Refactoring Interfaces: Toward a Widely Distributed Secure Data Plane for Unified Communication and Storage

Abstract

To be fixed, but in the end... We present the idea and design of Secure Temporal Streams (STS), a unified tool for secure communication and storage aimed at the Internet of Things (IoT). Compared to SSL/TLS, which is used for interactive communication between a client and a server, STS is a merkle-tree based uni-directional communication stream that can be stored on untrusted infrastructure and queried at a later time. Using STS provides a narrow waist of communication to physical devices, thus reducing the attack surface. It is a tool that device manufacturers or application developers can use to incorporate good security practices, while maintaining easy composability of applications/devices to create a better internet of connected things.

1 Introduction

It is a widely accepted fact that the state of security in the Internet of Things (IoT) is very bad [6, 2, 10, 4]. With varied kind of smart devices present in every aspect of life, the impact of a security breach could range from a minor annoyance to failure of critical infrastructure. Not only security, such smart devices lead to endless new privacy issues that didn't exist before [5]. Securing IoT is as important as securing any other computer system, if not more so.

However, the balance of recent research work has been more towards offensive security rather than defensive techniques in the IoT landscape. Offensive security research has its place in discovering new attack techniques, bringing attention to the challenges and raising awareness in general. Providing cleaner interfaces to hardware and application designers that prevent the most obvious security challenges is the big elephant in the room that nobody wants to talk about. We propose to change this using Secure Temporal Streams (STS), a tool to incorporate good data and communication security practices for a wide variety of applications.

Before attempting to propose solutions for securing the IoT, we need to understand the challenges first. Except for the pervasive nature of devices, is there any fundamental difference between IoT security and traditional IT security? The Open Web Application Security Project (OWASP) has compiled a list[1] of the top 10 IoT vulnerabilities [1]. None of these security challenges are specific to IoT landscape. So what makes securing the IoT so hard? We attribute the security challenges to two broad reasons:

1. Heterogeneous Systems: Designing an absolutely secure system is challenging, time-consuming and costly. Any system involving more than a few hundred lines of code is prone to software bugs and security issues. The heterogeneity of devices/software in the IoT landscape

[1]OWASP Top 10 IoT vulnerabilities: (1) Insecure web interfaces, (2) Insufficient authentication/authorization, (3) Insecure network services, (4) Lack of transport encryption, (5) Privacy concerns, (6) Insecure cloud interface, (7) Insecure mobile interface, (8) Insufficient security configurability, (9) Insecure software/firmware, (10) Poor physical security.

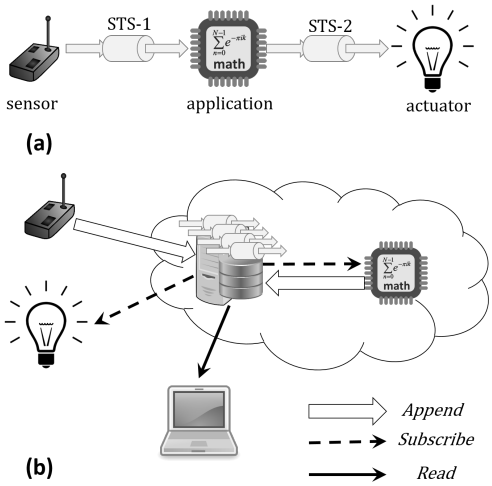

Figure 1: (a) A logical view of STS: it provides an analogy of a pipe with an attached bucket. A sensor appends data to 'STS-1' which is consumed by an application, which processes the sensor data and writes it to an actuation STS. An application developer does not need to interact with physical devices; instead the STS provides a virtual device with associated history. (b) The actual physical view of the STS, where an STS-server physically stores data and multiple readers can be subscribed to the same STS. **fix the diagram: a bit too cartoony, at least for (b)**

leads to a large number of unique designs and code-bases; statistically a significant fraction of these systems will have security issues. In traditional computing, de-facto standard tools and software libraries have helped focus the efforts to create better and well maintained software that get regular security updates. Such practices are hard to realize in the heterogeneous IoT world, especially when there is little financial motivation for a device vendor to provide long-term security patches. Lack of software re-usability and market pressure to quickly release products leads to the necessary security features often being considered 'optional'. In addition, any resource-intensive security shims around potentially vulnerable software that work reasonably well for traditional computer systems (e.g. sand-boxing, firewalls, intrusion-detectors, anti-malware programs) are impractical for most of the IoT devices.

2. Management Overhead and Usability: Another security challenge for the IoT is usability. Better security usually is at odds with usability, especially when it comes to the management overhead of authentication and authorization (e.g. password management). In the case of IoT (or any machine-to-machine communication), connectivity and composability of devices and services is a significant driving factor. Good security practices are usually neglected in favor of such usability goals, especially in order to make stove-piped solutions talk to each other. In addition, securing a wide variety of devices requires one to be an expert of all possible systems, which, combined with reduced usability, leads to human errors. [13]

Certainly these are not the only challenges for IoT security, but these definitely are some of the distinguishing factors that separate IoT security from anything else. Guided by these two broad challenges, we present the idea of a Secure Temporal Stream (STS), a middle-ware that attempts to fix the IoT security scenario by providing a standardized narrow waist with reduced attack surface for communicating with physical devices.

Secure Temporal Stream is an authenticated data structure representing uni-directional streams of data with built-in data integrity and confidentiality using cryptographic tools, and lives on potentially distributed untrusted infrastructure. Such uni-directional communications can be stored

on-disk and queried at a later time, thus providing a time-shift property. STS supports publish/subscribe mode of operation enabling IoT style communication, as well as efficient random reads for older data, making it a unified tool for both storage and communication.

The reason STS fits IoT landscape is because many of the data-streams in IoT are either inherently uni-directional (sensor → application → actuator), or can be split into multiple uni-directional communication streams correlated with timestamps. STS can be used to represent a stream of data generated by a sensor, or consumed by an actuator, or input/output of an application processing data, thus virtualizing the physical devices in some sense (see Fig. 1). Using STS, any access-control, firewall, intrusion-detection, etc can be applied to the stream of data living outside the device, thus reducing replication of software functionality on individual devices. In addition, such standardized streams of data incorporate good security practices and enable easy composability, multicast, and hopefully end-to-end security in the IoT. Not only this, STS can be tuned to match the needs and resource availability of heterogeneous components in IoT.

fix this Our vision is to use STS to do to data-storage what SSL/TLS has done to communication. SSL/TLS has enabled us to get data around without trusting any of the network components in between. Internet access is now a commodity, and even though a malicious adversary may observe traffic between two end-points, some level of security/integrity is still maintained. We hope that with STS in a distributed environment, individual users and enterprises can put their own storage hardware that plays along well with a storage provider's service in the same way networking hardware does. We hope that the trust requirements are just as minimal for storage as they are in communication. In addition, because of the major use-case of streaming data as evidenced by IoT, the boundaries between data-storage and communication should not be as strict as they are today.

