SURF: Software Update Registration Framework

by

Woojung Kim

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2018

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

Chapter 3 borrows content from two papers "Conifer: centrally-managed PKI with blockchain-rooted trust" [32] and "Bitforest: a portable and efcient blockchain-based naming system" [31].

Abstract

BlockSURF (**Block**chain-based **S**ecure **U**pdate **R**egistration Framework) or SURF in short is a software framework designed to enable developers to build a blockchain-based secure update system which distributes trust over a blockchain. The primary objective of SURF is to create an immutable anchor for each software update registration on a blockchain and enable a wide spectrum of clients ranging from high-end servers to low-profile IoT devices to securely verify updates with minimal performance overhead. By introducing a partially trusted entity which serves client requests and handles blockchain-related business logic, SURF successfully decouples clients from an underlying blockchain, making the system blockchain-agnostic.

Acknowledgements

I would like to thank all the people who made this thesis possible.

Dedication

This is dedicated to my wife, Ellie, who has been supportive for the entire study.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software update systems perform a myriad of tasks. They deliver a multitude of revisions to end user, such as adding new features, removing outdated programs, providing bug fixes, and most importantly, fixing security vulnerabilities that have been discovered. Traditionally, software update systems are built as a centralized system such that all end users query the latest updates from one single server and blindly trust what it gives is the correct information [52]. Though convenient, such a centralized trust scheme has caused tremendous problems over the history of software industry [16, 52]. Hackers with malicious intents often target update servers because they are the easy subjects which can effectively introduce large-scale attacks once successfully compromised. Although many solutions have been proposed to solve security problems of software updates [45, 1, 15, 21, 52], there is one important question which has not been answered: given an adversary has successfully compromised a software update system and can respond to client requests with arbitrary data, can a client securely verify updates? If it is possible for clients to determine the information they received is different from what they expect, clients can maintain secure states even with the presence of a powerful adversary who can overwhelm the centralized update system. To answer this question, we propose **Block**chain-based **S**oftware **U**pdate **R**egistration **F**ramework (BlockSURF or SURF in short) that decentralizes trust using blockchain technologies.

SURF takes away trust from a centralized entity and places it in a distributed ledger — blockchain. By introducing a partially trusted intermediary, SURF allows developers to write software updates to a blockchain via the intermediary and enables clients or end-users to verify stored updates through the same intermediary. All write operations on the underlying blockchain should leave verifiable proofs such that allows developers and clients

to cryptographically verify the integrity of stored updates without placing trust on the intermediary.

Given a secure blockchain, SURF enables clients to verify updates even if the intermediary and its infrastructure are fully compromised. With this strong threat model, SURF assures the resiliency of clients with the following properties:

- **Security**: Communications between the intermediary and clients are end-to-end secure. That is, clients can safely verify update records even in an unreliable, possibly rogue network. Note that this does not necessarily mean the communication channels between the intermediary and clients are secure, but the contents delivered from the intermediary to clients are tamper-free and verifiable by clients.
- **Performance**: SURF decentralizes trust only and keeps the control centralized to guarantee the performance. Operations such as update registration or lookup are comparably fast to existing systems.
- **Blockchain-agnostic**: SURF is designed to be blockchain-agnostic so that it can leverage any UTXO-based blockchains (e.g., Bitcoin, Litecoin) as an underlying system. It is also possible to use account-based blockchains (e.g., Ethereum, Hyperledger Fabric) with small modifications.
- **Ease of use**: SURF is an easy-to-use framework that can be efficiently integrated with existing update systems.
- **Quantum-safe**: SURF can adopt post-quantum cryptography to keep the system safe from powerful adversaries with general quantum computers. SURF can migrate to a quantum-safe blockchain when the quantum attacks becomes feasible.

The thesis is structured as follows.

Chapter 2 introduces threats and commonly observed attacks toward software update systems and existing attempts remedying the problems as well as their shortcomings, necessitating the development of SURF . SURF effectively decentralizes trust and resolves problems of existing solutions in software update industry; we cover the designs and implementation details of SURF in Chapter 3. By taking advantage of semi-centralized structure, SURF provides comparable performance in terms of latency while maintaining clients secure from attacks at constant storage cost; Chapter 4 evaluates security and performance of SURF as well as the operation cost structure that SURF incurs. Finally, Chapter 5 summarizes the key contributions and concludes the thesis.

Chapter 2

Background

In this chapter, we explore the attack surfaces of software update systems, how traditional (centralized) update systems respond to potential threats, and how blockchains can help build secure software update systems by providing secure naming systems.

2.1 Attack surfaces

The purpose of having a software update system is to enable user devices, which are likely dispersed geographically, to receive the up-to-date software packages. While software providers make their best effort to securely deliver updates to users, there are a few pitfalls that adversaries may exploit in order to compromise update systems and the users relying on such systems.

2.1.1 Cryptography and implementation weaknesses

Many update systems employ cryptography to protect themselves from attackers. However, not all cryptographic methods are safe from attacks. For instance, some cryptographic methods such as MD5 and SHA1 are outdated and known to be vulnerable as they are no longer collision-free [54, 8]. Surprisingly many systems, which are supposedly security-sensitive, still use insecure, weak cryptographic methods [52]. This is largely due to the difficulty of switching from one cryptographic scheme to another. Although many traditionally preferred cryptographic primitives become obsolete due to recent technological advances, systems built a long time ago often cannot simply replace the old schemes with

new secure cryptographic methods as their implementation is tightly coupled with the used primitives.

2.1.2 Key theft

Although an update system can be fortified with a strong cryptography, the security of a system entirely relies on a set of cryptographic keys that the system uses. Thus, an update system that is built upon a single cryptographic key can be easily hacked if the key is compromised.

One of the methods adversaries use to steal keys is exploiting system vulnerabilities [36]. For example, attackers may exploit a known software bug of a web server to steal the root privilege of an update system [27, 28]. After successfully exploiting the known bug, attackers can freely access to the keys that the update system uses given that the keys are stored in the disk unencrypted. The chance of such an attack to take place is surprisingly high. As today's systems depend on many third party software packages, the chance of breaking into the system is equal to the chance of breaking into the weakest one, and it is therefore difficult to assure a system is free from vulnerabilities. Even the most famous, reputable software packages can contain vulnerabilities. For example, the Heartbleed attack [11], an exploit using an OpenSSL vulnerability, shocked the developer communities and industry. The attack, which is now patched, was effective in stealing the server credentials by allowing attackers to sneak a peak of a partial image of the server memory. The impact of the attack was phenomenal given that OpenSSL was and is the most commonly used software in the world. This clearly shows that even the most prestigious software can be exploited.

Evidently, no software update systems are completely safe from key theft threats which exploit software vulnerabilities. The likelihood of update systems being compromised due to key theft consistently increases as the level of sophistication of attacks and the number of exploits increase.

2.1.3 Naming system vulnerabilities

Generally, an update system uses a client-server architecture which lets clients (i.e., users or user devices) download updates from a server. The clients are likely located remotely and, thus, need to connect to the server through a possibly insecure network such as the Internet. As it is difficult for clients to securely locate and identify the update server over an insecure network, using a reliable naming system such as a public key infrastructure

(PKI) is crucial for them to connect to the right server. A weak or vulnerable PKI can lead a client to falsely believe an adversary-chosen software package is the requested software package, significantly undermining the integrity of the update system. That is, without a reliable naming system, any update system will eventually fail [52] as a faulty, vulnerable naming system can be exploited by an attacker to trick a user to believe a malicious software package is the software the client requests, leading it to install malicious software binaries without knowing.

2.2 Traditional security mechanisms

Traditional update systems mostly focused on securing the delivery channel between the server and the clients and the contents to be delivered on this channel. Existing systems achieved such properties by leveraging transport layer security (TLS) and digital signature algorithm (DSA). Although both TLS and DSA work as intended, the risk of key theft still remains. To address this problem, some studies such as the update framework (TUF) [52] started focusing on using multiple keys for multiple different roles, making the system resilient to key compromises such that unless adversaries compromise many keys the system's integrity remains intact.

2.2.1 Transport layer security

TLS is one of the popular methods that is being used for authenticating the communicating party. Many software update systems use TLS to authenticate the contents (e.g., update binaries) they are retrieving by verifying the public key certificate of the server they are connected to [45]. However, the security of such systems is entirely dependent on the security of a PKI which warrants the integrity of the certificate. As a result, clients requesting software updates put their trust into a set of trusted root CA certificates, exposing themselves to the above mentioned PKI risks.

2.2.2 Digital signature algorithm

Software update systems employ DSAs to make sure that the contents of the software update have not been tampered. Many dependency management systems use this technique to enable clients to check the authenticity of dependencies or packages they download [15, 1]. Clients obtain the public key of the software update system they are communicating

with either out-of-band or through a TLS channel. For instance, a client who has already established a secure communication channel using SSL/TLS can retrieve the public key from the server and use the key to verify the data that follows after using insecure channels. Alternatively, some implementations hardcode the public key in the clients to avoid the key exchange; such methods, however, significantly undermine the ability to revoke keys in case the paired secret key is compromised.

2.2.3 Multiple Role-Based Keys

TUF is a software update system that relies on multiple role-based keys [52]. Instead of using a single signing key, TUF uses multiple keys for different roles in the system. In this case, adversaries must compromise all keys or a combination of keys to effectively compromise the system and lead clients to install malicious software. According to TUF, compromising many keys is not only difficult but also necessitates a time window that an attacker has to undergo. TUF expects the software update systems can leverage such time window to detect and respond to the attack in a timely manner.

2.3 Decentralized Security Solutions

Centralization is one of the critical factors which impairs the security of a software update system. In a traditional centralized system, the update server is the only source of information for clients to retrieve and verify update records. That is, clients can only authenticate the retrieved data based on the proofs or cryptographic keys provided by the update server. Considering the high probability that update systems, or more precisely the keys used to sign the updates, can be compromised, trusting a centralized server can be extremely dangerous.

Alternatively, one can decentralize trust by distributing data or the proof of data authenticity across multiple servers and use them to verify updates. This way, even if the centralized server fails, clients can still verify the updates based on the replicated data.

2.3.1 Blockchain

A blockchain is a continuously growing list of records, called blocks, which are linked and secured using cryptography. Each block in a blockchain stores a cryptographic hash of

a previous block, making the blockchain resilient to modification or forgery as altering a record in a block changes the hash of the block which consequently requires modifying all subsequent blocks.

Blockchains are ideal for building a secure append only ledger that is easy to distribute across a network of participants. Although not all blockchains are decentralized, we assume that the blockchains referred to here employ a suitable consensus algorithm, such as HashCash-like proof-of-work (PoW) [18] algorithms including Nakamoto consensus [46]. The latter is designed to allow numerous participants to agree on a block each time new records are added to the chain. Once configured with an appropriate consensus algorithm, a blockchain can replicate blocks and maintain the same state across the network of participants, functioning as a secure, decentralized, append-only ledger.

We believe blockchain technologies are suitable to decentralize and securely store the proofs of software update records. For example, Bitcoin, one of the most popular blockchain cryptocurrencies in the world, is the largest blockchain network with a hash rate 53,994,000 TH/s[1] [3] and more than 9,500 full nodes[2] [9]; that is, any data written to Bitcoin will be captured in a block and replicated to more than 9,500 nodes, and, to revert a block, an adversary must have more than 50% of the network hash power. Generally, anything written in a Bitcoin block is considered irrevocable and immutable unless the block is in the wrong branch. The same property holds for all blockchains with a similar replication factor and hashing power. If we store the proofs of update records such as cryptographic hashes in a sufficiently large blockchain, clients can leverage such proofs to authenticate update data retrieved from an update server.

2.3.2 Software Update Systems Using Blockchain

A few works leveraged blockchain technologies to build a secure update system. Nikitin et al. developed CHAINIAC [49], a system for software update transparency with a decentralized validation process. Software developers publish a new update by creating a hash tree over a software package and the corresponding binaries. Developers then sign the tree root and submit the signed roots along with the package release to co-signing witness servers known as *co-thority*. The witnesses requires (1) a threshold of valid developer signatures and (2) a valid mapping between source and binary to approve the package for release. The witness servers will collectively sign the approved release.

[1] As of Aug 12, 2018. The network hash rate is a cumulative hash power of all miners.

[2] As of Aug 12, 2018. A Bitcoin full node is a node which has replicated all blocks from the genesis block to the latest block. A full node is not necessarily a miner.

To maintain the history of releases of a package, CHANIAC employs skipchains, novel data structures combining skiplists and blockchains. Skipchains allow clients to efficiently navigate arbitrarily long update timelines, both forward (new releases) and backward (past releases).

It is significant that skipchains leverage the structural advantage of blockchains and resolve the limitation of traditional blockchain design by introducing forward signature pointers. However, although CHAINIAC has decoupled the validation process by introducing the witness servers, the outcomes— skipchains containing validated update releases — are stored in a single datastore. While it may be possible to decentralize skipchains by spreading the blocks over multiple nodes which form a private blockchain, the level of security and decentralization provided by such a private blockchain is limited compared to existing large blockchains such as Bitcoin and Ethereum.

2.3.3 Blockchain-based naming systems

Building a software update system is, in fact, analogous to building a naming system that keeps records of software package releases. Software package names must be bound to cryptographic identities (e.g., public keys) in a trustworthy way, and all changes to a name (i.e., software release) must be made under the authorization of the name owner (i.e., the person or the organization that controls the private key paired with the public key registered to the name) and be securely auditable by anyone who wishes to look up the name and accompanying changes. This is what a naming system is supposed to do. Thus, if we have a secure naming system, we can use it to build a secure software update system.

Unfortunately, building a secure naming system has proved to be difficult. An ideal secure naming system should keep names secure, distributed, and human-readable. However, *Zooko's triangle* [39] illustrated that simultaneously achieving all three properties is infeasible. Most traditional systems, like hierarchical PKIs used in TLS and S/MIME, gave up the distributed property and achieved security and human-meaningfulness by introducing trusted third parties, such as certificate authorities (CAs) or key servers. Although such centralized systems repeatedly caused problems due to compromised or incompetent trusted parties [29, 48], Zooko's triangle seemed to imply that there could be no better alternatives.

Namecoin

The advent of blockchains effectively ended Zooko's triangle. Though the first blockchain, Bitcoin [46], was initially conceived only as a cryptocurrency for financial purposes, its first ever fork, Namecoin [14], pioneered the idea of building a secure naming system by encoding name-value pairs inside a blockchain. The blockchain is used as an append-only global log of state transitions to provide consensus on the mapping of names to cryptographic identities without trusting any third party; any changes to name-value pairs are broadcast in new blocks and appended to the globally replicated log. Several newer blockchain naming system designs, such as Certcoin [34], follow the same general design.

However, though Namecoin-like designs bring significant security improvements, they also face many new challenges. In Namecoin-like systems, all nodes in the network must synchronize and validate a complete copy of the blockchain. This requires anyone who wishes to look up names in a secure fashion to afford large, linearly-increasing storage costs, directly impacting the usability of the system. Another problem of Namecoin-like systems is security. Inherently, without a massive user-base, blockchains employing PoW algorithms are vulnerable to the 51% attack [58]. As Namecoin-like systems generally have significantly fewer miners compared to popular, big blockchains, the likelihood of 51% attack taking place is higher than that in Bitcoin or Ethereum [17]. Finally, adding any new features to a Namecoin-like system is very difficult, as the network participants must all agree to run a newer version of the protocol within a short period to maintain the distributed consensus.

Blockstack

Newer blockchain-based PKIs such as Blockstack [17] do attempt to mitigate the issues observed in Namecoin by introduing *virtual chains*. 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.

