the most secure technique(s) available for *that* chain. It's possible that no single technique provides the greatest security, and in that situation the minimum number of techniques are combined to provide the highest level of security that the device can reasonably sustain.

Protecting Against Censorship

Finally, the security of DPKI must address censorship: whether the datastores are accessible to end users. A blockchain isn't useful if an ISP is censoring it.

Censorship circumvention technologies such as mesh networking, proxies, and onion routing, can be used to

bypass censorship of a blockchain network.

A separate but related concern is censorship of the data that's referenced by a blockchain, such as when a hash is stored in the value for an identifier, and the data represented by that hash is stored elsewhere. In this situation, the same techniques (e.g., onion routing, proxies) can be used in addition to looking up the hash over various different storage mechanisms [ref: see IPFS, Blockstore].

6. Recovering Lost Identifiers - Private Key Management

Section Contributors Alphabetical By Last Name: Vitalik Buterin, Christian Lundkvist, Pavel Kravchenko, Jude Nelson, Duke Dorje, Arthur Brock, Greg Slepak, Noah Thorp, and Harlan T Wood

Strong, reliable ownership of identifiers can make those identifiers highly valuable. Identifiers could be used to authenticate a user to the door of their house, their car, etc. These identifiers begin to represent the "keys to one's kingdom". It would be catastrophic if these identifiers were lost or compromised. Addressing that problem is therefore of paramount importance to DPKI's success.

Two Forms of Loss

Because of its importance, use of the *master key* must be minimized by any identity system that's built on top of DPKI. Indeed, this is the approach already taken by identity systems like Blockchain ID. Instead of using the master key to sign messages, *subkeys* are created for each new service that the identifier is used with.

This means there are two types of keys that can be lost or compromised:

- The *master private key*, which controls the data that's associated with the identifier. Losing this key can mean loss of control of your online identity.
- The *subkeys*, which are linked to the identifier and are stored as part of the identifier's data.

The security and recovery properties for the *master key* and *subkeys* are slightly different. The following are overviews of both possibilities; a full treatment of this topic is beyond the scope of this paper and is left for future work.

Recovery of the Master Key

He who controls the master key to an identifier is the identifier's master.

There are various mechanisms that can be used to recover a *master key* in a decentralized system.

Recombining Shards of the Master Key

Principals can protect themselves against *master key* loss by distributing *shards* of the master key to trusted entities. Shamir Secret Sharing and Threshold Signatures are two techniques that can be used to generate and recombine these shards.

In the event of loss, the principal would ask for N shards of the master key from M entities. N is the number of distinct shards required for recovery. Upon receiving the N shards, the master key would be successfully recovered.

This technique does little to protect principals in the event of master key compromise, however.

Sharding Diagram

Protecting Against Compromise

The danger of compromise comes about from a single entity having the master key in their possession any point in time. We can address this issue by ensuring that *no single entity possess the master key at any single point in time.*

For example, we can envision a system where, upon registration, users select five entities that they trust to guard their identity. These entities could be represented by trusted persons, organizations, or even devices. Though they act like authorities, they are never forced upon anyone and are always chosen by the principal themselves.

A master key is then generated ephemerally, broken into shards, sent to these entities, and immediately destroyed. Threshold signature schemes can be used in place of Shamir Secret Sharing so that a master key never needs to be recombined in its entirety on any given device.

Using Smart Contracts

Some blockchains, such as Ethereum, support arbitrary computation. In such cases,*principals can construct recovery mechanisms proportional to their level of paranoia.*

As a trivial example, a company like Google could secure their control over a blockchain-domain by using a namespace where a smart contract is used to update its value. The smart contract can be coded to function only when it receives a message signed by 6 out of 10 entities, or follow any other arbitrary logic.

Recovery And/Or Revocation of Subkeys

Subkey compromise or loss is less of a concern than loss or compromise of a*master key*, because verification is typically done using the current set of subkeys for an identifier. If a subkey is lost or compromised, the master key can simply be used to securely generate and replace the old subkey(s) in a blockchain. However, depending on how they are used, old subkeys might still require recovery or revocation.