STS is developed in the context of ExaPlane (ExP)[2], which is a peer-to-peer system of storage and routing nodes that provide a physical backing for such STSs. At a high level, ExP is to an STS as a distributed file-system is to a file. The distributed nature of ExP provides an opportunity to use locality wherever possible for better performance, at the same time enabling a seamless integration with the Cloud. In addition, ExP provides various fundamental services to make STS more usable, e.g. directory service mapping a human-readable name to an STS. Although motivated by the needs of IoT, STS and the ExP are not restricted to IoT and can be readily used for traditional computation scenarios. In particular, STS in itself can be easily deployed on existing cloud infrastructure. For the rest of the paper, we will work at the abstraction of a client and a server, irrespective of whether the server is part of a P2P network or hosted in a cloud-datacenter.

Note that this paper is a general philosophical guide with a high level design overview for STS, rather than a detailed specification manual. We skip unnecessary details in order to focus on the high-level design decisions throughout the paper. The paper is organized as follows: we introduce STS interface and describe the application model and a formal threat model in Sec. 2. Next, we discuss the internal implementation of the various operations on an STS and the possible optimizations/enhancements in Sec. 3. We follow-up with the security argument and evaluation in Sec. 4. A brief summary of the related work in this direction is presented in Sec. 5, followed by a summary of our contributions in Sec. 6.

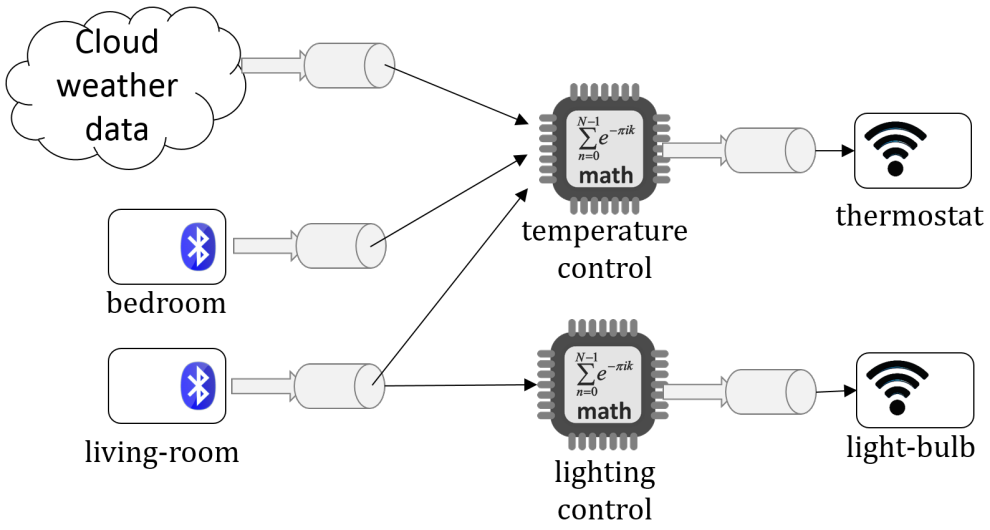

Figure 2: An example home-automation solution, where physical sensors and actuators are represented by STSs to applications. Each application can have multiple input/output STSs, including those representing external data sources. Each sensor STS can be subscribed by multiple applications as well.

2 Secure Temporal Streams

2.1 STS User Interface

In this section, we describe the STS interface for application developers and device manufacturers who want to just use it as a tool, without necessarily knowing the internal details.

An STS is a single-writer, append only authenticated data structure; each update is associated with a signed 'heartbeat' generated by the writer. Read queries can be verified against a particular state of the data-structure, identified by the 'heartbeat'. The single writer and multiple readers of an STS are collectively referred to as clients. Data is physically stored on untrusted STS-servers; STS-servers respond to 'append' queries by the writers and 'read' or 'subscribe' queries by the readers. Encrypted data can be read by anyone (including untrusted STS-servers), but only the authorized readers have the correct decryption key to make sense of data. Clients use digital signatures and encryption as the fundamental tools to enable trust in data than in infrastructure.

At the very basic level, an STS is an ordered list of immutable records; a record is the unit of read or write to the STS, and can be variable sized. The key operations available to a client are: 'append', 'read' and 'subscribe'. As the name suggests, an 'append' operation is adding new records to the STS. 'Read' is for fetching records by addressing them individually, while 'subscribe' allows a client to request future records as they arrive. An STS can be truncated based on policy decisions, where truncation is marking data older than a threshold 'safe-to-delete'. In addition, there are certain meta-operations, such as 'STS-creation' and 'STS-attach-request' that we will discuss shortly. There are various internal operations such as migration, replication, etc. that STS-servers can perform on the STS, however these operations are opaque to a client and we do not discuss them in detail here.

Each STS has a special record at the beginning, called the metadata. Like other records, the metadata is also immutable. The STS is identified by a flat 256-bit identifier called the 'STSname',

[2] A fictional name for a real system; renamed to avoid self-reference

which is the SHA256 hash of the metadata. The metadata is essentially a list of key-value pairs identifying properties of an STS. At the very least, the metadata contains a public signature key belonging to the designated single writer of the STS, which is used to validate data-integrity and perform write access control. We describe the structure and properties of records and metadata in detail in Sec. 3.1.

Why an append-only design? An append-only design is the most natural choice for streams of data. Further, the data can not be changed once it is written, which transforms data-consistency problems into a data-freshness problem, thus simplifying the design. An STS is a narrow, yet sufficient interface to enable other higher level interfaces, such as that of a file. In many way, append is has similar semantics of 'publish' in a publish-subscribe model, which is a popular communication paradigm in IoT.

Why a single-writer design? The single-writer design is also inspired by the IoT landscape, where a single writer STS is a representative of data-stream from a sensor. It is a reasonable assumption that such sensors have private signature/encryption keys embedded in the device. With such cryptographic capabilities, the writer not only can sign the individual data entries, but also fix the ordering of such entries. An alternative design choice could have been a multi-writer STS, however the added complexity of granting/revoking write permissions is difficult to justify. In addition, ordering of data requires some level of trust from untrusted STS-servers. If need be, multi-writer STS can always be created by feeding multiple single-writer STSs to a service that combines them based on some application-level logic.

2.2 Application Model

With this background about STS, we describe how an application writer would design secure applications and how does STS alleviates the IoT security challenges we identified in Sec. 1.