CONIKS-based approaches

EthIKS [24] is based on CONIKS, a transparency-based PKI combining a centralized key server with semi-decentralized auditors and monitors to reliably detect malicious behavior

by the key server, rather than proactively prevent them from happening [42]. EthIKS inherits the benefits of CONIKS centralization in maintaining the namespace and leverages Ethereum smart contract to guarantee self-consistency of the system, keeping names secure with the same level of security Ethereum offers. However, EthIKS, though not requiring its own blockchain, is tightly coupled to the Ethereum blockchain in using its platform-specific Turing-complete smart contracts and built-in key-value storage system, making the concept practically impossible to generalize to other blockchains.

Catena [56] is an approach that embeds CONIKS name records in an UTXO(unspent transaction output)-based blockchain such as Bitcoin and forms a list-like data structure called transaction chain, enabling clients to traverse name records without the need for scanning all blocks. Though it inherits all good properties of CONIKS while bootstrapping trust onto a blockchain in a less dependent way, Catena is still a passive security solution which requires auditors and whistleblowing, remaining short in terms of decentralized trust. It is apparent that a new naming system, achieving fully decentralized trust over a blockchain and providing a secure thin client while maintaining blockchain neutral property, is still needed.

Conifer

Conifer [32] is a recent blockchain-based naming system that solves the problems identified in all predecessors by providing a secure thin client with fully decentralized trust in a blockchain neutral way. By introducing the concept of name administrator (NA), a minimally trusted entity with centralized control, Conifer provides a comparable performance to that of centralized naming systems while keeping trust fully decentralized over any public blockchain.

The basic idea of Conifer is to separate control and trust. All name bindings are stored in an NA controlled datastore, but the metadata of the bindings are stored in an underlying blockchain. This way, clients who wish to query names and relevant changes from the NA can validate the received data using the proofs stored in the blockchain. A reference implementation of Conifer using Bitcoin successfully demonstrated all of the properties mentioned above. As Conifer allows to implement various policy enforcements, it is possible to build an application on top of Conifer which requires a secure naming system.

2.3.4 Need for SURF

Traditional software update systems tend to protect themselves by protecting the communication channels and providing cryptographic signatures of the updates. However, it is evident that they all collapse once the keys get compromised. Although TUF remedies this problem by requiring multiple keys and separating roles, the update system itself and trust are still centralized.

CHAINIAC opens up a new possibility for building a decentralized software update system by distributing trust over multiple co-thority nodes, a group of witnesses co-signing updates. However, it is still possible to compromise most of the co-thority nodes at once given that adversaries have sufficient resources since the degree of decentralization is far less than that of existing public blockchains such as Bitcoin.

Blockstack and CONIKS-based approaches attempted to solve the problems of existing blockchain solutions by reducing the amount of data to be verified or by introducing the concept of auditors. However, they either lack thin-client support or locked in a specific platform, hindering their adoption.

Clearly, there is a need for a highly decentralized software update system which can leverage any blockchain and enable a wide range of clients to query and validate the history of software releases even if the system is completely compromised. SURF aims to achieve this, *i.e.*, providing a secure update system, by leveraging Bitforest [31], a successor of Conifer which improves usability and namespace management by introducing a novel data structure *index tree*.

Chapter 3

SURF

In this chapter, we discuss (i) the security model that SURF aims to achieve, (ii) security assumptions of SURF, (iii) high-level descriptions on how SURF-based update systems maintain software package updates and create verifiable update records called *operations*, and (iv) how SURF clients securely authenticate operations. We also discuss post-quantum cryptography integration to improve the security of SURF against attacks leveraging quantum computers. We finally describe *Tofino*, our Bitcoin-based SURF reference implementation and discuss SURF can be ported to Ethereum to showcase its blockchain-agnostic ability.

3.1 Security Model

It is often difficult to measure the security of a system or define a set of criteria that a system needs to satisfy to be considered secure. However, it is important to explicitly define what makes a secure system with precise requirements. To answer this question, SURF inherits the security model of Bitforest [32, 31] and defines a secure software update system as an instance satisfying the following attributes: *identity retention* and *policy enforcement*. Because update systems are analogous to naming systems in maintaining software names and changes (i.e., updates), an update system holding the same set of properties which makes a naming system secure can also be considered secure.

- **Identity Retention**: Identity retention is the inability for anybody to impersonate an identity already registered to somebody else [35]. It is a property enforcing that any bindings in the namespace can be changed given the authorization by the name owner, and preventing adversaries from deceiving clients to accept forged bindings that disagree with the authentic binding registered to a name.
- **Policy Enforcement**: Policy enforcement, in general, refers to the external enforcement of system- or service-specific policies governing how names are registered. An example of policy enforcement is identity-based certification where having a name in a namespace requires real-world evidence such as business registration. The possibility of implementing flexible policy enforcement is a key element for building a software update system as different businesses require different external security policies.

Simply put, identity retention enforces that only name owner can make changes to a name, and policy enforcement ensures that only the service provider or the system administrator can make changes to the system such as adding new names and making changes to names. From the security point of view, preserving identity retention is far more critical than providing policy enforcement as violations of identity retention can result in a severe problem as it compromises the foundation of secure communication. In the meantime, providing well-defined policy enforcement attracts businesses and developers as it helps them to implement proper policies that can screen potentially malicious users based on external properties such as real-world identities. Such two properties may not sound compatible at first glance. Nonetheless, it is possible to enable strong policy enforcement while establishing secure identity retention by leveraging digital signatures. Bitforest has achieved both identity retention and policy enforcement via public key cryptography, providing an efficient method to implement a secure naming system. SURF extends Bitforest

and inherits its security model. By providing a concrete implementation of Bitforest with modifications such as lightweight client supports and quantum resistance, SURF aims to achieve a secure, practical software update for wide-range of computing devices.

3.2 Architecture

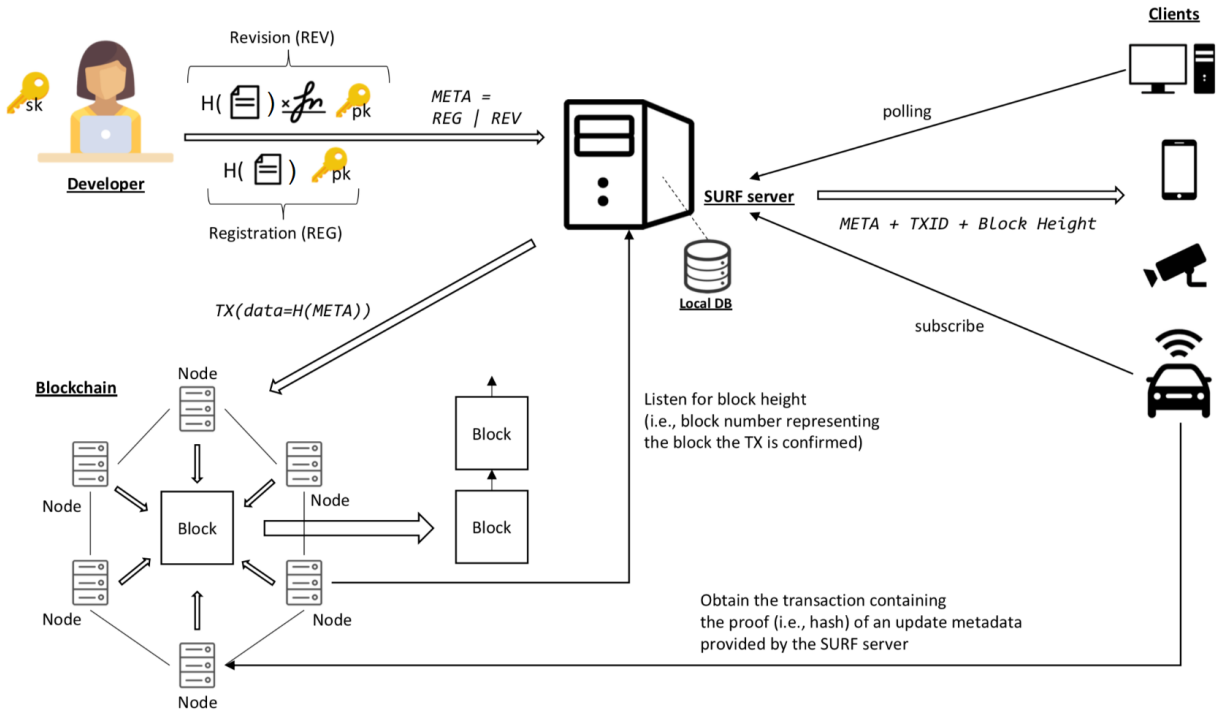

Figure 3.1: SURF architecture and stakeholders

The goal of SURF is to allow developers to build a software update system implementing the SURF security model given the presence of powerful adversaries defined by the threat model in Section 3.3. Figure 3.1 illustrates relationships among SURF components: *developer*, *server*, and *client*. In SURF, there are two types of users: developers and clients; developers register software packages and add updates via a SURF server; clients request proofs for software updates to verify the requested updates. SURF server is the one, as an intermediary between users and the underlying blockchain, delegating user requests (e.g., developer requests to add a new software update, client requests to verify

software updates). The following bullets describe the roles and expected behaviours of SURF components.

- Developers submit either (i) the metadata of an original software package or (ii) the metadata of an update binary. We call the metadata of an original software *registration* and the metadata of an update binary *revision*. A valid registration creates a placeholder for a list called an *update chain* in the SURF server; revisions can be added to the corresponding update chain. Note that SURF does not host update binaries; developers and manufacturers can upload binaries to their preferred repositories such as Github, SourceForge, or their own infrastructure.

- SURF server or *server* runs Bitforest, a blockchain-based PKI that can securely map a name to a cryptographic identity. SURF leverages Bitforest to manage registrations for developers and adds revisions to update chains. Only the registration owner (*i.e.*, developer) can add revisions to his or her registrations; the ability to add revisions must be proven by cryptographic proofs verifiable by developer submitted public keys. Secure hashes of submitted registrations and revisions will be embedded in blockchain transactions and broadcast to a blockchain.

- SURF client or *client* can be thought of as an end-user software using SURF client library or Software Development Kit (SDK) which defines the SURF protocol and provides APIs allowing end-users or devices to subscribe, fetch, and verify software updates. Clients obtain registrations and revisions along with cryptographic proofs submitted to the SURF server and verify them using the hashes embedded in a blockchain. Blockchain transactions are considered immutable and so are the embedded data; clients can reproduce hashes of obtained data and compare them with embedded hashes.

In summary, blockchain in SURF acts as a decentralized, append-only ledger containing secure hashes of registrations and revisions. SURF server embeds hashes of software registrations and revisions in blockchain transactions and broadcast to blockchain nodes. Once the transactions are confirmed in blocks, the embedded hashes become distributed and immutable. The information required for retrieving update related transactions, such as transaction IDs and block heights, must be kept by the server. Clients obtain registrations and revisions and verify them by reproducing and comparing hashes with the blockchain embedded hashes.

3.3 Threat Model

Before we dive into the details, we need to define an explicit threat model that SURF assumes. SURF is designed to function in an unreliable, possibly adverse network. That is, it is entirely possible for an active adversary to control the network and return falsified information. For example, a hostile Internet Service Provider (ISP) may respond to client requests with malicious software pretending that it is the legitimate software update. While SURF provides end-to-end secure communication to defeat such threats, there are few assumptions it makes to provide proposed security guarantees.

- **Trusted bootstraping**: SURF server has data store (e.g., database) containing software updates where each element in the data store corresponds to a blockchain transaction. Given a SURF server, we assume that clients connecting to the server can securely obtain the unique identifier of the server, which is the transaction ID of the first element of the data store.
- **No double spending**: We assume that the adversary is unable to attack the underlying blockchain of SURF in such a way that two transactions which send the same money to different destinations both appear to be valid. It is generally accepted that the distributed consensus algorithms of popular public blockchains such as Bitcoin and Ethereum are highly robust to any adversary with a reasonable amount of resources, and double spending by the adversary is unlikely to succeed.
- **Compromised SURF Server**: A SURF server can be compromised to break identity retention. That is, a compromised server can fool clients into associating an existing software registration to an adversary-chosen value without noticing the owner of the software. Nonetheless, this does not mean the data stored in the blockchain can be modified without the owner's consent.
- **Secure verification of transaction existence**: We assume that there is a way for lightweight clients not participating in blockchain replication to confirm that a given transaction exists in the blockchain. Furthermore, we assume that the adversary is unable to fool a client to falsely believe that a transaction exists on a blockchain.

3.4 Software Developers

In SURF, developers submit formatted metadata of software packages. There are two types of metadata: registrations and revisions.

3.4.1 Registrations

A developer can register his/her software package in the SURF server by submitting a software registration, a JSON object containing a unique software name, a secure hash of the software package, and an *identity script* including a set of public keys that can later verify signatures provided with revisions. Once the registration is confirmed in the underlying blockchain, the registration becomes secure, and SURF server allows the developer to add revisions to his or her registration.

```
{
   "name": "software_xyz",
   "version": 1,
   "data": "00000001[SHA-256 hash of the original software package]",
   "idScript": {
      "n": 2,
      "m": 3,
      "scripts": [
         { "type": "ED25519", "pub_key": "[pub_sig_key_alice]" },
         { "type": "ED25519", "pub_key": "[pub_sig_key_bob  ]" },
         { "type": "ED25519", "pub_key": "[pub_sig_key_choi ]" }
      ]
   }
}
```

Listing 1: An example of a registration with (2,3)-quorum identity script

Listing 1 describes an example registration that a developer may submit to a server for the first time registering a software package called "software_xyz". A detailed explanation of the fields is as follows:

- **name**: A string representing the name of the software package being registered. Name string must be unique.
- **version**: A numerical value for the update version. A revision added to the registration must have a version number higher than that of the previous version.

- **data**: A base-16 encoded string where the first 4 bytes provide version number and the remainder is a hash of the software package binary.
- **idScript**: An identity script which defines a quorum of required signatures that must be provided by revisions to be added to this registration.

Identity script is a simple stack-based script defining a quorum of signatures. The developer includes an appropriate identity script in the registration with public keys representing people who can authorize future revisions. For instance, the identity script presented in Listing 1 is a (2,3)-quorum identity script which returns true only if at least two signatures, generated by secret keys paired with the registered public keys, are provided. That is, any revision follows after the example registration must satisfy the quorum by providing at least two valid signatures.

3.4.2 Revisions

A revision is a JSON object containing formatted information describing a software package update with cryptographic proofs authenticating the validity of the revision. Registrations and revisions share a similar data format except for minor changes. Listing 2 describes an example revision that a developer may submit to a server for the registration presented in listing 1. A detailed explanation of the fields is as follows:

- **name**: A string representing the name of the software package corresponding to this revision.
- **version**: A numerical value for the version which is higher than the version presented in registration or a previous revision.
- **data**: A base-16 encoded string where the first 4-byte is version and the remainder is a hash of the software update binary.
- **signatures**: A list of signatures obtained by signing data, verifiable by registered public keys.

The revision shown in Listing 2 is valid and can be added to the registration shown in Listing 1 as it provides two signatures derived from secret keys which are paired with public keys presented in the registration (*i.e.*, *pub_sig_key_alice*, *pub_sig_key_choi*). By requiring signatures that can only be provided by the registration owner, SURF ensures only developers can add revisions to registrations, guaranteeing ownership for all software registrations.

```
{
   "name": "software_xyz",
   "version": 2,
   "data": "00000002[SHA-256 hash of software update binary]",
   "signatures": [
      "[signature_signed_by_secret_key_alice]",
      "[signature_signed_by_secret_key_choi ]"
   ]
}
```

Listing 2: An example of a first revision of "software_xyz"

3.4.3 Locating update binaries

Notice neither registrations nor revisions points to update binaries. One may add an URL in each registration and revision to indicate where clients can obtain the binaries. This method could effectively convey location information for update binaries alongside the necessary data to verify updates. However, SURF avoids embedding such data in a blockchain as it is possible that the data residing outside of a blockchain can change at any moment while the information stored in a blockchain cannot be modified (e.g., a file stored at abc.com/file1 can be removed), creating a gap that cannot be concealed. To this extent, SURF recommends delivering update binary locations out-of-the-band with minimal to no data embedded in registrations and revisions.