As mentioned previously, the importance of the master key implies that identifiers will be authenticated through messages signed by the *subkeys* of an identifier, and not messages signed by the*master key*. However, since those messages are typically associated with the identifier itself, they are in effect being signed by the master key (since the *master key* is directly tied to the identifier). Therefore, the master key can still be used to sign and disseminate messages revoking one or more historical subkeys.

Recovery of lost subkeys can be done using the sharding mechanisms described previously. Alternatively, as with the group-based recovery schemes described above, a principal can choose to designate authority over their identifier to a group. This group could have the ability to sign new subkeys as belonging to the identifier, as well as the ability to sign messages that indicate an old key was compromised and therefore revoked.

Conclusion

In this paper, we discussed how identity is managed online today through globally-readable identifiers like website domains. We identified various security and usability problems in the Internet's two primary identity management systems: DNS and X.509 PKIX. We pinpointed the source of these problems to be the centralized nature of these systems, which prevents the entities represented by these identifiers from truly controlling them, making it possible for third-parties to compromise their security.

We then showed how the security and usability problems of DNS and PKIX can be addressed through the use of decentralized key-value datastores, such as blockchains, to create a specification for a *Decentralized* Public Key Infrastructure (DPKI). In describing the properties of DPKI, we showed that DPKI works even on resource-constrained mobile devices, and that it is able to preserve the integrity of identifiers by protecting organizations from private key loss or compromise.

Our future work is to develop a full specification for DPKI through an Internet standards body like the IETF.

References

Government Innovation in eID + Citizen EngagementBC Identity Citizen Consultation Results

Namecoin's UNO Commitments by Daniel Kraft

- https://forum.namecoin.info/viewtopic.php?f=5&t=2239

Namecoin's Analysis of various Thin Client Models

- https://github.com/hlandau/ncdocs/blob/master/stateofnamecoin.md#spvutxo-cbc
- https://github.com/hlandau/ncdocs/blob/master/stateofnamecoin.md#spvutxo-cbcuno-nx-cbc

Ethereum Related Documents On Thin Client Relevant Material

- https://easythereentropy.wordpress.com/2014/06/04/understanding-the-ethereum-trie/
- https://github.com/ethereum/wiki/wiki/Patricia-Tree

Appendix: Thin Client Details

Section Contributors Alphabetical By Last Name: *Vitalik Buterin, Greg Slepak*

Types of Information

What kind of information do thin clients want to know?

The following are some examples:

- Determining the time that a particular record was created or first seen
- Finding the public key currently associated with an account ID
- Finding the IP address and other data currently associated with a domain name

- Learning about key revocations (with no specified process for finding a replacement key)
- Determining the most recent version that a developer has published for a particular software package

More generally, we can decompose this into three categories of problems:

- **Proving existence**: proving that an event of a particular kind happened at time T
- **Proving inexistence**: proving that an event of a particular kind did not happen between times T_1 and T_2
- **Proving state**: proving that the "state" of an application, which can potentially be a result of complex “state transition” rules from applying many transactions, is equal to X at time T

These are roughly arranged in increasing order of hardness; proving existence is the easiest, and proving state is the hardest.

Degrees of Security

In general, a thin-client protocol on top of a blockchain can offer three levels of security:

- **Maximum security**: if a thin client makes a query, and any node responds with a valid reply to that query, then (*provided we can detect block withholding attacks*) the thin client can immediately either learn (i) the correct answer to the query, or (ii) that the response was invalid and should be ignored.
- **1-of-N trust security**: if a thin client makes a query, and it is accepted as a security assumption that at least one honest node will respond correctly within T seconds, then the thin client can learn the correct answer after T seconds, regardless of how many offline/faulty/byzantine nodes there are.
- **N/2-of-N trust security**: if a thin client makes a query, it must select some set of nodes (say 100) that it trusts to respond, and then randomly sample 3 or so nodes out of that list and require a unanimous agreement for the response. It will learn the correct answer as long as all 3 nodes do not collude. If any node is offline, it can continue to randomly search for an online node, until it’s checked some upper threshold of nodes, and hard-fail with an error once it reaches that threshold.