Keeping the single-writer model of STS in mind, sensors are assigned their own STSs at the time of sensor provisioning. Each sensor appends the data it generates to the associated STS, thus making an STS the only interface to the rest of the world for the sensor. Similarly, each actuator is subscribed to an actuation STS from which it gets the actuation commands. The designated single writer of the actuator STS is either a pre-configured application, or a service representing the actuator (See Fig. 1).

An STS interface makes dumb sensors and actuators significantly more functional. Low-power sensors usually only generate data, but can't answer any queries. If data values are written to an STS by such sensors, the STS can be used as a proxy that supports a much richer set of queries, especially for historical data. A subscription to such an STS provides the latest sensor values in almost real time, thus virtualizing the sensor in some sense.

Actuators, on the other hand, usually need to maintain some kind of access control – by physical isolation, some authentication method, or a combination of both. Instead, if an actuator were to subscribe to an actuation STS to read the actuation commands as we described, access control could be implemented at the STS level. This makes actuator design simpler and avoids the pitfalls of ad hoc authentication mechanisms hastily put together by hardware vendors.

Further, there's no need to expose the physical devices with potentially questionable standards of software security to the entire world, while still being able to connect things together. This is especially important because it takes the burden of implementing security off the device vendors' shoulders. All a device manufacturer has to do is publish data to an STS (in case of a sensor), or subscribe to an STS (in case of an actuator). This greatly reduces the attack surface, because STS implements security best practices.

Representing sensors and actuators with STSs separates policy decisions from mechanisms, enabling cleaner application designs. Applications can be built by interconnecting globally addressable STSs, rather than by addressing devices or services via IP addresses. Further, with applications running inside containers (Docker, Uni-kernels, Intel SGX enclaves, and so on), forcing data-flows in and out of the container through STSs enables any filtering at the STS level (for example, access control). Last but not least, the narrow waist provided by globally addressable STSs avoids stove-piped solutions and provides for a heterogeneous hardware infrastructure.

Let us describe a few sample application scenarios, and we will illustrate how STS enables a cleaner solution compared to a traditional design.

2.2.1 Home Automation

> Imagine a combination of generic off-the-shelf environmental sensors and actuators to create a customized home-automation solution. Based on any environment changes (such as motion of door opening), we would like to trigger certain activities: turn on the lights, adjust air-conditioning appropriately, etc. Such applications really focus on the composability and real-time triggering of actuators.

Since the data from a single sensor could be used for multiple purposes, a conventional application flow would probably use a publish-subscribe mechanism (such as MQTT) to make the data from one sensor available to multiple subscribers (applications/actuators). The open question is: what's the security model for data in transit and at rest? One might consider the home network to be a protected domain and disregard all security practices to achieve the highest composability. However, as has been shown in the case of automobiles where adversaries are able to remotely access a CAN bus, such assumptions of a protected domain are not necessarily true.

Using an STS based solution (see Fig. 2) provides the same level of functionality as a typical publish-subscribe mechanism. Sensor STSs can be subscribed to by multiple applications; even applications can be chained together using STSs (similar to Unix pipes). Because of the very nature of STS, an end-to-end integrity verification guarantees are achieved. Thus, an adversary who has infiltrated the home network can not, for example, deactivate a burglar alarm by spoofing data from some sensor. An additional side-benefit is the storage aspect of STS, which enables one to perform long-term analytics without worrying about security of data at rest.

2.2.2 Wearable devices

> Imagine a wearable sensor (e.g FitBit) continuously monitoring physical activity of a user. During normal operation, the device provides aggregated physical activity data to the user, with the capability of zooming in on certain periods of interest. In a high-power mode, such data can be streamed in real time during specific exercises, and can be used to provide real-time feedback to the user based on step-count and the heart-rate variation, or sync ambient music based on the intensity of exercise.

This is a neat example of situations where long-term storage as well as real-time communication play an important role. In order to conserve energy, typical commercial devices only transmit the data every few minutes[3]. Such periodic data-syncing can be performed with any custom protocol without any real-time requirements. However, for the specified high-power mode, the device ought to switch to 'communication' mode than just a 'data-sync' mode to be most efficient.

[3] A FitBit syncs with a smartphone every 15 minutes. The developer API for FitBit also provides a subscription API for any data changes.

Leaving the complexity of implementation aside, a privacy focused user has to give away all the data to FitBit, when the only useful service provided by their servers is data-storage. Using STS for such situations provides end-to-end security and integrity, where the device manufacturer provides a highly available storage service, but the data belongs to the users.

2.2.3 Beyond a log interface: CAAPIs

Although an STS abstraction shelters developers from low-level machine and communication primitives, many applications are likely to need more common APIs or data structures. In fact, STSs are sufficient to implement any convenient, mutable data storage repository. A CAAPI can present a key-value store, file system, or database interface. Because STS serves as the ground truth, the benefit of consistency, durability, scalability, and availability are carried over to CAAPIs for free. **Add more details about CAAPI**

2.3 Threat Model

In summary, the general philosophy for STS is to be able to operate without a single point of trust in the system, be it STS-servers, network providers, or some other third party. In order to achieve the useful goals of a time-shifted secure channel, STS ought to provide at-least two properties: confidentiality and integrity. Confidentiality (data secrecy) is broadly achieved by using encryption on the data, while integrity is achieved using a combination of digital signatures and cryptographic hashes.

A secure system should provide 'availability' as an additional property, which translates to access to old data and data-freshness problem for STS. STS by itself does not and can not provide strong availability guarantees. As a building block of a larger system (ExP), STS can be used to provide confidentiality and integrity while other techniques (such as replication) are used for providing availability. We do, however, provide a mechanism for data-freshness in the form of 'heartbeats', that we discuss in Sec. 3.

Even though the scope of potential adversarial actions is very broad, we summarize the most common threats as follows:

2.3.1 Network level

STS is a secure stream of individual records over an insecure channel. With no security assumptions from the underlying networking layer, a non-exhaustive list of attacks a malicious third party can mount: active or passive man-in-the-middle (MITM) attacks, replay attacks, using signed items in different contexts than an original signer intended, etc. In that sense, threat model for STS is similar to that of TLS protocol (Transport Layer Security).

2.3.2 Storage level

A malicious or corrupt STS-server can try to modify stored data. Since the STS is a linked list of records, integrity of STS roughly translates to two concepts: individual record's integrity and the ordering of the records. Thus, an STS-server can violate data-integrity not only by changing on-disk data for individual record, but also by re-ordering records. Readers of the data ultimately trust only the writer of data, and no other single entity in the infrastructure.