3.4.4 Changing Identity Scripts

A developer can add or remove registered public keys by providing new identity scripts in subsequent revisions. If a revision does not submit an identity script, it directly inherits the identity script provided in the registration or from a previous revision. A revision can include an identity script using the `idScript` field — the same way as registrations provide identity scripts. Listing 3 shows a possible revision submitted after the one shown in listing 2. This revision provides valid signatures satisfying the previous identity script (*i.e.*, the one presented in the registration since the previous revision did not provide an identity script) and a new (2,4)-quorum identity script including an additional public key along with the previously submitted keys.

SURF can accommodate complex authentication mechanism by encoding nested identity scripts. The example identity script shown in listing 4 enforces an additional authentication approved by a manager in addition to developer signatures. More details on identity

```
{
   "name": "software_xyz",
   "version": 3,
   "data": "00000003[SHA-256 hash of software update binary]",
   "signatures": [
      "[signature_signed_by_secret_key_alice]",
      "[signature_signed_by_secret_key_bob  ]"
   ],
   "id_script": {
      "n": 2,
      "m": 4,
      "scripts": [
         { "type": "ED25519", "pub_key": "[pub_sig_key_alice]" },
         { "type": "ED25519", "pub_key": "[pub_sig_key_bob  ]" },
         { "type": "ED25519", "pub_key": "[pub_sig_key_choi ]" },
         { "type": "ED25519", "pub_key": "[pub_sig_key_dan  ]" }
      ]
   }
}
```

Listing 3: Changing identity scripts

scripts are provided in appendix A.

```
{
   "n": 2,
   "m": 2,
   "scripts": [{
         "n": 1, "m": 1,
         "scripts": [{ "type": "SECP256k1", "pub_key": "[pubkey_manager]"}]
      }, {
         "n": 2, "m": 3,
         "scripts": [
            { "type": "ED25519", "pub_key": "[pubkey_alice]" },
            { "type": "ED25519", "pub_key": "[pubkey_bob  ]" },
            { "type": "ED25519", "pub_key": "[pubkey_choi ]" }
            ]
      }
   ]
}
```

Listing 4: Complex identity script

3.5 SURF Server

The SURF server manages registrations. Specifically, it performs the following tasks:

1. Creating registrations
2. Adding revisions to registrations
3. Encoding hashes of registrations and revisions in blockchain transactions
4. Broadcasting transactions
5. Listening for confirmations to obtain transaction IDs (TXID) and block heights required to locate confirmed transactions
6. Creating operations[1] by combining developer provided data, TXIDs, and block heights
7. Providing operations to clients

[1] An operation is a set of necessary data for clients to verify a software update. More details will be provided in Section 3.5.2

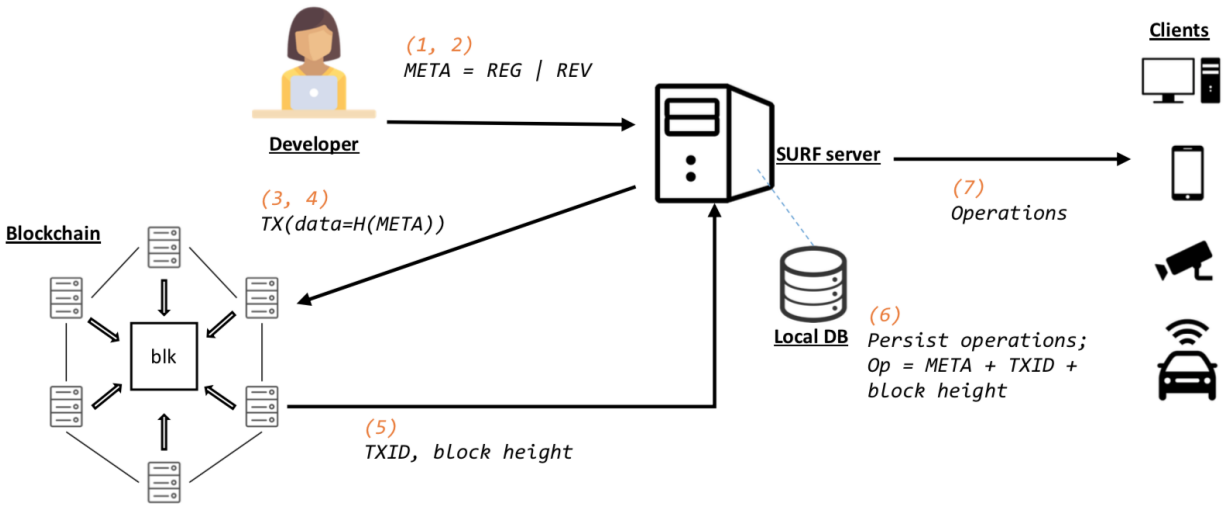

Figure 3.2: SURF server tasks

Figure 3.2 illustrates server tasks and how a SURF server interacts with stakeholders. In this section, we will describe how SURF server (i) manages registrations and revisions, (ii) communicates with the underlying blockchain, (iii) generates verifiable updates records — operations, and, finally, (iv) delivers operations to clients.

3.5.1 Encoding Blockchain Transactions

SURF server creates blockchain transactions to store hashes of registrations and revisions. We call such embedded hashes *proofs* as one can use them to verify the authenticity of data by reproducing hashes. Once transactions are confirmed in blocks, the included registrations or revisions are securely verifiable. To avoid unnecessary cost of locating transactions, the SURF server is responsible for providing blockchain information required to retrieve transactions for relevant registrations and revisions.

Policy enforcement

In order to exercise policy enforcement, SURF server builds abstract data structures called *index trees*, variants of binary search tree containing transactions, and *update chains*, Catena-like lists of transactions [56].

In Bitcoin, two transactions are considered to be chained if the subsequent transaction spends the transaction output of the previous transaction. This property is transitive such that, if A is chained to B and B is chained to C, then A is chained to C. No more than one transaction can be chained to a single transaction output as Bitcoin's *double spending prevention* inhibits a transaction output from being spent more than once [46]. However, a single Bitcoin transaction can have multiple transaction outputs, allowing multiple transactions to be chained to a transaction. Transaction outputs can be created for a particular recipient identified by a Bitcoin address, which can be uniquely converted to a public key. To spend an unspent transaction output (UTXO), the recipient must provide a cryptographic proof — a signature derived from his or her private key that is verifiable by the public key obtained from the address.

SURF server achieves policy enforcement by leveraging Bitcoin double spending prevention. The idea is simple: if a SURF server starts with a transaction — *genesis transaction* — with transaction outputs it controls, then the SURF server has total control in deciding which transactions to be chained assuming all new transactions also have SURF controlled outputs. By defining valid registrations as registrations within transactions that are chained to the genesis transaction, the SURF server can enforce arbitrary policies on registrations as it is the only entity that can chain new transactions. For this reason, SURF server pays fees for all transactions it generates.

Index tree

If we limit the number of transaction outputs per transaction to two, then the overall structure formed by transactions would look like a binary tree. Since each transaction encodes a hash of a software package name, we can compare transactions by hashes, chaining a new transaction by spending the first output if the hash is less than the current transactions' hash; otherwise spending the second output. Accordingly, SURF server builds a binary search tree of transactions — the index tree.

Bootstrapped with a genesis transaction, an index tree manages software registration transactions. For each software registration, SURF server encodes a transaction which (i) embeds the verifiable random hash of the software name and the secure hash of the submitted software package, (ii) spends a UTXO of a previous registration transaction or the genesis transaction, (iii) creates a UTXO for *update chain*, and (iv) creates two UTXOs for later registration transactions. For convenience, SURF refers to a registration transaction in the index tree as a *tree node*. Except for the root node (*i.e.*, genesis transaction), each tree node is a transaction that:

- Spends an UTXO of a parent tree node
- Embeds a verifiable random hash of a software package name
- Embeds a secure hash of a registration
- Creates an UTXO for an update chain
- Creates two UTXOs for subtrees

The root tree node meets all of the above properties except the first one. Data embedding can be done by leveraging Bitcoin OP_RET, a method used to include arbitrary data by adding an unspendable output to a transaction.

The position of a tree node is deterministically decided by the hash value of the name to be inserted and the availability of UTXOs at the time of insertion. Since the hash of a name is randomly calculated by a *verifiable random function* (VRF) [43], tree nodes will be randomly distributed, and the index tree will become reasonably balanced, resulting into a maximum height of the tree in $log(N)$ for N-many tree nodes.

Note that each new transaction requires additional funding to create necessary transaction outputs and cover transaction fees. SURF server can introduce additional funding at any time by adding more transaction inputs to a transaction. Detailed operation cost is discussed in 4.3.

Update chain

Each tree node is a placeholder for an update chain, a data structure designed to contain revision transactions. An update chain is a list abstract data type, similar to the one in [56]. To ensure policy enforcement, transactions encoded for revisions are supposed to spend either the UTXO of a previous revision transaction or the UTXO of the tree node. For convenience and consistency, SURF refers to a revision transaction as a *list node*. Each list node must encode a transaction that:

- Spends an UTXO of a previous list node or a tree node
- Embeds a secure hash of a revision
- Creates an UTXO for a subsequent list node

As discussed in the previous section, all revisions must satisfy identity scripts presented in previous revisions or registrations. Accordingly, adding revisions to a registration (*i.e.*, adding list nodes to an update chain of a tree node) is only possible when provided with valid developer signatures.

The figure 3.3 describes an index tree showing three registrations (*i.e.*, three tree nodes under the root node) with one update chain for a software registration called "vscode", and accompanying blockchain transactions. Since it is located on the right side of the root, we know $H(\text{“}vscode\text{''}) > H(\text{“}surf\text{''})$ holds. Once we look up a tree node, finding an update chain attached to it is easy — you can follow the transaction output prepared for the update chain. In the figure, the tree node with $H(\text{“}vscode\text{''})$ has an update chain with two list nodes indicating two revisions. When we combine all accompanying transactions $(tx0, tx1, tx2, tx3)$ from the root tree node to the list node containing the last revision of "vscode", we form a *transaction chain*, which will be discussed more with details, that can prove the integrity of a software package and its updates.

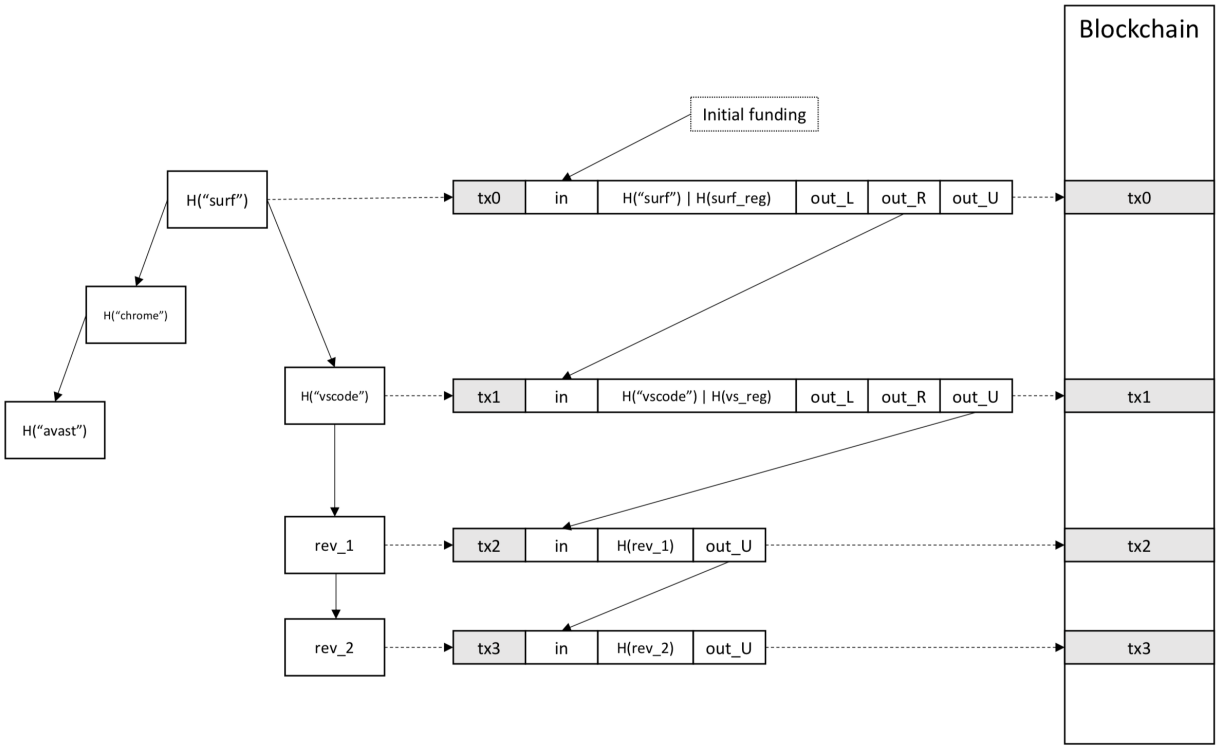

Figure 3.3: An example index tree with transactions

Summary

SURF builds an index tree and update chains to maintain control over the addition of registrations and revisions and, hence, the software update process. SURF spends UTXOs of previous nodes to form cryptographic links that are difficult to forge, chaining transactions. Transaction chains are protected by the double spending prevention provided by the underlying blockchain; unless adversaries successfully compromise the blockchain and break the double spending prevention, they cannot equivocate clients (i.e., showing different transactions stemming from a transaction output). As a result, the SURF server guarantees policy enforcement.

3.5.2 Operations

All nodes correspond to transactions that need to be broadcast to the underlying blockchain for inclusion into blocks, eventually becoming permanent, immutable records. Upon creating a valid node, the SURF server broadcasts the corresponding transaction to one or more blockchain full-nodes and waits until at least one full-node confirms that the transaction has been included in the latest block. Note that it is recommended to wait for some number of confirmations, which is platform dependent (*e.g.*, six confirmations for Bitcoin), to avoid diverged chains.

Once the SURF server receives a sufficient number of confirmations to ensure that the broadcasted transaction is safe, it marks the node and its data (*i.e.*, registration or revision) as 'secure' and creates a record called an *operation* including the blockchain information required for clients to verify the record. An operation must contain the following data:

- **Data**: Original, non-hashed developer submitted data
- **Raw transaction**: Original transaction (hereby 'TX')
- **Transaction ID (TXID)**: Unique hash generated from TX
- **Anchor**: Number indicating the block height of a block where TX is stored

Eventually, for each node in an index tree, there is a corresponding operation. Clients obtain operations from a SURF server and verify whether the data is correct by reproducing and comparing the hash of the data with the proof found in the blockchain transaction specified by the blockchain information provided in the operation.

3.5.3 Operation Log

An *operation log* of a software package is an ordered list of operations required to verify (i) the existence of the registration of the software package in a SURF server, and (ii) the authenticity of revisions registered to the software package. An operation log has two parts: *proof of existence* and *update chain*. Each part serves a different purpose. Figure 3.4 graphically describes an operation log for software package registration "vscode".