As an example of how these models apply, consider the simple case of "finding the current public key associated with an account", assuming that there exists a single master key (or set of master keys) that has the right to revoke and replace keys. Suppose that we have a blockchain where only transactions are placed in Merkle trees. Then, a thin client sends a request asking the network for a Merkle proof of the most recent transaction that replaced the public key. If the client receives an answer, it knows:

- **With the maximum security assurance** that this was the valid key at some point in the past.*This is a "proof of existence" problem.*
- **With 1-of-N trust security assurance** that this is still the valid key. (Theoretically, a newer replacement transaction could exist, but a 100% collusion or censorship could lead to the client never learning about it.) *This a "proof of inexistence" problem.*

For protocols that have more complex needs (eg. implementing complex name registrar rules), we are forced to deal with the more generalized problem of "proving state". If we use a simple blockchain that only keeps

track of transactions, clients would only know the answer with N/2-of-N trust security assurance. However, if we have a blockchain where the state is in a Merkle tree, then the client can learn absolutely any fact with maximum security assurance.

Because different blockchains have different levels of usage of Merkle proofs, our proposed solution is to develop an abstract protocol by which different blockchains can be used (as no single blockchain is 100% guaranteed not to be fatally flawed, we want an abstract model similar to that used for encryption algorithm choices), and which automatically attempts to provide the best level of security available depending on the blockchain's capabilities. Notaries would be available as a backstop, but blockchain plugins would exist which the client could install to support specific blockchains. These plugins would intelligently make blockchain queries that would provide as much security as possible, based on whether the blockchain supports strong security assurances for the specific kind of problem (proof of existence, proving state, etc) in question.

Thin Client Protocols

Thin-client protocols typically work in two stages.

First, the thin client downloads only a portion of the chain, typically the *header chain***.** The header chain typically contains a very small amount of information (typically 80-600 bytes) for each block containing metadata, such as (i) proof of work nonces; (ii) the root of a cryptographic hash tree, such as a Merkle tree, containing data such as transactions; and (iii) possibly the state of the application that the blockchain keeps track of.

Second, the client validates the header chain by using the blockchain's underlying consensus algorithm (e.g. checking proof of work or proof of stake signatures). Afterward, the client treats the header chain as "trusted". It applies cryptographic techniques that use the data in the header chain as a "root hash", from which it can verify claims about the rest of the data stored in the blockchain.

Fetching the Header Chain

The first task for a thin client is to download and verify the header chain. Assuming a working network connection, this is easy. For example, in the proof-of-work case the client asks the network for as many block headers as it can provide, the network replies back, and the client checks to make sure that each header has valid proof of work and then determines the "longest" chain of valid block headers (where "longest" is taken to mean "represents the most cumulative work"). More advanced protocols using skiplists exist so that clients do not even need to download every block header, though in-depth discussion of this is beyond the scope of this paper.

The main challenge with this mechanism is simple: what if the network connection is compromised? Potentially, an internet service provider could attack a user by censoring replies that tell a client about the official chain, and instead tell the user about their own fork. With proof-of-work protocols, one can statistically detect this by noticing a reduction in the rate of block production; however, more research is needed on determining the best and most reliable way to do this.

Verifying with Merkle Trees

After a thin client has successfully received a small piece of data that is "trusted" it must be able to verify

claims about the rest of the data in the chain. This relies on Merkle trees. A Merkle tree is a hashing algorithm where a large number of “chunks” of data are hashed a few pieces at a time, and then the resulting hashes are themselves put into small groups and hashed and so on recursively until the process results in one single hash, called the **root**. A simple depiction of this is as follows:

Merkle Trees

The benefit of this method is that the membership of any single chunk of data in the tree can be proven via a Merkle branch, which is the subset of nodes in the tree whose values are used in the process of computing the root hash.

Merkle Subsets

With just this set of nodes, a thin client can verify that a particular chunk is in the tree has a particular proof. The scheme is secure up to collision resistance; in order for an attacker to cheat the scheme, the attacker would need to break the underlying hash function. There are many different kinds of Merkle trees, including simple binary trees and more advanced designs such as Merkle Patricia trees that allow for efficient insert and delete operations, but the basic principle is the same.