Data integrity can be challenged by not only a malicious STS-server, but unrelated third parties by attempting to write to an STS without authorization, by replaying signed data for example.

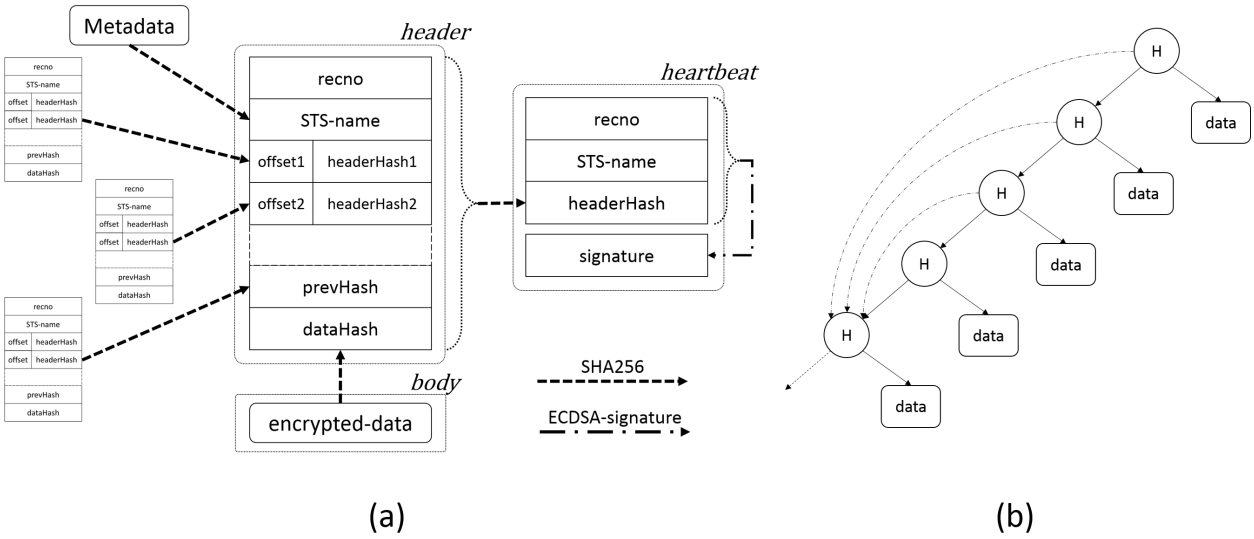

Figure 3: (a) Record structure demonstrating record-header, body and heartbeat. (b) Without hash-pointers *(offset, headerHash)*, the records in an STS make a skewed Merkle tree. Hash-pointers include additional links and transform the Merkle tree to a Directed Acyclic Graph (DAG).

Third parties can also try to gain more information by reading encrypted records in an STS without access to the appropriate decryption key.

In addition, a malicious STS-server can perform a wide variety of denial-of-service attacks, such as denying legitimate append-requests, denying read-requests from readers by either providing stale data or truncating an STS prematurely. We consider the denial of service class out-of-scope of the current work.

3 STS: Internals

In this section, we describe the internal mechanisms that provide the desired security properties in presence of the threat model described above. For concreteness, we developed a proof-of-concept version of STS in Python using Google's Protocol Buffers as a portable data-structure schema. Note that unless otherwise specified, 'hash' refers to a SHA256 hash function. Also, 'signatures' refer to ECDSA instead of a non-EC algorithm such as RSA or DSA for the reasons described in [3].

Names and Addresses: As we have mentioned earlier, STSs are identified by a 256-bit flat identifier called the STSname derived from a hash of STS-metadata. Not only STSs, but other entities (clients, STS-servers, etc) are also addressed using similar flat 256-bit identifiers. Each such entity (or 'end-point') also has an associated metadata that contains a public signature key, along-with other user-defined items. The 256-bit name/address of end-points is derived in the same way as an STS, i.e. by performing a SHA256 hash over the metadata. To avoid overloading the term 'metadata' with the STS-metadata, we will explicitly use the term 'end-point metadata' where it is not clear from the context.

3.1 In-memory Representation

In this section, we describe the various data structures as they are represented on various end-devices (clients, STS-servers, etc.)

3.1.1 STS-Metadata and Ad-Cert

Metadata is a special type of immutable structure at the beginning of an STS. Metadata is essentially a signed list of key-value pairs, describing essential information about the STS. A non-exhaustive list of such information is a public signature key for the signing records ('writer-key'), a public signature key of the STS-creator (if different from 'writer-key'), a source for obtaining decryption key for the data, creation time, truncation policies, a human readable name/description of the STS, etc. The signature over a serialized version of this list of key-value pairs is performed by the 'creator' to create the metadata, and the SHA256 hash of this immutable metadata is used as the name for the STS.

At the very minimum, metadata contains the 'writer-key', which serves the following purposes: (1) individual records are signed with the private part of 'writer-key', providing data-provenance and non-repudiation properties; (2) a write-access control mechanism for a correctly functioning STS-server, where a signature associated with an 'append' is validated against the 'writer-key'. To ensure the single-writer semantics with data-ordering decided on the client side, the private part of this 'writer-key' ought to be protected by the designated writer and should not be shared.

Why use signatures for write access control? Using signatures to maintain write access control is computationally expensive. However, it allows a reader to detect any malfunctioning/malicious STS-server. Further, it requires minimum state sharing across STS-servers, which is useful in case the STS is hosted collectively by more than one STS-server. In addition, signatures are needed to provide data-provenance anyways.

The process of creating a new STS involves sending this signed metadata and an 'advertisement-certificate' (Ad-Cert) to an STS-server. An Ad-Cert is a signed statement by the STS-creator that a particular host with a 256-bit address can host/serve an STS for a certain time-duration. In case of federated collection of STS-servers owned by a service provider, the metadata and Ad-Cert are sent to a designated 'creation-service' instead of a single STS-server. The Ad-Cert grants the service provider a capability to host an STS. In some ways, the 'creator' acts as a Certificate Authority issuing a certificate to a particular STS-server to serve read requests on behalf of the STS.

Why associate the physical host with STS? As we will describe in detail later, any operation performed by a writer ought to be acknowledged securely. Otherwise, an active MITM can simply drop any 'append' operations and send a spoofed acknowledgement to the writer. Ad-Cert allows a writer to ascertain that it received an acknowledgement from the actual STS-server and not from a MITM. An alternate strategy could be to include the designated STS-server in the metadata itself, which could work in certain situations. However, because the metadata is immutable, this alternate strategy has the downside of fixing the STS to a particular STS-server for eternity.

3.1.2 Records

As mentioned earlier, records are the unit of read/write to a log, and STS is primarily an ordered list of records with the metadata at the beginning. A record is an immutable structure composed of a header and the body, with an associated heartbeat (see Fig. 3).