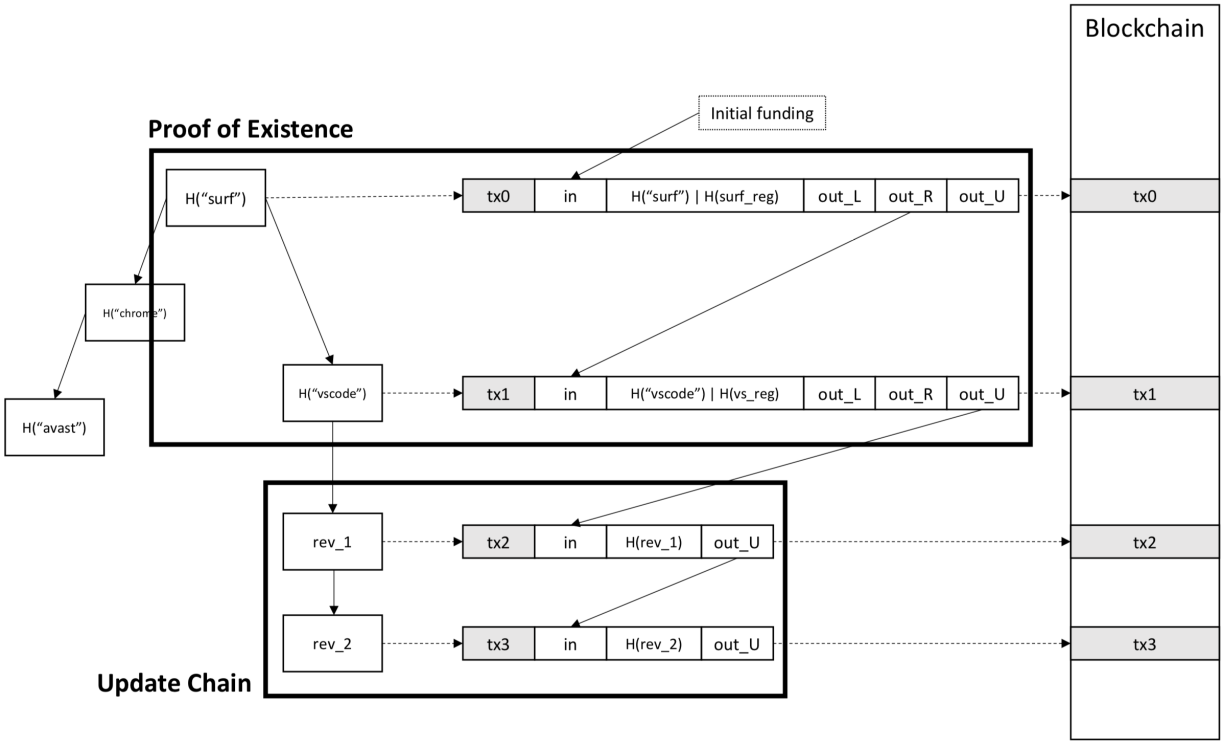

Figure 3.4: An operation log for software "vscode"

Proof of existence

A proof of existence (PoE) of a software package registration x is a list of operations in a tree path p_x that starts from the root, TN_{root}, to a tree node corresponding to x, TN_x. That is, p_x is a tree traversal generating a sequence of tree nodes such that $p_x = (TN_{root}, ..., TN_x)$. If the index tree does not allow duplicates and the hash function used in calculating a tree node value is collision-free, then, for a software package registration x, we can obtain a

unique path p_x by traversing from TN_{root} to TN_x if and only if x exists in an index tree. Construction of such p_x is trivial since an index tree is essentially a binary search tree; we can build p_x by recursively searching for the hash of x in subtrees.

PoE allows a client to securely verify that the software registration does exist in a SURF server; a client can verify the validity of a registration by reviewing transaction outputs presented in a PoE. The SURF server cannot lie about the existence of a registration as tampering with the operation log violates double spending prevention. However, a compromised SURF server can still hide registrations by not providing operations. We will discuss more security caveats in Section 4.1.

Update chains

The rest of the operation log is for the update chain. Operations come after the PoE convey information for revisions and related transactions. Once the PoE is verified, a client can verify the rest of the operations from the end of the log back until it reaches the registration. Detailed verification mechanism will be discussed in Section 3.6.

3.5.4 Update delivery mechanism

Different software updates need different delivery methods. Security patches need to be delivered immediately; updates for new features can be delivered within hours; minor fixes such as removing non-feature related bugs can be introduced within days. To accommodate different needs, SURF provides three update delivery modes — *poll*, *publish/subscribe*, and *push*.

Poll-based delivery

For intermittent software updates, polling can be an effective delivery mechanism. Using polling mechanism, clients can regularly check the mediator for new updates with decreasing polling frequency after each unsuccessful trial.

The upside of polling is fault tolerance. Since polling retries when a request fails, clients using polling will eventually receive an update even if they are deployed in an unreliable network. Within a maximum interval (*e.g.*, 24 hours) and random jitter, polling can keep clients reasonably up-to-date while effectively distributing polling requests in time.

The downside of polling is unnecessary overhead where unsuccessful requests (i) consume power, and (ii) waste SURF server's network bandwidth. While a SURF server can avoid this problem through active caching by deploying CDN services, the power consumption problem remains. Thus, polling is recommended for those devices needing intermittent non-mission-critical updates with a reliable power source (*e.g.*, plugged into a power source, equipped with a large battery).

Publish/subscribe-based delivery

For immediate software updates, publish and subscribe (pub/sub) model is a more appropriate delivery mechanism. Once subscribed to a given software update, clients can promptly receive update notifications from the SURF server (publisher) which triggers clients to download new operations.

The upside of pub/sub is short latency. Compared to polling which can delay updates, pub/sub triggers clients to download updates as soon as they become available. Time-sensitive and mission-critical applications pub/sub delivery to eliminate unnecessary delay between the time an update becomes available and the time it gets reflected in client devices.

The downside of pub/sub is the necessity of a reliable connection and the infrastructure cost. Pub/sub requires clients to maintain bi-directional connections with the pub/sub middlewares. For devices that (i) are deployed in an unreliable network or (ii) have intermittent connectivity, pub/sub may not be appropriate as it wastes resources to establish and maintain connections. Cost of running infrastructure can vary. However, it can be costlier than running CDN services since publishers need to maintain stateful connections that consume memory space.

Push-based delivery

Push is the ideal solution for software update systems. If push is enabled, neither stateful connections nor periodic polling are required; the SURF server can send updates to clients directly using their registered addresses.

Due to the depletion of IPv4 addresses, the only effective way that arbitrary devices can expose themselves on the Internet is to obtain IPv6 addresses. However, IPv6 deployment is an on-going process and the current penetration rate is less than 25% [10]. While it is possible to implement a push service relying on tunneling such as 6to4 or Teredo, it is beyond the scope of this work.

3.6 SURF Client

3.6.1 Update Verification

Suppose we want to perform update verification for a software package x. The goal is to verify the operation log of x, OP_x, to determine if the revision indicated by the last operation in OP_x is correct.

A naive implementation of this procedure is to download an entire OP_x at once and process operations one by one. For each operation o_i in OP_x, clients first need to check whether the transaction, tx_i, indicated by o_i is in the claimed block. Checking the existence of tx_i in a particular block can be done using Bitcoin Simple Payment Verification (SPV) [46]. SPV enables clients to query a Bitcoin full node (*e.g.*, miner) to determine whether a transaction exists in a specified block at which time the full node returns a *merkle branch* for the queried transaction. A Merkle branch is a sequence of hashes that are necessary to reconstruct the root of the Merkle tree of TXIDs presented in the block header [19]. Since the size of a Merkle branch is proportional to the logarithmic of the number of transactions included in a block, clients can quickly calculate the root of a Merkle tree given a non-empty Merkle branch.

Note that clients must provide a TXID (what trasnaction to look for) and a block height (where to look in the blockchain) to query a Merkle branch. TXID, if not provided, can be easily derived from the raw transaction. However, without the block height, the client must linearly scan through all blocks to find the target transaction, wasting bandwidth and time. For example, assuming block heights are unknown, to verify an operation log, a SURF client needs to download all block headers starting from the block containing the genesis transaction to the block containing the last operation in the operation log. Regardless of the number of operations in the operation log, the amount of header data a client needs to download for a single verification can be at least 6.68 MB (87,600 block headers; 80 bytes per header) in the best case (*i.e.*, the last operation is found in the last block) if the log has operations dispersed in time where the time difference between the first operation and the last operation is one year.

Figure 3.5 shows an example index tree containing a registered software package "vscode" with two revisions. In order to verify the operation log for software package "vscode", clients must download all block headers starting from `block_k_hdr` to `block_k+5_hdr`. Although downloading hundreds of megabytes is not a problem for desktops with a reliable network connection, it can be problematic for smartphones with a small data plan or worse for IoT devices with small flash memory.

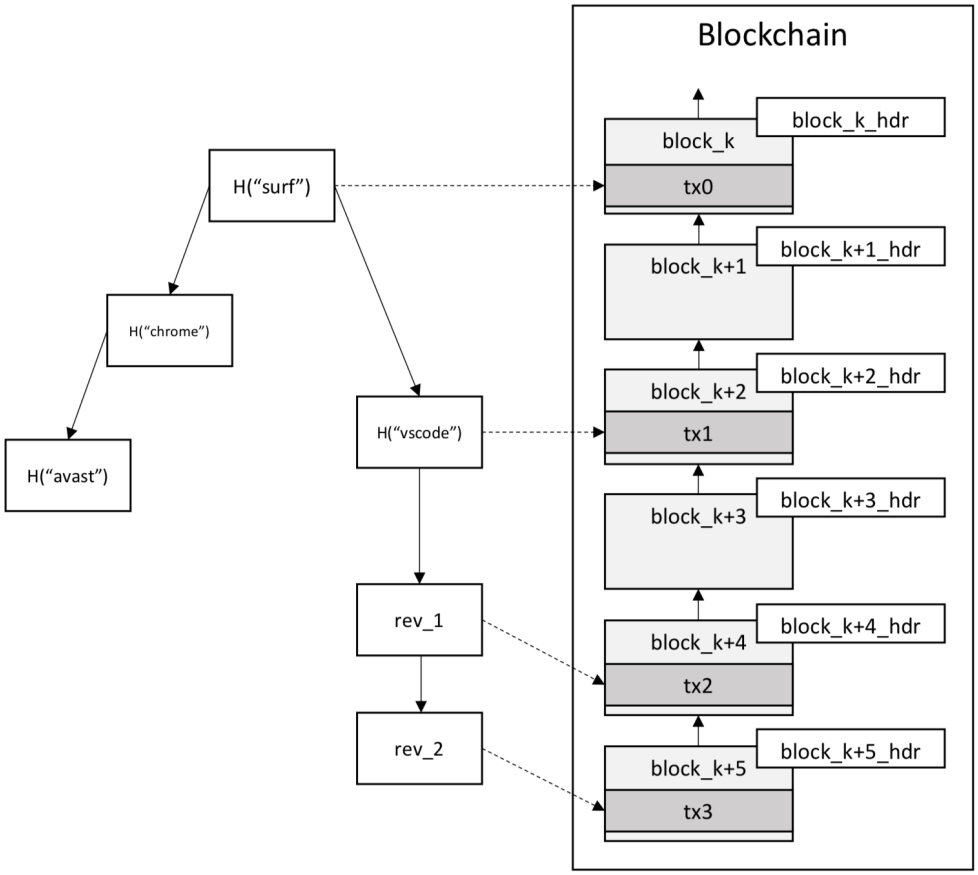

Figure 3.5: Blockchain headers for SURF operations

To reduce bandwidth cost, the SURF server provides `anchors` in operations, allowing clients to pinpoint blocks. One may argue that there is a security risk in downloading selected block headers since it is possible that a blockchain node colluding with a compromised SURF server can show fabricated blocks. Nonetheless, since transactions are securely chained, and all clients expected to verify whether transactions are in blocks using headers obtained from multiple sources (*e.g.*, multiple Bitcoin full nodes), skipping blocks would not degrade the security of SURF . For this reason, SURF recommends clients to wait for some number of confirmations, which is platform dependent (*e.g.*, six confirmations for Bitcoin), to ensure block headers they obtained are consistent. Figure 3.6 shows the use of anchors for the same operation log illustrated in Figure 3.5. Notice that, in the worst case, we only need to fetch four headers, one header for the last revision and three headers indicated by preceding transactions, to thoroughly verify the operation log

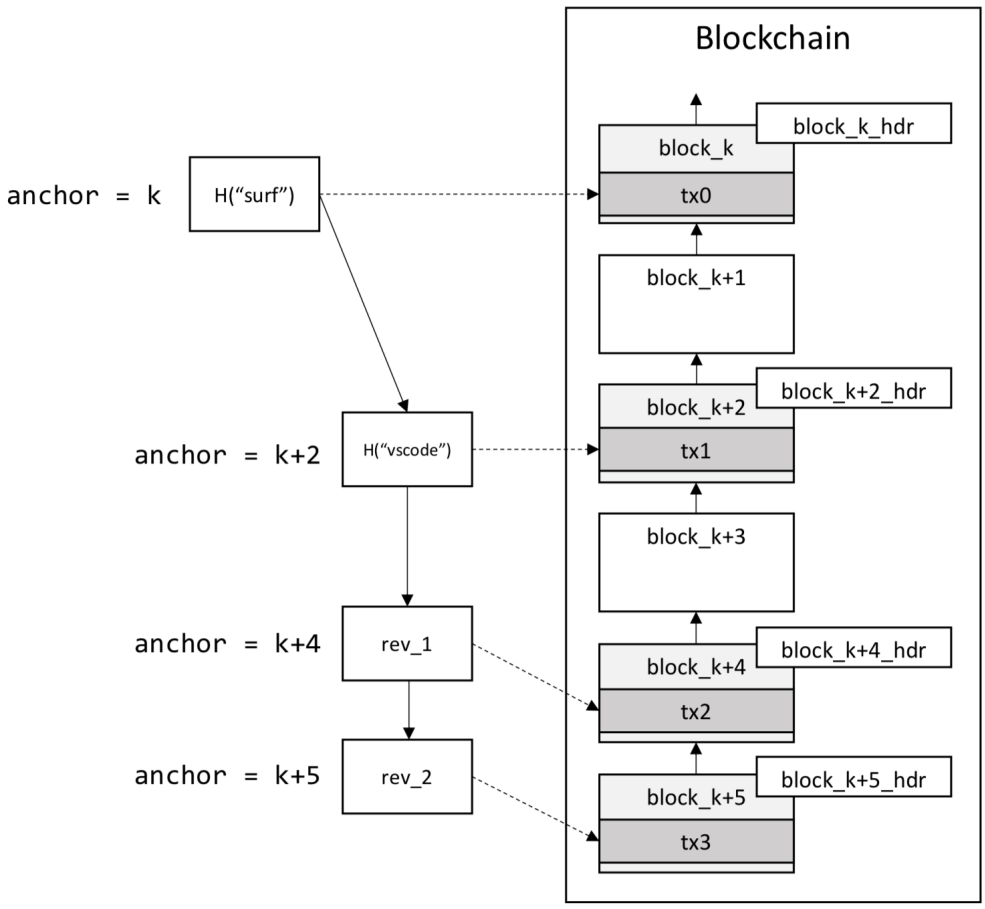

Figure 3.6: Anchor example

for software package "vscode".

Once SURF clients confirm the transactions are in the claimed blocks, they need to check whether each transaction has spent the correct output of a previous transaction, forming a valid chain. Validating cryptographic links is necessary to confirm that the transactions are indeed encoded by the SURF server providing the operation log. Note that if a client successfully verifies all transaction chains derived from OP_x, then we can conclude that the PoE in OP_x is verified.

Following the verification of the transaction chains, SURF clients can now verify the integrity of revisions. They reproduce and compare hashes of data observed in operations to the hashes stored in transactions. This step proves that the data provided by the

operation log is the same data submitted by the developer who owns the registration. If the hashes match, then we can run identity scripts presented in the revisions to validate signatures.

The purpose of running identity scripts is to check whether the data provided in the revision has been altered since the registration, breaking ownership. In case the data has been changed, identity script will always return false since signatures included in the revision are obtained from the data.

In summary, for each operation, from the tail to the head of the operation log, the following verification tasks are performed:

- The transaction is in the claimed block.
- The transaction has spent the correct output of the previous transaction.
- The transaction has the correct hash of the current operation's revision.
- The operations signatures satisfy the previous operations identity script.

If any of the above tasks fails, then a client can conclude that the operations are not valid. Additionally, knowing that all operation logs start with the root node of the index tree, clients must check if the last operation (*i.e.*, head of the operation log) in the verification process has a matching transaction with the root transaction of the index tree.