Record body contains the application level data. This data is opaque to the STS-server (or any other intermediate entity) and is padded and encrypted using (preferably) a fast, symmetric encryption scheme, such as AES. For our proof-of-concept implementation, we use AES-128 in CTR mode, where the counter is initialized from the STS-name and `recno` (see below). Using CTR mode enables a reader to individually decrypt records and reduces management overhead by deriving an IV implicitly from the other already available information. The decryption key is communicated to the allowed readers using either an 'STS-attach request' (described later), or any out-of-band

communication channel suggested in the STS metadata.

Record header contains meta-information necessary for integrity verification, data-ordering and storage. At the very essence, the header contains a monotonically increasing integer (`recno`), the 256-bit name of the STS (`STSname`), a variable number of *hash-pointers* to older records in the form of `(offset, headerHash)` pairs, the hash of most-recent record header than the current one (`prevHash`)[4], and the hash of the record body (`dataHash`). Within the namespace of an STS, a record could be referenced either by `recno` or by the hash of the header `headerHash`.

Record heartbeat is a signed stand-alone piece of data associated with a record that contains `recno`, `STSname`, the corresponding `headerHash` and a signature over the three items using the private part of the 'writer-key'. Heartbeats are small, self-sufficient pieces of data that can be distributed widely in a network (because of the small size) and have a the same built-in ordering as the records (because of the included `recno`). The purpose of 'heartbeats' is to provide data-freshness guarantees and authenticity of the corresponding record.[5] However, as we will describe shortly, signature verification using 'heartbeats' is not the most optimal way for reading old data.

To 'append' data to an STS, a writer forms the appropriate record structure and sends it to the STS-server along with the signature included in 'heartbeat'. The STS-server performs a basic sanity checking on the received record structure and verifies the signature before accepting the 'append'. Subscribers are notified of the new data using 'heartbeats' either by the STS-server, or a separate highly available 'heartbeat-server' (whose only job is distribution of heartbeats). Readers query records either by their `headerHash` or `recno`. In case of query by `recno`, the STS-server provides an integrity proof relative to a more recent record that the reader is aware of.

A Merkle tree: Ignoring the *hash-pointers* for a moment, the record structure described above is essentially a very skewed Merkle tree, with each new record as the root of a new tree (see Fig. 3(b)). Provided that the root of the Merkle hash tree is known, integrity of any individual node can be verified easily by traversing the hash tree. The depth of the tree is equal to the number of records, which makes for very long proofs. In order to limit the size of proofs, we introduce *hash-pointers*, which are pointers to any arbitrary record older than a given record. This makes the graph of pointers a Directed Acyclic Graph (DAG) instead of a simple tree. Using arbitrary *hash-pointers* allows us configurability of the structure of a given STS for the desired read or append performance.

Integrity verification: hashes or signatures? Even though a 'heartbeat' is associated with each record, signatures are orders of magnitude more expensive to compute and verify[6]. Because of this heavy cost, we use hashes as much as we can for all integrity verification. With the design of a record header described above, one single signature verification at a particular instance in time allows a reader to verify everything up to that time in the past with only hash-verification. Additionally, because old-data can be 'expired' using log-truncation, we'd like to make sure that the integrity verification is tolerant to missing data before a certain point in time.

Why not make a balanced Merkle tree? Instead of using *hash-pointers*, an obvious design choice would have been to make a balanced Merkle tree rather than a skewed tree. We do, in-fact, use *hash-pointers* to create a rather balanced Merkle tree for certain use-cases. However, the reasons to not go with only a balanced tree approach are: (1) it does not work very well with STS-truncation (imagine a situation where we just like to keep last 10 records around); (2) extra data transfer in the simplest of the cases (a skewed tree is extremely efficient on network for range queries, as we will show later); (3) Merkle trees relatively require larger state to be maintained on writer, which

[4]For the first record, `prevHash = STSname`

[5]Note that heartbeat is different than a typical *keep-alive* used in various protocols.

[6]One-time signature schemes are much cheaper than traditional digital signatures, however they suffer from excessively large data-sizes. Our space-time equation can not be too biased towards data-size, because it needs to be transferred over network.

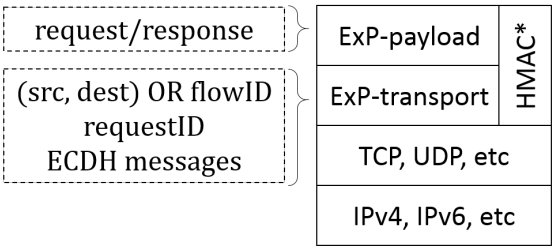

Figure 4: ExP protocol layered on top of TCP/UDP or other transport protocol. ExP protocol is split into two parts: transport and payload. An HMAC is calculated for acknowledgements back from the server.

may not be available on all devices.[7]

Generating integrity proof in a DAG: When queried for a record m against a more recent record n (that a reader may have obtained by verifying a signature from a 'heartbeat'), an STS-server has to find the shortest path from n to m in a weighted DAG, where the weight on an edge is calculated as the size of the record-header where the edge is pointing to. However, for most practical purposes, a simple greedy strategy works well enough, where starting from n, one picks an edge to the oldest record more recent than m.

Choosing the 'hash-pointers': **could be improved** Depending on the usage scenario, the performance of an STS can be tuned by choosing an appropriate strategy for 'hash-pointers'. Typically, it's a tradeoff between the cost of 'append' and integrity proofs for 'read'. The simplest strategy without any 'hash-pointers' results in a highly skewed Merkle tree (essentially a linked-list), where the integrity proof is as long as the number of records between the queried record and an already known record. However, this simple linked-list design is very efficient in range queries (a range of records in a linked-list design is self-verifying with respect to the newest record in the range).

An very powerful strategy could be to choose 'hash-pointers' in combination with CAAPIs. For example, a file-system CAAPI may include a checkpoint record and future records point to such a checkpoint-record. Another general strategy would be to use a skip-list style linking using 'hash-pointers' for quick and efficient searches. However, one can always hand-tune a specific design for specialized use-cases.

Naively sending the records over network can lead to quite a large overhead in terms of bytes transferred. Especially for embedded devices with constrained energy supply, network communication is very expensive. Hence, we need to be efficient in (1) sending only the necessary information, and (2) minimizing the size of proofs. To encode the information and ensuring the security of the associated operations, we use the ExP protocol that we describe in the next section.

3.2 On Network: ExP protocol

In this section, we describe how the data-structures described in previous section are sent over network securely using the ExP protocol. Even though the STS can theoretically be an entirely application level construct, we implement it directly on top of transport layer (TCP/UDP). From an OSI layer viewpoint, it sits at the same level as SSL/TLS (see Fig. 4). An alternative could have been to overlay ExP protocol over TLS, however that introduces inefficiencies because of duplication of effort.

TCP vs UDP: Even though sending PDUs (Protocol Data Unit) over a reliable transport pro-

[7]This becomes especially important in case of commercially available microchips with hardware ECDSA support but very limited storage, e.g. Atmel ATECC108A that ships with only 10Kb EEPROM.

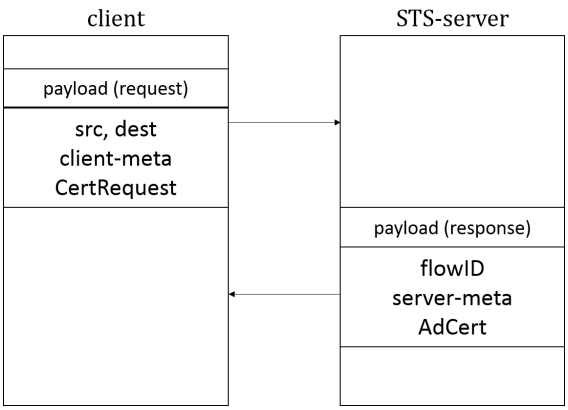

Figure 5: ExP-transport level data exchange. Note that 'client-meta' is the end-point metadata for the client, and 'server-meta' is the end-point metadata for the STS-server. These are used to establish a shared secret using ECDH, and are different than the STS-metadata.

vided by TCP is the easiest, we also need to account for tiny sensors sending small data via UDP. Even with the lack of reliability and congestion control, small enough PDUs that fit in a single datagram can be sent over UDP without any changes. Our design makes sure that the extra bytes per request than the actual payload are minimal, which makes for a wider range of applications to be able to use UDP as the transport mechanism without worrying about fragmentation issues.

The ExP protocol is loosely split into two parts: Transport level and Payload level. ExP-transport primarily contains 'source' and 'destination' information, a requestID and an EC Diffie Hellman key-exchange between server and client, while the ExP-payload contains the actual request/response to/from an STS-server. Finally, there is an HMAC using the established DH key over ExP-transport and payload for acknowledgements from the STS-server. Separating transport level from payload enables us to perform key-setup in parallel with actual data transfer (see below for details). This is a notably different design then, say TLS, where key-exchange happens requires multiple round-trips before any actual data is sent.

Why both source and destination? A single STS-server can host multiple STSs. Since most of the operations are directed to a particular STS, a PDU (Protocol Data Unit) needs to have a 256-bit 'destination' identifying the exact STS. Further, ExP protocol is primarily an overlay protocol not tied to the underlying IP layer. Including a 'source' address allows us to achieve greater mobility without any effect on the communication, even when underlying IP addresses change. Additionally, server acknowledgments to clients should be acknowledged securely; as we will describe later, we bind the acknowledgements to a specific 'source' and 'destination' pair to prevent replays.

3.2.1 Transport functions: FlowID and key setup

Including two 256-bit addresses in each PDU is a significant data overhead. For efficiency purposes, only the first 'client → server' PDU contains the full source and destination address (the client's and STS's address, respectively). Any immediate 'client → server' PDUs contain a 128-bit MD5 hash of 'client-addr, STS-addr', called a flowID. Similarly, all 'server → client' PDUs contain the MD5 hash of 'STS-addr, client-addr' as the flowID. The use of cryptographically insecure MD5 is only an optimization, it could as well be a random flowID assigned by the client and/or server. In fact, for acknowledgements from the server, the HMAC is calculated over the original source and destination address (and not the flowID).

The other important thing that ExP-transport accomplishes is validating the STS-server, and establishing a shared secret key that the remote server uses to securely send acknowledgements back to the client. Imagine a situation where a client 'appends' data to a log, but a malicious man-in-the-middle drops the data, and instead issues a spoofed acknowledgement. To prevent this, an STS-server can digitally sign the acknowledgements using it's own private key (which is different than any key associated with the STS). The client needs to (1) obtain the public key, and (2) verify that this key belongs to an STS-server designated to host the STS in question. As an optimization, instead of using digital signatures, the client and the server establish a shared secret key using EC Diffie Hellman, which is then used by the STS-server to perform an HMAC over acknowledgements.

For server validation and key-setup, the client sends it's own endpoint-metadata (which includes an EC public key)[8]. The STS-server provides an Ad-Cert that it presumably received from the STS-creator at creation time (which includes the 256-bit name of the STS-server), and an endpoint-metadata information block containing the STS-server's public key; this endpoint-metadata should hash to the 256-bit name contained in the Ad-Cert. Assuming everything goes well, both client and the server have each other's EC public key, which they use to derive a shared secret using ECDH, which is further used for the HMAC as we described above.

Another important bit of information included in ExP-transport is a requestID, which is a unique 32-bit monotonically increasing number that (1) the client uses to correlate responses with requests, and (2) ensures an acknowledgement by the server for one request can not be replayed as a response to another request.

3.2.2 ExP Payload

ExP-payload contains compressed, network efficient versions of 'STS-create', 'append', 'read', 'subscribe' and their corresponding responses. With the key idea being not to transmit information that can be securely generated on the other side, it is just a careful encoding of information. However, note that any signatures/hashes are over a serialized version of the in-memory representation, and not the on-the-wire bytes. This makes sure that any accidental/intentional incorrect assumption of shared state does not compromise security.

Log creation request/response: A client sending signed metadata and Ad-Cert as it is, and expects a signed acknowledgement. There is no room for optimization for 'create'.

Append request/response: Append request involves a client creating a header with appropriate hashes and a signed heartbeat. However, the actual data sent to the server does not contain any hashes, since they can be generated locally by the server (only offsets for 'hash-pointers'). On receipt of a log-append request, the log-server fills in the appropriate hashes that the client skipped transmitting on the wire, and then performs signature validation. If everything checks out, a signed acknowledgement is sent back.

Read/Multiread request/response: A client can read a record either by 'recno', or by the corresponding 'headerHash'. For a client with no information about a log other than the log name and appropriate decryption key, integrity verification is a challenging problem; it must acquire the most recent 'heartbeat' either from an STS-server or a separate 'heartbeat-server'. A server provides the actual data and the integrity proof in case of a request by 'recno', or simply the header and body in case of query by 'headerHash'.