3.6.2 Enhancing Update Verification

Challenges

Software update is essential to all software-powered devices ranging from enterprise servers to low-powered devices such as smartphones, tablets, wearable devices, and embedded computers. The recent rise of the Internet of Things (IoT) adds even smaller, less powerful devices to the list while emphasizing the need for a truly secure software update system. However, providing secure software updates to low-powered devices can be challenging as they come with extremely limited resources. Unlike desktop computers, small devices lack in many aspects: CPU clock, memory, storage, and network bandwidth. For instance, in 2018, a reasonably priced desktop computer comes with high-clock multi-core CPU, gigabytes of memory, terabytes of storage, and sufficiently fast network interfaces. Such specification make it possible to run a Bitcoin full node for a length of time with no

difficulties. However, this is not possible with smartphones and IoT devices. Due to power consumption and small design, portable computing devices and embedded systems are generally equipped with single- or dual-core CPUs and tend to be limited in memory, storage, and network capacity.

For SURF, limitations mostly come from the need to download and store data. Although SURF does not require clients to download a lot of data initially, the data required to verify software updates — operation log — linearly grows over time, eventually becoming too large for small devices to hold indefinitely. Considering a reasonably small IoT device comes with few megabytes of storage while each operation can be arbitrarily big if not regulated by a mediator, storing such a linearly growing operation log per software package makes SURF impractical.

There is also a memory limitation that SURF has to overcome when supporting resource-constrained devices. Recall that update verification is the process of verifying operations. Clients download an operation log and iterate over the log to check the authenticity of each operation and the accompanying revision. A naive implementation such as the shown in subsection 3.6.1 may download an entire operation log at once, store the log in the memory, and process operations one by one. However, such implementation can only run on devices with at least few megabytes of memory as each operation verification round requires complex signature validation.

Incremental verification

SURF addresses aforementioned challenges by implementing *incremental verification*. Incremental verification is simple: a SURF client obtains operations one at a time and verifies them in reverse chronological order until it reaches a known, verified transaction called *check point*. The following describes an example of incremental verification:

- Suppose we have a client, C, and a firmware, f, which C wants to update regularly. The developer of f has published three updates and the operation log containing those updates is OP where $OP = [o_1, o_2, o_3, o_4, o_5, o_6]$ and $[o_1, o_2, o_3, o_4]$ represents the PoE of f.
- Let $TX(o)$ be a function extracting the transaction ID portion of an operation, o.
- Initially, C has a check point $TX(o_1)$, the genesis transaction. Then, C needs to verify o_i for $i = [1, 6]$ to successfully authenticate OP since $TX(o_1)$ is the check point. Once OP is checked, C updates its check point to $TX(o_6)$, the transaction of the latest operation it verified.

- Suppose the developer of f publishes a new update and creates a new operation log OP' with o_7. Now, C does not need to verify the entire OP'. Instead, it verifies o_7 and checks whether o_7 leads to an operation o_6 which contains its check point $TX(o_6)$.
- C updates its check point to $TX(o_7)$.

Though straightforward, incremental verification effectively reduces the network bandwidth required for the verification process. Storage-wise, clients do not need to store previously verified data or block headers as they only need 32-byte TXIDs to continue verifying future updates. Furthermore, since we download one operation at a time, the total memory requirement per operation is considerably smaller compared to the case where we download the entire operation log.

```
def op_verify(O, N, last_op_pos, check_point):
   for i=N-1..last_op_pos:
      curr <- download O[i]
      if curr not exists in blocks[curr.anchor]:
         return false
      elif subsequent is null:
         subsequent <- curr
         pass
      # verification process omitted for brevity
      delete subsequent # C-like memory deallocation
      subsequent <- curr
   if curr.txid is check_point:
      return true
   return false
```

Listing 5: Incremental operation verification

Putting it all together

Incremental verification saves substantial network bandwidth, storage cost and memory. Listing 5 shows the pseudo code of improved update verification with virtual anchor and incremental operation verification.

3.7 Quantum-Safe Software Updates

With some exceptions, most of the modern public key cryptography algorithms are based on the difficulty of factoring integers or the discrete logarithmic problem. While integer factorization is believed to be computationally infeasible for an ordinary computer if the number is the product of large prime numbers [40], the recent advances in quantum computing suggests that using Shor's algorithm, both integer factorization, and the discrete logarithmic problem can be efficiently solved by a quantum computer with sufficient number of qubits [41], implying that many of public key cryptography algorithms currently employed will eventually collapse.

Although there are no true quantum computers yet, researchers predict that quantum computers will surpass the limits of classical computers, achieving quantum supremacy in near future [23]. Since public key cryptography underpins the security of SURF , there is a high quantum-risk threatening the foundation of software update system including SURF . Therefore, we designed SURF to be cryptographically agile by supporting different cryptographic techniques including quantum-safe cryptography.

3.7.1 Cryptographic agility

Cryptographic agility is the ability of a system to accept or transition to new cryptographic algorithms at minimal cost. It is a likelihood of a system moving from using one cryptography to another (with improved security) without re-writing much code [21].

The concept is not new and has been in development for a long time. For example, x.509 digital certificate standard, which was released in 1988, was created with crypto-agility, making the standard flexible to accept extensions such as handling version differences or the use of different cryptographic algorithms in generating keys and certificates. However, recent outbreaks of failing cryptographic algorithms (*e.g.*, ROCA [47], ROBOT [22], KRACK [57]) and subsequent need for replacing old vulnerable cipher schemes generated renewed attention for crypto-agility.

SURF is crypto-agile by design in its use of digital signatures. Cryptographic algorithms are represented as plugins in SURF and a new scheme (*e.g.*, signing algorithm) can be added to SURF as a plugin without modifying existing code. Clients can choose cryptographic algorithms and specify them in the identity scripts. Recall that identity script has a field `type`; this field states the type of cryptographic algorithm. Clients can freely choose a digital signature algorithm and provide signatures generated by the chosen algorithm. Listing 6 shows an identity script using three different signing algorithms.

```
{
   "n": 2,
   "m": 3,
   "scripts": [
      { "type": "ED25519",   "pub_key": "[pub_sig_key_ed25519]" },
      { "type": "SECP256r1", "pub_key": "[pub_sig_key_p256r1 ]" },
      { "type": "SECP256k1", "pub_key": "[pub_sig_key_p256k1 ]" }
   ]
}
```

Listing 6: An identity script with key scripts using three digital signature algorithms

SURF does not rely on one cryptography; it uses many of them and can easily support more in the future. Indeed, if and when a public key cryptography that can generate quantum-safe signatures becomes available, SURF can add a new plugin to achieve quantum-safe software updates.

3.7.2 Post quantum cryptography (PQC)

Developing quantum-safe cryptographic algorithms or post-quantum cryptography (PQC) is an active research area. Because Shor's algorithm is only effective for integer factorization and discrete logarithmic problems, public key cryptography algorithms relying on other models are considered safe from Shor's algorithm and, ultimately, quantum attacks.

There are five different approaches to developing PQCs: lattice-based, code-based, hash-based, multivariate, and supersingular elliptic curve isogeny cryptography. Note that this thesis rules out symmetric cryptographic algorithms as it is not relevant in the context of SURF. SURF does not necessarily encrypt data to hide, but rather transparently shows that they are indeed the right data by presenting valid signatures. Table 3.1 briefly describes the five approaches with representative algorithms.

3.7.3 Integration in SURF

Of those PQC algorithms presented in Table 3.1, BLISS-II, GLYPH, and SPHINCS-256 are designed to generate fixed length digital signatures with 128-bit post quantum security level [33, 26, 20]. Our analysis of these PQC algorithms for the purpose of SURF is as follows:

Type	Description	Algorithm
Lattice-based	Based on lattice problems; can be most efficient in terms of key sizes [44]	BLISS-II [33], GLYPH [26]
Code-based	Relying on error-correcting codes which can be used to decode a general linear code; public key size is large (100KB to several megabytes) [50]	Goppa-based McEliece [50]
Hash-based	Based on the idea that a secure digital signature scheme is analogous to a secure, collision-resistant hash function (*e.g.*, lamport signature) [25]; the recent algorithm, SPHINCS-256, can sign multiple times with one private key [20]	SPHINCS-256 [20]
Multivariate	Based on solving nonlinear equations over a finite field (considered NP-hard); relying on multivariate trapdoor functions [30]	Rainbow [51]
Supersingular Isogeny	Based on the conjectured difficulty of finding isogenies between supersingular elliptic curves; although there is a sub-exponential quantum algorithm constructing isogenies between ordinary elliptic curves, in the supersingular case, the same algorithm remains exponential, making supersingular model quantum resistant (SIDH) [38]	SIDH [38]

Table 3.1: PQC approaches and algorithms

- Although extensively studied, BLISS-II does not have a security reduction to a known hard mathematical problem, failing to prove that breaking the algorithm is as difficult as solving the problem. For this reason, we ruled out BLISS-II and all NTRU variant algorithms.

- GLYPH is the best candidate out of the three. It has a security reduction to an NP-hard problem — Shortest Vector Problem (SVP) [26] and generates comparably small digital signatures (public key 2KB, signature 1.8KB). Nonetheless, the algorithm is relatively new and not extensively evaluated by the research community.

- Unlike most hash-based signature schemes, SPHINCS-256 is stateless, allowing it to be a drop-in replacement for current signature schemes used in SURF. Nonetheless, comparably large signatures (41Kb) make SPHINKS-256 less desirable as it

drastically increases the average size of registrations and revisions which clients will download.

SURF can integrate both GLYPH and SPINCS-256, allowing developers to use either PQC as they desire to maintain identity retention. No changes are required in submitting registrations or revisions; there will be two additional public key types that can be used in identity scripts; Listing 7 shows an example identity script using both GLYPH and SPHINCS-256 public keys.

```
{
   "n": 1, "m": 2,
   "scripts": [
      { "type": "GLYPH",      "pub_key": "[pub_sig_key_glyph   ]" },
      { "type": "SPHINCS256", "pub_key": "[pub_sig_key_sphincs ]" }
   ]
}
```

Listing 7: An identity script with key scripts using PQC

3.7.4 Security Against Quantum Computers

Assuming the integrated PQCs are truly quantum-safe, we can evaluate the security of SURF as follows:

- **Identity retention**: Recall that revisions can be only appended when they are submitted with valid signatures. Employing PQCs effectively prevents adversaries with quantum computers from adding malicious revisions to a registration, enforcing only developers with valid private keys can add revisions to registrations. Hence, identity retention is still maintained.
- **Policy enforcement**: Policy enforcement in SURF is protected by the double spending prevention provided by the underlying blockchain, Bitcoin. Since Bitcoin uses SECP256k1, a public key cryptography vulnerable to quantum threats, to prove the ownership of UTXOs, adversaries with quantum computers can break policy enforcement without even compromising SURF server. In fact, the only required information for adversaries to overthrow policy enforcement of a SURF server is a set of public

keys controlled by the server to spend UTXO(s); adversaries can use quantum computers to derive private keys from the public keys and spend UTXO(s), sabotaging SURF server's operations (*e.g.*, spending all UTXOs to disable the service).

Although the lack of policy enforcement does not impact identity retention (*i.e.*, clients still verify registrations and revisions that are made in the past), it discourages service providers who would run SURF servers to make profits (*e.g.*, "Software Update"-as-a-Service). Solving this problem is hard. Because the problem comes from the underlying blockchain, mitigating the threat of losing policy enforcement would require SURF to be deployed on a quantum-safe blockchain. Unfortunately, as of May 2018, there are no sufficiently large quantum safe public blockchains.

However, growing demand for preparing to the post-quantum era suggests there will be quantum safe blockchains in the future. In this perspective, SURF proposes *blockchain migration* to address the quantum threat by allowing SURF to easily migrate from one blockchain to another.

3.8 Blockchain Migration

Although blockchain technology enables applications to decentralize data, leveraging a blockchain generally results in tight coupling between applications and blockchains (*i.e.*, an application is locked in a specific blockchain). While this is not problematic in most cases, as we discussed in the previous section, there is a situation where an application needs to migrate to another blockchain in case the current blockchain lacks features the application requires (*e.g.*, quantum-safe cryptography).

SURF is designed to be blockchain-agnostic. That is, SURF can use any blockchain as long as it provides features required to implement SURF operations. The following section describes the properties of a blockchain required by SURF .

3.8.1 SURF Compatibility Requirements

SURF can leverage any blockchain as its security foundation if it:

1. Has strong double spending prevention.
2. Allows applications to embed arbitrary data.
3. Provides controlled write access.
4. Provides expressive data structures to build index tree and update chain.
5. Provides open read access with verification support.

We call the above requirements *SURF Compatibility Requirements* (SCRs). Any blockchain satisfying these SCRs can be used as an underlying blockchain for SURF. In the following, we explain how the SCRs contribute to supporting SURF.

- **SCR-1**: Double spending prevention is a cornerstone of SURF in which protects identity retention from the adversaries. This requirement is trivial since recent cryptocurrencies and blockchain projects do achieve strong double spending prevention. The remainder of this chapter assumes that all cryptocurrencies satisfy this condition.

- **SCR-2**: This is a core requirement for SURF — recording hashes of developer submitted data (*i.e.*, proofs) in blocks and taking advantage of a secure, immutable ledger. In its current implementation, SURF embeds hashes of registrations and revisions in Bitcoin transactions using OP_RET, and the integrity of embedded data is protected by the Bitcoin network as transactions cannot be forged without compromising the network. Any SURF-compatible blockchain must provide a mechanism that allows the SURF server to record arbitrary data in blocks with equivalent security guarantees.

- **SCR-3**: This requirement is essential for policy enforcement since we want to limit write privileges to the SURF server. For instance, Bitcoin's Pay-to-PubKeyHash (P2PKH) ensures that UTXOs created for addresses owned by a SURF server can be only spent by the server, not anyone else [46].

- **SCR-4**: This refers to the ability to build an index tree and update chain. Having an index tree is essential for SURF to achieve fast lookup; assuming the tree is reasonably balanced, the lookup performs in $O(logN)$ where N is the number of software packages registered. Implementing an update chain using a list data structure is imperative for SURF as such an ordered list of revisions starting with a unique registration clearly represents the software update history. Combined together, the overall time complexity required to verify the latest software update with k-many revisions results in $O(logN) + O(k)$ where N is the number of registrations present in the index tree.

- **SCR-5**: This last requirement is crucial for thin client support; access to the embedded proofs must be open, and the underlying blockchain should provide a mechanism for clients to verify the proofs without downloading blocks. Without this, only a limited number of clients with the capacity to replicate and execute blocks can benefit from SURF.

3.8.2 Different Types of Blockchains

There are two broad categories of blockchains: UTXO-based such as Bitcoin and account-based such as Ethereum. The difference between UTXO-based and account-based blockchains is the way they represent the balance. In UTXO-based blockchains, balance is a sum of UTXO values paid to an entity. On the other hand, in account-based blockchains, balance is a numeric value kept by the system representing the number of tokens that an entity can spend.

UTXO-based blockchains have more expressive, rich transactions than account-based blockchain transactions. For example, Bitcoin transactions can embed arbitrary data in transactions (*i.e.*, OP_RET) while each transaction can have multiple outputs (*e.g.*, transferring funds to multiple parties using a single transaction). Account-based blockchains, however, use somewhat simplified transactions; no data embedding nor multiple transaction outputs; they merely transfer funds from one address to another. They generally provide *smart contracts*, high-level scripts that can be executed based on blockchain activities (*e.g.*, transactions) and conditions observable in blocks (*e.g.*, block heights).

Tofino is a SURF implementation successfully deployed on Bitcoin. Since the majority of existing UTXO-based blockchains, including Litecoin, are the forks of Bitcoin with no groundbreaking changes affecting the aforementioned SURF requirements, it is safe to assume that SURF is compatible with UTXO-based blockchains. Indeed, Litecoin, for instance, satisfies all SURF's SCRs, allowing us to implement SURF with Litecoin. In fact, Tofino can leverage Litecoin with very minor modifications.

Account-based blockchains, most famously represented by Ethereum, do not satisfy all SURF 's SCRs; since their simplified transactions do not allow data embedding nor transaction chaining. In this case, we need to provide a proper smart contract that can securely store and replicate arbitrary data to blockchain participants. Providing such a smart contract is not trivial. In the following, we provide an Ethereum smart contract for SURF to demonstrate it is indeed blockchain-agnostic.

3.8.3 SURF on Ethereum

In this section, we first recall how transactions are supported in Bitcoin and then describe how SURF transactions are supported using smart contracts in Ethereum.

Bitcoin Transactions

Recall SURF leveraged UTXOs to form unforgeable links between registrations and revisions. Bitcoin uses UTXOs to describe one's balance; if Alice has ten bitcoins, then Alice has a set of UTXOs which sums up to ten bitcoins. Spending bitcoins is not very intuitive because it involves destroying old transaction outputs (*i.e.*, making them unspendable) and creating new spendable transaction outputs. For example, when Alice sends one bitcoin to Bob, Alice creates a transaction that (i) consumes a set of UTXOs, U_I, and (ii) outputs a set of UTXOs, U_O. The sum of U_I must be greater than or equal to 1 bitcoin plus the anticipated transaction fee to broadcast the transaction. U_O contains at most two

transaction outputs: one for Bob, and one for Alice if there are any changes. Once the transaction gets confirmed (*i.e.*, included in a block), Alice can no longer spend 'spent' transaction outputs, and Bob will have a new transaction output at his disposal.

Ethereum Transactions

Ethereum transactions are simpler than Bitcoin transactions. This is because, instead of relying on transaction outputs and accompanying cryptographic proofs, Ethereum allows a sender to issue a transaction to a recipient only if such transaction moves a fund that is less than or equal to the balance of sender's account.

Ethereum has two types of accounts: externally owned accounts (EOAs) and contract accounts. The term transaction is used in Ethereum to refer to the signed data package that stores a message to be sent from an externally owned account (*i.e.*, a user) to another account on the blockchain [7]. A typical message carrying Ethereum transaction contains the followings:

- **nonce**: A scalar value equal to the number of transactions sent by the sender.
- **to**: The recipient address of the message.
- **value**: The amount of Wei (a unit of Ethereum cryptocurrency; 1 ether = 1e18 wei) to transfer from the sender to the recipient.
- **gasPrice**: A scalar value representing the fee the sender is willing to pay for one unit of gas. Each Ethereum Virtual Machine (EVM) operation code charges a specific amount of gas to be executed.
- **gasLimit**: A scalar value representing the maximum number of computational steps the transaction execution is allowed to take.
- **data**: An optional field, which can contain an unlimited size byte array specifying the input data of the transaction.
- **v, r, s**: Values corresponding to the signature of the transaction and used to determine the sender of the transaction.

Notice that Ethereum transactions do not have 'transaction outputs' (hence, no UTXOs). For instance, if Alice issues a transaction which transfers one ether to Bob, once the

transaction gets confirmed, both Alice's and Bob's balances change accordingly, without generating transaction outputs.

As described above, such simple transaction scheme cannot fulfill SCR-3 and SCR-4 using the same strategy Tofino uses with Bitcoin. Since Ethereum does not use secure transaction outputs, we cannot ensure ownership of transactions nor create cryptographic chains between transactions, which are essential to building index trees and update chains.

Smart Contracts

Ethereum supports smart contracts, allowing developers to execute arbitrary code and store values in Ethereum blockchain [6]. Ethereum smart contracts (or simply 'contracts') are programs deployed at random addresses. Contracts are considered as accounts just like externally owned accounts and can receive transactions from other accounts. Contracts, or more specifically contract functions, are executed when transactions are sent to the contract address; for each new block, Ethereum full nodes iterate over transactions in the received block and run corresponding contracts for those transactions sent to contract accounts. Arbitrary execution results of contracts are persistent in Ethereum in the form of *states*. Similar to blocks, states are replicated across the network. All full nodes replicating blocks should have the same states as no transaction can invoke a contract more than once, or multiple contracts simultaneously due to Ethereum's double spending prevention. Thus, states are safe from equivocation (*i.e.*, no adversaries show different data to different parties). SURF can successfully migrate to Ethereum by developing a contract using Solidity, one of Ethereum high-level programming languages. The following summarizes how SURF managed to satisfy SCR-2∼4.

- SCR-2 can be achieved by leveraging Ethereum contract `state`. The state is part of the Ethereum blockchain storing contract permanent data. Ethereum full nodes maintain the same states across the network by executing transactions in replicated blocks.
- SCR-3 can be satisfied using `require`. By recording the ownership to contract state, only the contract creator can call functions that can modify the contract state. Listing 12 in Appendix B presents a custom modifier using `require`, limiting the function invocation to a specific sender.
- SCR-4 can be emulated by exploiting `mapping`, hashtable-like data structure provided by Solidity. Due to limitations in Turing completeness, Solidity does not offer

recursive data structure which is essential to traditional binary tree abstract data type. Instead, one may represent a binary tree using an array of tree nodes where each node has references to subtrees using array index. Listing 10 in Appendix B shows how SURF emulates a tree with `mapping`.

Notice that functions managing index tree, and update chains are now entirely off-loaded to an Ethereum smart contract. SURF server still accepts developer and client requests and stores local copies of operations, but adding registrations and revisions are not taking place in Ethereum. Note also that while this does not impact SURF functionality, it sacrifices code flexibility as Ethereum contracts cannot be redeployed without changing addresses. There are ways to seamlessly migrate a contract to a newer version such as using a proxy contract routing transactions to an active contract. However, it is unnecessarily complicated Appendix B provides the full Solidity script used for SURF to work with Ethereum.

Thin Client Support for Secure Verification

SCR-5 is a bit tricky. Clients can obtain state value of a contract by calling contract functions via `web3`, an interface to Ethereum blockchain. Nonetheless, only full nodes who replicate blocks can securely verify the correctness of the latest state by executing transactions sent to the contract. Tofino solved this problem by leveraging Bitcoin SPV, allowing clients to verify operations by validating transactions; a compromised Bitcoin node cannot provide false updates without forging many blocks which requires tremendous hashing power if the number of blocks to forge is high. However, in Ethereum's case, states only contain the final result and clients cannot verify the contract state as it is impossible to replay transactions. Hence, unlike Bitcoin, the Ethereum-based SURF client may accept false operations if the communicating Ethereum full node is adversarial. While this can be considered as a major security setback compared to Bitcoin, clients can mitigate this problem by querying multiple Ethereum nodes to validate operations, overcoming the risk of compromised full nodes. Section 4.1 discusses this security challenge in more details.

Summary

We demonstrated SURF portability to Ethereum by providing a smart contract managing index tree and update chains. As account-based blockchains in general offer equivalent or more expressive scripting languages to accommodate Ethereum-like smart contracts, SURF can also be migrated to other account-based blockchains such as EOS [5] or Hyperledger

Fabric [12] by implementing smart contracts similar to the one developed in Solidity for Ethereum.

3.9 Conclusion

In this chapter, we explored how SURF manages software package registrations and revisions. With strong guarantees in terms of both identity retention and policy enforcement assuming a sophisticated threat model, SURF provides a secure mechanism for clients to safely verify the authenticity of software updates, answering the question 'how does a client know the patch is correct' without trusting a centralized entity. It is important to point out that the security of SURF is derived from that of the underlying blockchain; compromising SURF is as difficult as compromising the blockchain used by SURF . As of April 2018, it is virtually impossible to compromise Bitcoin, the blockchain Tofino uses, without possessing a significantly large hashing power amounting to more than 13 Exahash per second [3].

However, there are attacks that SURF does not fully handle such as *eclipse* [37] and replay attacks. We will discuss such attacks and outline possible remedies in Section 4.1.2.

In summary, SURF provides a secure software update system by accomplishing the followings:

- **Secure identity retention**: Only the registration owner(s) can add revisions and the violation of this property can be easily found by consulting the underlying blockchain.
- **Strong policy enforcement**: The SURF server holds all rights to make changes to the namespace by owning all UTXOs; unless payment keys are compromised, the SURF server can enforce strict policies to all software names.
- **Secure update verification**: Clients do not rely on a centralized trust to verify updates; they obtain operation logs from the SURF server and verify the logs by querying the underlying blockchain.

The following chapter evaluates SURF in terms of security and performance.

Chapter 4

Evaluation

4.1 Security

We evaluate SURF security by demonstrating how SURF mitigates commonly known attacks that are launched against software update systems. We also discuss security caveats. Although SURF does not fully handle some attacks, we provide mitigation ideas which can effectively close these security gaps.

4.1.1 Mitigating Common Attacks

The following list enumerates commonly observed attacks against software update systems.

1. Arbitrary installation attacks
2. Extraneous dependencies attacks
3. Fast-forward attacks
4. Mix-and-match attacks
5. Replay attacks
6. Denial of service attacks

Assuming adversaries cannot compromise the underlying blockchain (*e.g.*, rewriting multiple blocks), SURF effectively mitigates the following attacks.

1. Arbitrary installation attacks

This is probably the most desired attack by an attacker to compromise an update server. A successful arbitrary installation attack allows an attacker to lead a client to a fake update and eventually install anything the adversary wants on the client system.

Attackers, who have successfully compromised a SURF server, may attempt to fool SURF clients in various ways:

- **Providing malware binary**: attackers, who also have compromised software binary repositories, may attempt to provide malware to clients. Since the last operation has a secure hash of a valid binary, clients can verify the downloaded binary before installation.
- **Injecting malware hashes**: attackers can provide an operation containing a malware hash (*i.e.*, the last 32-byte of `data` is malware hash). A SURF client naturally mitigates this attack as such a modified `data` would result in verification failure when the client runs an identity script. Hence, unless attackers have also compromised developer signing keys, registrations and revisions cannot be altered, and clients can safely verify the authenticity of operation `data`.
- **Forged transactions**: attackers may forge transactions and create a completely different update chain for a specific software package registration. However, all update chains must be chained to a registration by spending the valid transaction output. Since SURF clients can verify transaction outputs without consulting SURF server, adversaries cannot deceive clients.

2. Extraneous dependencies attacks

Using this attack, an adversary can cause clients to download or install software dependencies that are not the intended dependencies.

This attack is similar to the arbitrary installation attack in that its main goal is to install software against client intention; mitigating extraneous dependencies attacks, hence, is possible for the same reason SURF clients can defeat *malware hash injection*. A successful extraneous dependencies attack assumes that adversaries are capable of misleading clients to download a modified set of software libraries containing malware. However, deceiving SURF clients requires modifying `data` field. As explained in the previous attack mitigation, adversaries cannot alter `data` without getting caught unless they compromise

a sufficient number of developer signing keys. Assuming developers devise a reasonably secure identity script (*e.g.*, requesting multiple keys with two-factor authentication), it is difficult for adversaries to launch extraneous dependencies attacks. Therefore, SURF effectively mitigates this type of attacks.

3. Fast-forward attacks

Fast-forward attacks allow attackers to arbitrarily increase the version numbers of metadata of a software update well beyond the current value and thus tricking a software update system into thinking any subsequent updates are trying to rollback the package to a previous, out-of-date version.

Recall the version number is a part of `data`. Thus, changing the version number will create discrepancies between provided signatures and the expected outcome of signature verification, and cause the previously defined identity script to fail, notifying SURF clients that the server is compromised.

4. Mix-and-match attacks

The mix-and-match attack is a variant of extraneous dependencies attack; it allows attackers to trick clients into using a combination of metadata that never existed together on the repository at the same time.

In SURF, mix-and-match attacks cannot occur by design. There are two types of metadata managed by SURF: registration and revision. Associating metadata such as adding revisions to a registration can only happen by spending transaction outputs. Since spent transaction outputs cannot be unspent nor re-spent thanks to double spending prevention, it is impossible to combine metadata in a different way than the form already stored in SURF . Furthermore, all revisions in SURF are ordered such that every subsequent revision satisfies the identity script of the previous revision or registration with developer signatures. Hence, associating forged metadata with existing metadata cannot happen without compromising enough number of developer signing keys.

4.1.2 Caveats

There are some security caveats that are not relevant or fully resolved by SURF .

Replay Attacks

A replay attack is an attack issued by providing a stale, previously valid update with known vulnerabilities. Known variants are *freeze* and *rollback attacks*. The purpose of this type of attacks is to stall updates, opening up a time window for adversaries to exploit unpatched vulnerabilities. An attacker may collude with a malicious repository mirror in addition to, or instead of, compromising centralized update servers. For example, a malicious mirror can launch a replay attack by giving a previously signed update with known bugs to clients requesting the most recent updates.

SURF only partially solves this problem. The primary purpose of SURF is to separate update metadata and store them in an unforgeable data structure, allowing clients to query and verify the update records of a software package. The assumption is if clients can verify an operation log then the last revision in the log is a valid update; there is no guarantee that the last revision always indicates the latest update. Thus, in case the SURF server is compromised, an adversary can launch a replay attack on clients by responding with partial operation logs.

However, the attack is limited as (i) adversaries cannot equivocate clients (*i.e.*, all clients should see the same operation logs), and (ii) clients can check whether the last revision has a valid UTXO which has not been spent in cooperation with reputable full nodes.

A simple workaround for a developer to detect replay attacks is to run a client. Within a reasonable period (e.g., 72 hours for Bitcoin; average delay for a transaction to get confirmed), if the developer who added the last revision to the SURF server does not get the complete operation log, it can suspect that the server is compromised and actively deploying a replay attack.

Another workaround is to leverage known reputable full nodes as a trusted third party. For UTXO-based SURF implementations, clients can cooperate with well-known full nodes to check whether the UTXO of the last revision is spent or not. Since our threat model assumes even a powerful adversary cannot rewrite the blocks, a spent UTXO cannot be reverted. Since SURF server only spends UTXOs when registering software packages and appending revisions, if the last revision has a spent UTXO, it implies that the server is deliberately hiding part of the operation log.

As of May 2018, no cryptocurrencies offer the possibility for clients to verify UTXOs; one must run a full node and synchronize blockchain data to check whether a UTXO has been spent. Hence, clients have no options but to trust some well-known, trusted full nodes (e.g., blockchain.info for Bitcoin) to query whether the UTXO of the last revision, supposedly reserved for the next revision, is spent to determine the completeness of an operation log. However, this method significantly degrades SURF's security as it relies on trusting third parties. In this case, it is highly recommended to have multiple full nodes and cross-validate results to ensure full nodes are honest.

Denial of Service

Another possible attack that an adversary can carry out is the denial of service (DoS) attack. A compromised SURF server may ignore or deliberately slow down client requests to deter software updates. Attacks known as *endless data attacks*, which respond to clients with huge amounts of data, and *slow retrieval attacks*, which intentionally delay responses until clients give up requesting updates, are variants of DoS.

Although SURF cannot fully prevent denial of service attacks as they are indistinguishable from benign service outages, a successful DoS attack does not affect client integrity as it does not exploit client vulnerabilities or propagate malware to subvert clients. Thus, the impact on clients from this type of attack is minimal.

4.1.3 Inherent Limitations

The SURF threat model assumes that the underlying blockchain is resilient to compromises. It is, however, possible to compromise a blockchain with sufficiently large computation power or launch a targeted attack on some clients by partitioning the network momentarily.

51% attack

Blockchains employ a consensus algorithm to allow participants to come to an agreement for each block. Bitcoin's consensus algorithm assumes that if more than half of the consensus participants, or miners, are honest, then the network is safe and free from Sybil attacks and collusions. This idea is based on Bitcoin's incentive mechanism which only rewards those blocks appended to the longest chain; if the sum of computational power provided by honest miners is greater than the computational power collectively owned by malicious miners, then the rate of adding good blocks will always be higher than the rate of adding bad blocks since creating a block requires computationally expensive mathematical calculation. Nonetheless, the opposite is also true — if malicious miners own more computational power than the good ones, the network is vulnerable to the so-called 51% attack [58].