Subscribe request/response: Subscription can be classified in two broad categories: (1) mere notification of new records, and (2) notification of new data as well as the data itself. A client includes the type of desired subscription in the subscription request. For clients interested in mere

[8] we assume that client and server already agree on ECDH shared parameters

notification of data, any new 'append's to the STS result in the associated heartbeat being sent out to the subscriber. The client can then optionally query for the individual records as and when desired. For clients interested in real-time data delivery, both the heartbeat and the record are sent in a single response.

3.2.3 STS-attach request

An STS-attach request is a special type of message that is not sent to an STS-server, rather to an end-point (a service, application, user). STS-attach request provides for a way to perform in-band key exchange, and acts as a glue to enable composability of STSs without compromising on security. It could be sent by an STS-creator, an administrator, or anyone else with the appropriate capabilities.

STS-attach request uses the general ExP-protocol, and proceeds with a DH-key exchange as in the regular ExP protocol. However, the key is used to encrypt the payload instead of just calculating an HMAC. An STS-attach request specifies a number of input and/or output STSs, along with the necessary decryption keys required to access the encrypted data, all encrypted with the shared secret key.

3.3 On disk storage on STS-server

An STS-server is free to choose the data representation as it seems fit, and is pretty much dependent on the specific implementation. The only contract it has to satisfy is to be able to serve the requests that are not older than the truncation threshold.

A single STS-server can not store arbitrarily large STS. For this reason, we introduce the concept of 'extents', a large enough chunk of data representing a range of records in an STS that can be independently replicated or migrated. Extents are opaque to the clients; an STS-server may move certain extents to remote systems or slower but larger storage media depending on the access pattern. The append-only design ensures that old data is read-only, making way for easier replication/durability of data.

Extents could be used for optimization of STS-truncation. Since truncation is merely a 'safe-to-delete' flag anyways, instead of doing per-record cleanup, an STS-server can perform such cleanup at the extent level.

An important tradeoff that an STS-server has to make is the storage vs computation costs for extents. In particular, the question is: whether to store record hashes along with the data or regenerate them when an extent is loaded into the memory? With a compaction similar to the data transferred over network, storage overhead of hashes can be as small as a few bytes per record.

3.4 Discussion

3.4.1 Query by Time

A much desired feature in IoT landscape is querying for information by time (or a range of time). We refer to 'time' as the notion of time relative to the STS-writer.[9] An encoding of timestamps of appropriate resolution into monotonically increasing integers representing record numbers achieves this property of query by timestamp with only a minor change to the read-requests.

However, a naive encoding of timestamps leaks quite a bit of information to a 3rd party without a decryption key. The problem is worse for STS than a purely communication based streams, because the data storage aspect of STS increases the window of opportunity for a malicious 3rd

[9] Issue of absolute correctness of time on the STS-writer is out-of-scope of the current work.

party interested in this information to more than just the duration of communication. In order to achieve some confidentiality, an Order Preserving Encryption scheme can be used to separately encrypt the record numbers before creating 'append' without any loss of functionality.

3.4.2 Encryption Key rotation

The decryption keys are capabilities for reading the data. It is a reasonable assumption that not all readers with access to a certain STS will be absolutely secure and trustworthy to protect the decryption keys. Especially for long-running communications, this can lead to serious data compromises. To alleviate this inevitable issue, we recommend that the decryption keys be rotated periodically. The new keys can either be generated by the writer itself and communicated to the designated readers using the STS-attach request, or it can be distributed by a key-broker (indicated in the metadata) to both the writers and readers. Such key-rotation mechanisms can be used to provide selective access to ranges of data, or implement a revocation scheme where future access to data is denied.

3.4.3 Truncation

STS-truncation is marking old data 'safe-to-delete'. The notion of 'old' can be either described with respect to time, or number of records. Ensuring data is deleted from remote storage is probably an unsolvable problem; truncation only provides a hint to the server that the data is safe to delete. STS-truncation significantly affects the design choice for *hash-pointers* in the record headers and one of the reasons a simple balanced Merkle tree is not the optimal choice for certain use-cases.

3.4.4 Transient STS

Taking STS-truncation to an extreme level, a creator can designate an STS to only require a small number of records. Such kind of STS can be purely held in memory of a collection of collaborative STS-servers, without ever needing to commit data to a physical disk. Such optimizations can improve the performance significantly, extending the communication potential of STSs.

3.4.5 Signing frequency

Signature creation/verification is one of the most expensive operation for writers and readers. Even though more and more devices include hardware-support for cryptographic operation, certain low-end devices might not be capable of (or need) such real-time signing. A tuning parameter for STS is the signing frequency, where a writer does not sign each individual record as it is generated. Rather, every *N-th* record is signed by the writer. Such optimizations are very well suited to low-power radio based sensors that send multiple records in batches to conserve power.

4 Evaluation and Analysis

As mentioned earlier, our unique combination of communication and storage makes comparison with pre-existing tools challenging. Because our target audience has a wide variation in storage/computation capabilities, we need to make sure that the most minimal devices are capable enough to participate and communicate with others.

The overhead of our design can be evaluated against several metrics:

Integrity verification cost: Any record arrangement scheme should provide for a way for the reader to perform integrity verification starting from a signature corresponding to the *most recent*

record. A simple linked list of records requires a reader to traverse the records one-by-one until it reaches the desired record. Integrity-verification overhead on reader's end can be measured in the size of data transferred, number of signature verifications, and number of hash calculations.

Writer overhead: Since the ordering is performed by the single-writer by inserting hash-pointers and performing signatures, it requires the writer to maintain some state. For a simple linked list of records, this state would be a single hash corresponding to the most recent record, which is inserted in a new record. Further, the writer needs to perform some signing and hashing operations, which adds to the cost of appending a single record.

STS server overhead: An STS-server needs to store the actual contents of a record. Any additional bytes other than the actual data add to the storage cost. Depending on the average record size, this overhead may or may not be significant. In addition, an STS-server also needs to perform some signature and/or hash verifications in order to accept or reject append requests.

4.1 Security Overhead

STS provides additional benefits of data storage and non-repudiation than just TLS. Hence a comparison against just a transport level security scheme such as TLS is unfair. Further, STS provides storage as well, which is absent in TLS. Comparison just against a storage system is not a very good case, since any communication with a storage repository has to have some transport level security anyways.

In order to measure security overhead of STS, we compare it against two system designs: 1) the traditional hop-by-hop security where a sensor and an actuator communicate with a storage repository using TLS, 2) A modified version of the above by adding confidentiality and data-repudiation guarantees that STS provides. This is achieved by performing encryption and digitally signing the data. More specifically, we have a 'writer' appending data to an STS hosted on an STS-server, or sending data to a storage repository. In each case, we also have a reader that 'subscribes' to the data or reads data in bulk. For various cases, we measure the number of bytes transferred over the network, data stored on disk, and cryptographic operations performed. Note that a fair evaluation is complicated by the wide choice of4d7766189e82c1381774dc840d05267b. cipher-suites and configuration parameters for TLS. Hence, the numbers should be used just as a guiding principle rather than an absolute benchmark.

fix after this ==========

Motivated by our application scenarios in Sec. 2.2, let us consider an environmental monitor transmitting small amount of data (20 bytes) at fixed intervals.[10] In order to compare the performance of STS, let us evaluate what would be the data sizes for various scenarios per data generation.