The 51% attack, also known as *the majority attack*, can take place when the attacker owns more than half of network hash power. While it is difficult to achieve such condition, a successful 51% attack gives adversaries an ability to perform double spending by replacing last n blocks with malicious $n+1$ blocks [13]. Although this is a severe concern for Bitcoin or any other blockchain, due to identity retention, an adversary with the majority hash power can only undo transactions (*i.e.*, reverting confirmed revisions) to hide some software updates or registrations from clients in the case of SURF.

Nonetheless, as already discussed in the previous subsection, hiding operations is only useful for delaying software updates, buying some time for adversaries to exploit known vulnerabilities — not injecting malware to clients. Of course, unlike replay attacks, the 51% attack can alter blockchain data with adversary chosen blocks, effectively ruling out one of the workarounds suggested before. Yet, developers can still check whether their updates are published by obtaining operation logs and quickly realize the server has been compromised.

In the context of the software update, performing the 51% attack is not economically sound since the resources required for the attack are enormous, and the consequences of the attack are limited.

Eclipse attack

The eclipse attack, on the other hand, is a far more effective, economically-sound attack which consists in partitioning the blockchain network. In an eclipse attack, an adversary monopolizes all incoming and outgoing network connections of a blockchain full node — the victim, isolating, or eclipsing the victim from the network. Then, the adversary manipulates the eclipsed victim's view of the blockchain by providing obsolete transactions [37].

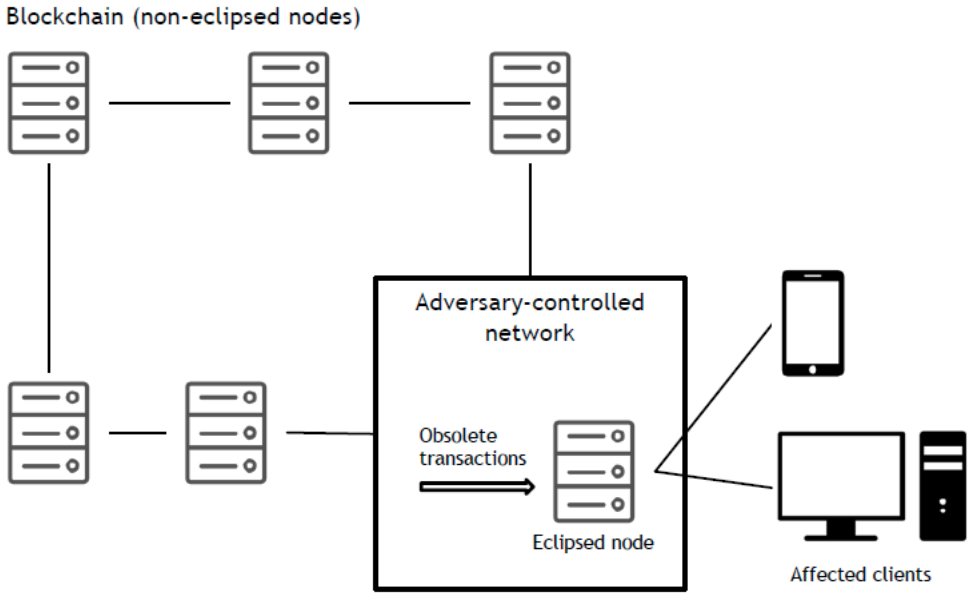

Figure 4.1: Eclipse attack

This can be a great threat to SURF. Adversaries can manipulate the eclipsed miners to generate blocks containing adversary-chosen transactions. If adversaries know which blockchain nodes (miners) a client connects to, they can eclipse those nodes and intentionally delay software updates of the client by removing relevant transactions. Figure 4.1 describes the case of a blockchain node being eclipsed into an adversary-controlled network. Consequently, a successful eclipse attack achieves the same goal as the 51% attack on a specific target at much lower cost.

4.2 Performance

Although SURF is designed for secure software update systems. For its wide adoption, it is imperative to provide good performance without requiring much resources. Indeed, SURF performs fast verification with relatively low resource consumption. To illustrate this, we evaluated SURF through two experiments: (1) verification latency and (2) efficiency.

Section 4.2.1 demonstrates that SURF performs comparably to centralized systems. We compare SURF to TUF, the secure start-of-the-art centralized software update system. We simulate software updates on local instances of SURF and TUF and compare secure lookup latency, which entails both lookup and verification of updates, of both systems. In this experiment, we used a 3.7GHz AMD Ryzen 7[1] processor and 32GB of RAM.

Section 4.2.2 shows the efficiency in verifying software updates by running SURF clients on resource constrained devices such as Raspberry Pi. We run SURF client on a Raspberry Pi 2 Model B[2] and verify simulated packages with varying lengths of operation logs. As a result, SURF successfully verifies all packages within reasonable runtime.

4.2.1 Verification latency

Both SURF and TUF lookup updates and verify them by checking the accompanying signatures. The objective of this evaluation is to compare the latency difference between SURF and TUF verification in a similar condition.

To achieve the experiment objective, first, we baseline the experiment by evaluating TUF. We simulate software updates on an arbitrary package including a single file by repeatedly changing the file and registering it to the TUF repository. For each new change, we run and time the TUF client which verifies the new update. Although the client fetches the updated file while performing verification, we keep the file size small (*e.g.*, less than 100 bytes) so the file transfer is negligible. We run one hundred updates and verifications to obtain a sample set that can best describe the latency of a single verification.

We use PySURF, our lightweight SURF client, to illustrate the performance of SURF in this experiment. We run two sets of tests; one with caching and one with no caching. Recall that in the most secure setup SURF client fetches the entire transaction chain of a software package and relevant blockchain headers to verify the updates. While this is necessary for the first time verifying a software package, we can always cache the last

[1] AMD Ryzen 7 2700X; 8 core, 16 threads

[2] 900MHz quad-core ARM Cortex-A7 CPU; 1GB RAM; 100 Base Ethernet

update record and use it to verify a new update without going over all past updates. Similar to the TUF evaluation, we simulate one hundred updates on an arbitrary software package. All updates use the same (2,3)-quorum identity script with three keys and come with proper signatures (*i.e.*, two or three signatures satisfying the identity script). In the caching scenario, we use the last known update record to verify the new update; and in the non-caching scenario, we fetch all update records and verify them all each time when there is a new update.

Figure 4.2 shows the latency comparison between PySURF and TUF. It shows that, with caching (the line with dots), PySURF performs relatively better than TUF (the line with empty circles). As expected, without caching (the line with squares), the latency grows linearly; however, this is only the worst case scenario where clients rarely update their software. In practice, it is likely that clients often check software updates with a frequency depending on the use case.

4.2.2 Efficiency

Since update verification involves cryptographic operations requiring high CPU computation, it is important to evaluate how SURF can indeed run on resource-constrained computers such as IoT devices. We ran PySURF on Raspberry Pi 2 Type B (RPi2B), one of the popular test boards for IoT development in many industries, to show the latency difference compared to that of the high-end desktop (DESKTOP) used in the previous experiment. In this experiment, we verify multiple software packages where each has a single update record. The number of signatures to verify per update record ranges between two to three, similar to the setup of the previous experiment.

Recall that the verification latency is directly proportional to the length of operation log. That is, the overall time consumed for a verification increases if an operation log contains many transactions that need to be verified. Although signature verification is the most significant factor when it comes to latency, there is proof of existence (PoE), an inevitable cost that the client has to pay when it bootstraps a software package for the first time. As SURF uses the index tree, the number of PoE transactions in an operation log can be at most $logN$ where N is the total number of software packages registered to SURF.

To show the possible latency gap between operation logs with a varying number of PoE transactions, we ran PySURF for twenty software packages with a different number of PoE transactions ranging from 2 to 9. The following plot illustrates verification latency observed from RPi2B and DESKTOP.

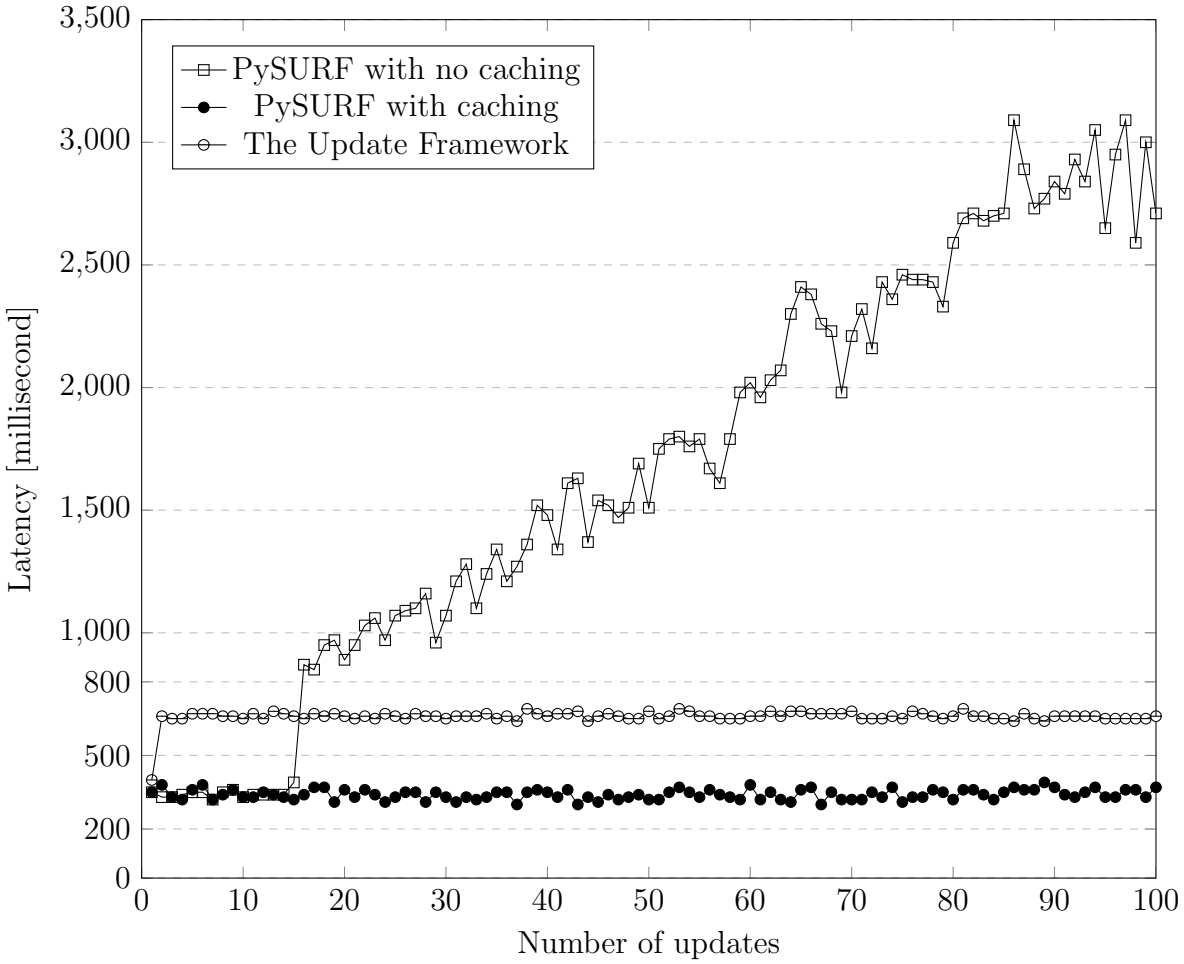

Figure 4.2: Verification latency for software package with varying number of updates

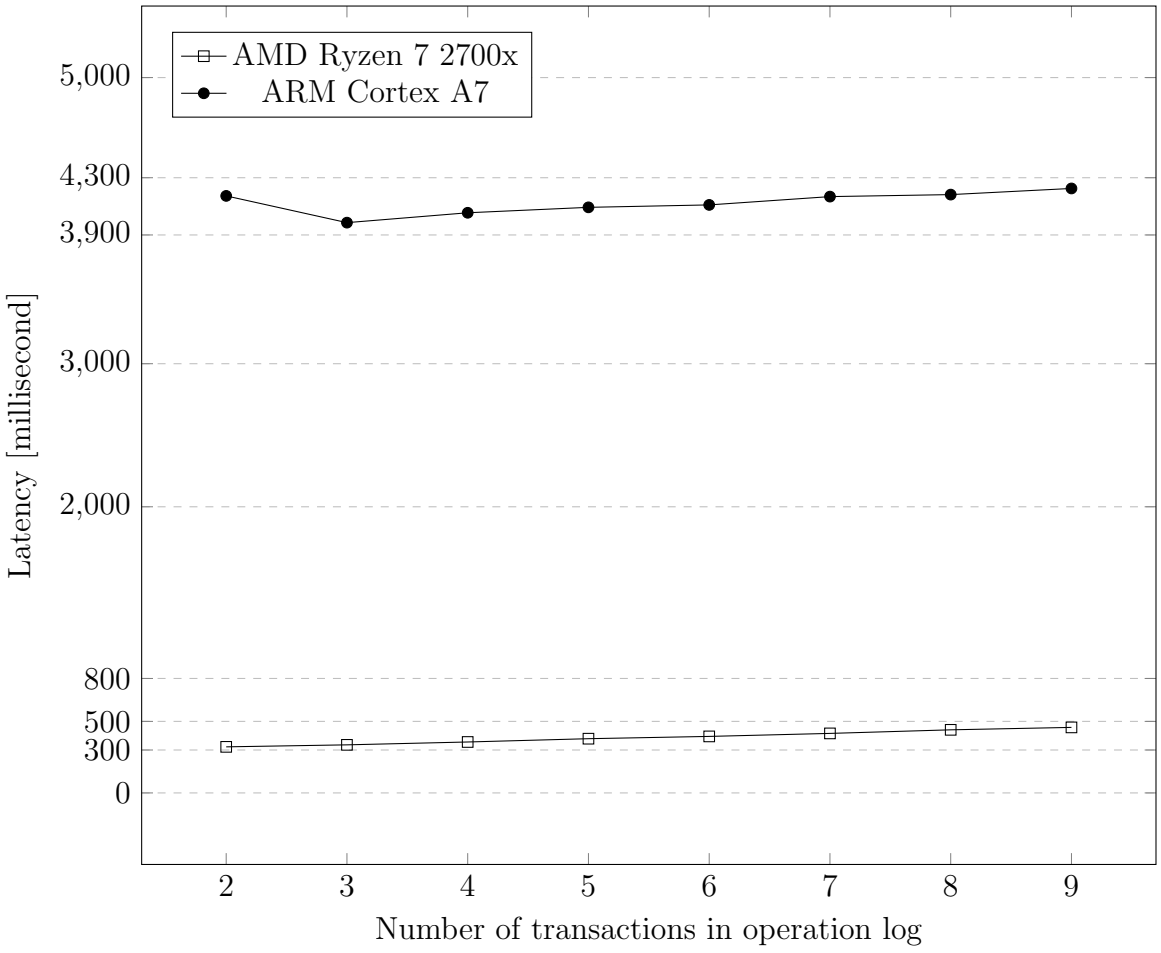

Figure 4.3: Verification overhead on different machines

Figure 4.3 shows the latency gap between DESKTOP and RPi2B. Since we only verify a single update record and PySURF utilizes a single core, the only meaningful hardware difference between RPi2B and DESKTOP is the CPU clock. Based on the benchmarks obtained from Geekbench [4], the CPU performance gap between AMD Ryzen 7 2700x and ARM Cortex A7 is about 15x (4,670 vs. 305). Assuming differences in the speed of memory and the network speed are negligible, PySURF performed well with limited-capacity hardware given that the latency gap is around 12x - 13x.

Notice that the number of PoE transactions (i.e., the number of transactions in operation log less one) marginally affects the verification latency, confirming that signature verification is the deciding factor in verification latency. We conclude that verifications can take place on limited-capacity devices comparable to RPi2B within few seconds.

4.3 Operation Cost

The cost of running a system is a determining factor for a sustainable operation in the long term. Even a good system can be considered impractical if it incurs too much operational cost. Most of currently available blockchains require clients to pay fees when they broadcast transactions mainly due to prevent abuses and network spamming. Since writing to a blockchain costs transaction fees, running a SURF server, which broadcasts transactions for every registration and revision, comes at a price.

Clients broadcasting transactions need to pay fees in the platform-specific cryptocurrencies (*e.g.*, BTC, ETH). Each blockchain employs different pricing mechanisms; the total cost for a transaction converted in fiat money such as USD varies based on the selected blockchain and the exchange rate (*i.e.*, cryptocurrency price). For example, the Bitcoin transaction fee depends on the number of bytes of a transaction whereas Ethereum fee is determined by the number of instructions a smart contract executes triggered by the client transaction. Thus, SURF operation cost may vary as different implementations of SURF encode transactions or write smart contracts in different ways.

Tofino with Bitcoin, for instance, creates a transaction which consumes a UTXO and outputs three additional transaction outputs (one for the left subtree, one for the right subtree, and one for the update chain) when registering a new software package. Since each transaction output must be greater than the *dust* amount, Tofino spends 0.00005 BTC per transaction output. Hence, the total money spent on a registration transaction is 0.0001 BTC. Additionally, with two public keys, the average size of a registration transaction is 700 bytes. Given the minimum transaction fee allowing a transaction gets confirmed within 80 minutes is 0.0000001 BTC/byte (10 satoshis/byte) [2], the minimum fee for a registration transaction is 0.00007 BTC. In total, registering a software package costs 0.00017 BTC, which is roughly 1.28 USD[3].

Blockchain	**Registration**		**Revision**	
	Crypto	USD	Crypto	USD
Bitcoin	0.00017 BTC	1.28	0.00006 BTC	0.45
Ethereum	0.00132 ETH	0.77	0.00071 ETH	0.40

Table 4.1: Operation Cost Comparison

Table 4.1 compares average operation costs of registering a new software package and

[3]Based on the exchange rate 7534.79 USD/BTC, coinbase.com on June 1, 2018

adding a new revision for Tofino integrated with Bitcoin and Ethereum. The exchange rates of cryptocurrencies used in the table are 7534.79 USD/BTC and 580.01 USD/ETH[4] where platform-specific transaction fees are 10 satoshis/byte (Bitcoin) and 10 Gwei/gas (Ethereum).

[4]Retrieved from coinbase.com on June 1, 2018

Chapter 5

Conclusion

The unique strength of SURF lies in that it inherits the security of the underlying blockchain without compromising performance. Indeed, to the best of our knowledge, SURF is the first software update system that fully leverages the protection offered by large public blockchains such as Bitcoin and Ethereum.

It is also worth noting that the use of the multi-signature scheme for individual records makes SURF resilient to key compromises. Although the current quorum-based implementation performs adequately, it is possible to improve the multi-signature support by employing Schnorr signatures [53] or CoSi [55] as demonstrated in CHAINIAC, reducing the overall size of the operation log and further cutting down the verification latency.

Thin client support has been the missing piece in blockchain-based security systems, and SURF successfully fills this gap by leveraging Bitcoin SPV and providing a light-weight Python implementation, PySURF. Although PySURF well serves its purpose of supporting a broad spectrum of client devices, in the future, it would be beneficial to develop a compiled program (*e.g.*, C, Go) to help even smaller devices such as microcontrollers powering small IoT devices.

In conclusion, SURF contributes to providing an efficient, secure, decentralized software update system that can leverage virtually any blockchain offering high security. Conducted a set of experiments show SURF has comparable performance to centralized update systems in terms of verification latency and demonstrate SURF efficiency in verifying updates on low-profile devices within reasonable runtime.

References

[1] Apt - debian wiki. https://wiki.debian.org/Apt.

[2] Bitcoin minimum fee. https://bitcoinfees.earn.com.

[3] Blockchain.info. https://blockchain.info.

[4] Cpu benchmark for rpi2b and ryzen 2700x. https://browser.geekbench.com/v4/cpu/search.

[5] Eos whitepaper. https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md.

[6] Ethereum smart contracts. http://www.ethdocs.org/en/latest/contracts-and-transactions/contracts.html.

[7] Ethereum yellow paper. https://ethereum.github.io/yellowpaper/paper.pdf.

[8] The first collision for full sha-1. https://shattered.io/static/shattered.pdf.

[9] Global bitcoin nodes distribution. https://bitnodes.earn.com/.

[10] Google ipv6 statistics. https://www.google.com/intl/en/ipv6/statistics.html.

[11] Heartbleed bug. http://heartbleed.com.

[12] Hyperledger fabric chaincode. http://hyperledger-fabric.readthedocs.io/en/release-1.1/chaincode4ade.html.

[13] Majority attack. https://en.bitcoin.it/wiki/Majority_attack.

[14] Namecoin. http://namecoin.info.

[15] pip - pypi. https://pypi.org/project/pip/.

[16] Update on petya malware attacks. https://blogs.technet.microsoft.com/msrc/2017/06/28/update-on-petya-malware-attacks/.

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

[18] Adam Back et al. Hashcash-a denial of service counter-measure, 2002.

[19] Georg Becker. Merkle signature schemes, merkle trees and their cryptanalysis. *Ruhr-University Bochum, Tech. Rep*, 2008.

[20] Daniel J Bernstein, Daira Hopwood, Andreas Hülsing, Tanja Lange, Ruben Niederhagen, Louiza Papachristodoulou, Michael Schneider, Peter Schwabe, and Zooko Wilcox-OHearn. Sphincs: practical stateless hash-based signatures. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 368–397. Springer, 2015.

[21] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing tls with verified cryptographic security. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 445–459. IEEE, 2013.

[22] Hanno Böck, Juraj Somorovsky, and Craig Young. Return of bleichenbachers oracle threat (robot). 2017.

[23] Sergio Boixo, Sergei V Isakov, Vadim N Smelyanskiy, Ryan Babbush, Nan Ding, Zhang Jiang, Michael J Bremner, John M Martinis, and Hartmut Neven. Characterizing quantum supremacy in near-term devices. *Nature Physics*, page 1, 2018.

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

[25] Johannes Buchmann, Erik Dahmen, and Michael Szydlo. Hash-based digital signature schemes. In *Post-Quantum Cryptography*, pages 35–93. Springer, 2009.

[26] Arjun Chopra. Glyph: A new instantiation of the glp digital signature scheme. Technical report, Cryptology ePrint Archive, Report 2017/766, 2017.

[27] Vulnerability Notes Database. Apache web servers fail to handle chunks with a negative size, 2002.

[28] Vulnerability Notes Database. Openssl servers contain a buffer overflow during the ssl2 handshake, 2002.

[29] Antoine Delignat-Lavaud, Martín Abadi, Andrew Birrell, Ilya Mironov, Ted Wobber, and Yinglian Xie. Web pki: Closing the gap between guidelines and practices. In *NDSS*, 2014.

[30] Jintai Ding and Bo-Yin Yang. Multivariate public key cryptography. In *Post-quantum cryptography*, pages 193–241. Springer, 2009.

[31] Yuhao Dong, Woojung Kim, and Raouf Boutaba. Bitforest: a portable and efcient blockchain-based naming system. In *Conference on Network and Service Management (CNSM)*. IEEE, 2018.

[32] Yuhao Dong, Woojung Kim, and Raouf Boutaba. Conifer: centrally-managed pki with blockchain-rooted trust. In *IEEE International Conference on Blockchain*. IEEE, 2018.

[33] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In *Advances in Cryptology–CRYPTO 2013*, pages 40–56. Springer, 2013.

[34] Conner Fromknecht, Dragos Velicanu, and Sophia Yakoubov. A decentralized public key infrastructure with identity retention. *Massachusetts Inst. Technol., Cambridge, MA, USA, Tech. Rep*, 6, 2014.

[35] Conner Fromknecht, Dragos Velicanu, and Sophia Yakoubov. A decentralized public key infrastructure with identity retention. *IACR Cryptology ePrint Archive*, 2014:803, 2014.

[36] Red Hat. Critical: openssh security update, 2008.

[37] Ethan Heilman, Alison Kendler, Aviv Zohar, and Sharon Goldberg. Eclipse attacks on bitcoin's peer-to-peer network. In *USENIX Security Symposium*, pages 129–144, 2015.

[38] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *International Workshop on Post-Quantum Cryptography*, pages 19–34. Springer, 2011.

[39] Dan Kaminsky. Spelunking the triangle: Exploring aaron swartz's take on zooko's triangle. http://dankaminsky.com/2011/01/13/spelunk-tri/, January 2011.

[40] Arjen K Lenstra. Integer factoring. In *Towards a quarter-century of public key cryptography*, pages 31–58. Springer, 2000.

[41] Enrique Martin-Lopez, Anthony Laing, Thomas Lawson, Roberto Alvarez, Xiao-Qi Zhou, and Jeremy L O'brien. Experimental realization of shor's quantum factoring algorithm using qubit recycling. *Nature Photonics*, 6(11):773, 2012.

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

[43] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 120–130. IEEE, 1999.

[44] Daniele Micciancio and Oded Regev. Lattice-based cryptography. In *Post-quantum cryptography*, pages 147–191. Springer, 2009.

[45] Greg Miller. Revving software with update engine, 2008.

[46] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.

[47] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. The return of coppersmith's attack: Practical factorization of widely used rsa moduli. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1631–1648. ACM, 2017.

[48] André Niemann and Jacqueline Brendel. A survey on ca compromises.

[49] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. Chainiac: Proactive software-update transparency via collectively signed skipchains and verified builds. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1271–1287, 2017.

[50] Raphael Overbeck and Nicolas Sendrier. Code-based cryptography. In *Post-quantum cryptography*, pages 95–145. Springer, 2009.

[51] Albrecht Petzoldt, Stanislav Bulygin, and Johannes A Buchmann. Selecting parameters for the rainbow signature scheme-extended version-. *IACR Cryptology ePrint Archive*, 2010:437, 2010.

[52] Justin Samuel, Nick Mathewson, Justin Cappos, and Roger Dingledine. Survivable key compromise in software update systems. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 61–72. ACM, 2010.

[53] Claus-Peter Schnorr. Efficient signature generation by smart cards. *Journal of cryptology*, 4(3):161–174, 1991.

[54] Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen K Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Md5 considered harmful today, creating a rogue ca certificate. In *25th Annual Chaos Communication Congress*, number EPFL-CONF-164547, 2008.

[55] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. Keeping authorities" honest or bust" with decentralized witness cosigning. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 526–545. Ieee, 2016.

[56] Alin Tomescu and Srinivas Devadas. Catena: Efficient non-equivocation via bitcoin. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 393–409. IEEE, 2017.

[57] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1313–1328. ACM, 2017.

[58] Jesse Yli-Huumo, Deokyoon Ko, Sujin Choi, Sooyong Park, and Kari Smolander. Where is current research on blockchain technology?a systematic review. *PloS one*, 11(10):e0163477, 2016.

APPENDICES

Appendix A

Identity Script and Examples

A.1 Types of Identity Scripts

Identity script is a simple stack-based script with two types of scripts: *quorum* and *key*.

A.1.1 Quorum Script

A quorum script is an identity script where:

- n is the number of required votes
- m is the number of total votes
- *scripts* is one or more identity scripts yielding 1 or 0 votes

A.1.2 Key Script

A key script is an identity script with two fields:

- *type* indicates a type of cryptographic algorithm used for this key
- *pub_key* is a public key that can verify a signature

A.2 Examples

```
"id_script": {
   "n": 2,
   "m": 2,
   "scripts": [
      { "type": "ED25519", "pub_key": "alice_pubkey" },
      {
         "n": 1,
         "m": 3,
         "scripts": [
            { "type": "ED25519",   "pub_key":  "alice_cellphone" },
            { "type": "SECP256k1", "pub_key":  "alice_tablet"    },
            { "type": "SECP256r1", "pub_key":  "alice_desktop"   }
         ]
      }
   ]
}
```

Listing 8: Example identity script illustrating Two factor authentication (2FA)

The example identity script in 8 illustrates a simple setup of a two factor authentication which enforces Alice to provide her password and additional authentication from one of her registered devices (*i.e.*, cellphone, tablet, desktop).

Appendix B

SURF Ethereum Smart Contract in Solidity

```
pragma solidity ^0.4.0;
contract Tree {
    uint public genesis = 0;
    address public owner = msg.sender;
    uint public creationTime = now;
    uint public next;

    struct Revision {
        uint revision;
        bytes32 opHash;
    }

    struct Node {
        bytes32 name;
        uint left;
        uint right;
        uint numRevisions;
        mapping(uint => Revision) revisions;
    }

    mapping(uint => Node) data;

    function Tree(bytes32 name, uint revision, bytes32 opHash) public {
        Node storage gen = data[genesis];
        gen.name = name;
        gen.revisions[0].revision = revision;
        gen.revisions[0].opHash = opHash;
        gen.numRevisions = 1;
        next = 1;
    }

    // functions omitted for brevity
}
```

Listing 9: Solidity script for SURF + Ethereum

```
function getPosition(bytes32 name)
        public
        returns (uint)
{
        return getPosition(genesis, name);
}

function getRevisions(bytes32 name)
    public
    returns (bytes32[])
{
    uint pos = getPosition(name);
    Node storage node = data[pos];
    bytes32[] memory ret = new bytes32[](node.numRevisions);
    for (uint i = 0; i < node.numRevisions; i++) {
            ret[i] = node.revisions[i].opHash;
    }
    return ret;
}

function getPosition(uint pos, bytes32 name)
        private
        returns (uint)
{
        Node storage curr = data[pos];
        if (curr.name < name) {
                require(curr.right > 0);
                return getPosition(curr.right, name);
        }
        else if (curr.name > name) {
                require(curr.left > 0);
                return getPosition(curr.left, name);
        }
        else return pos;
}
```

Listing 10: Solidity script 2 for SURF + Ethereum

```
function append(bytes32 name, uint revision, bytes32 opHash)
        public
        onlyBy(owner)
{
        append(genesis, name, revision, opHash);
}

function append(uint pos, bytes32 name, uint revision, bytes32 opHash)
        private
        onlyBy(owner)
{
        Node storage curr = data[pos];
        if (curr.name < name) {
                if (curr.right == 0) {
                        curr.right = appendNewNode(name, revision, opHash);
                } else {
                        append(curr.right, name, revision, opHash);
                }
        } else if (curr.name > name) {
                if (curr.left == 0) {
                        curr.left = appendNewNode(name, revision, opHash);
                } else {
                        append(curr.left, name, revision, opHash);
                }
        } else {
                uint idx = curr.numRevisions;
                require(curr.revisions[idx-1].revision < revision);
                curr.revisions[idx].revision = revision;
                curr.revisions[idx].opHash = opHash;
                curr.numRevisions = idx + 1;
        }
}

function appendNewNode(bytes32 name, uint revision, bytes32 opHash)
        private
        onlyBy(owner)
        returns (uint)
{
        uint pos = next;
        next = next + 1;
        Node storage n = data[pos];
        n.name = name;
        uint idx = n.numRevisions;
        n.revisions[idx].revision = revision;
        n.revisions[idx].opHash = opHash;
        n.numRevisions = idx + 1;
        return pos;
}
```


Listing 11: Solidity script 3 for SURF + Ethereum

```
modifier onlyBy(address _account) {
        require(msg.sender == _account);
         _; // replaced by the actual function body when the modifier is used
}
```

Listing 12: Solidity script 4 for SURF + Ethereum