Note that session establishment and key-exchange is a significant overhead. However, it's a one-time thing (and depends on so many factors, such as certificate size), so not really a good metric for comparison. **elaborate on why we skip the session stuff. But still, give a number**. The parameters we use are TLS-ECDH-ECDSA-WITH-AES-128-CBC-SHA256 with TLS 1.2. The curve of choice for EC is 'sect283r1' for both. The choice was fairly arbitrary, and should not affect things too much. **say something here**.

For the case of STS using a simple linked list design (i.e. no hash-pointers), and using the timestamp in the `recno`, a writer needs to send about 157 bytes of data per append. A breakdown of this is as follows (note that the numbers are approximate, because of Protobuf): 16 bytes for flowID, 4 bytes for requestID, 1 byte for command, 8 bytes for recno (and timestamp), (20+12) bytes

[10] Most Bluetooth Low Energy (BLE) devices using GATT are limited to 20 bytes per transmission.

for data, 78 bytes for signature and roughly 18 bytes for Protobuf overhead. An acknowledgement back from the server is about 66 bytes. Breakdown of that is: 16 bytes for flowID, 4 bytes for requestID, 1 byte for ack, 32 bytes for HMAC, and roughly 13 bytes for protobuf overhead.

In case of simple TLS, let's assume that the data sent to the server is "timestamp + data". Data sent to the TLS server is 85 bytes, breakdown is: 5 bytes for TLS record parameters, (20+8) bytes for data, 32 bytes for MAC + padding (to make it multiple of 16). As it is, there is no acknowledgement back from the server.

For the version where the data is signed and then encrypted, it's be 28 bytes for data + timestamp. **not including the identity of device in signed data, but whatever**. If we were to do sign-then-encrypt (recommended), then it'd be 28+78=106 bytes for data + signature, which when padded would be 112 bytes. IV storage is another issue. This 112 bytes is sent through TLS, which would lead to another layer of encryption (adding another block of 16 bytes) + 32 bytes of HMAC + 5 bytes of TLS overhead = 165 bytes.

In terms of cryptographic operations for STS, the most time consuming operation is the signature. To be honest, it outweighs everything so much that we don't even need to talk about other things. But the somewhat comparable TLS + NR design involves encryption as well.

The data sent to a subscriber is the same as well in each case. **Really?**

4.2 Security argument

With complex protocols, it is hard to achieve provable security. However, we provide arguments for why we believe it achieves the desired security properties.

4.2.1 Confidentiality

The body of a record, where the actual application data lives, is encrypted with a key unknown to anyone else than the readers and the writers. As long as this decryption key is kept safe, confidentiality of individual record can be guaranteed except for the size of individual records. However, just encrypting individual records is not sufficient, since the size and timing information about a data-stream is leaked to an adversary monitoring communication in real-time.

The information-leakage problem is especially challenging for STS because a 3rd party can monitor communication in real-time by just subscribing to the appropriate STS; they don't even have to be in a special position in the network to observe the traffic pattern. The time-shift nature and the storage aspect of STS also makes things easier for an adversary; an adversary doesn't even have to subscribe in real-time to get the metadata information. Instead, it can perform a bulk-read of records at a later time and get at the very minimum size information.

fix this

In real-world systems, this side-channel information leak can be partially alleviated by a smarter client and by distributing the trust on a collection of servers rather than just a single server. The solutions fall into two broad categories: obfuscation of real data by adding fake entries (either by the client itself, or a proxy in the path between a client and a server), or using 'mixing', i.e. adding information from multiple clients together such that an untrusted server can not correlate the real information back to a single client. However, note that none of these systems change the server side operations in a fundamental way.

In the particular case of STS, a smarter client can use 'pointer-records' to overlay a single STS ('primary' STS) over a collection of 'secondary' STSs hosted on mutually distrustful servers. Pointer-records can be used to describe a schedule of N future records using a secure pseudo-random number generator (PRNG) initialized with a seed s that generates integer values in the

range $[1, m]$ (m is the number of secondary STSs). A pointer record contains N, s and STSnames for each of the m STSs, all this information encrypted using the encryption key that the writer would otherwise use to encrypt records in a normal STS.

Only a reader with the appropriate decryption key can decrypt the information in the pointer-record, and follow the pointers to the secondary STS to retrieve the real payload. An adversary subscribed to just the primary STS only obtains the information about new pointers, where the information leak can be minimized by randomizing N and applying a random padding to obfuscate size of pointer records. An adversary subscribed to just a single secondary STS can obtain the size and time of interleaved records, however the adversary does not know how many records it missed. A more powerful global adversary subscribed to a subset of all possible STSs can perform some correlation of individual STSs as constituent STS of an overlay STS, however it can only never be sure of the missing information. **fix this, elaborate on how much information is leaked**

4.2.2 Integrity

Data integrity can be verified in presence of adversaries with control over the routing and storage infrastructure, starting from a single 'heartbeat'. STS, at its very core, is a Merkle tree with each new record being the new head of the tree (see Sec. 3.1). Starting just from the given 'heartbeat', a reader can query the metadata for the STS and obtain the public signature key of the designated writer, which can be used to perform the signature verification on 'heartbeat'. From the verified 'heartbeat', a reader can obtain the hash of the corresponding record-header, which is the root of the Merkle tree with a path to each older record by following the hash chain. In addition, the hash of the encrypted data is contained in the header as well, thus enabling a reader to verify that the data was not modified by the untrusted infrastructure.

4.2.3 Availability

As we mentioned earlier, STS by itself does not provide any data/service availability guarantees. An STS-server can deny valid 'append' requests and ignore any 'read' or 'subscribe' requests. However, we ensure that such availability failures are detectable by a client.

In particular, secure acknowledgments for operations that involve a state change ('append' and 'STS-create') ensure that an active man-in-the-middle can not impersonate an STS-server, causing a writer/creator to assume that the STS has been changed. Regarding truncation, a truncation policy based allows an STS-server to 'expire' data older than a certain threshold (number of records, or time), which is easier to verify as well.

5 Related Work

TODO: Add missing citations. We can not add all previous work, so categorize it and provide a few notable examples in each category.

STS provides a secure unified tool for communication and storage, and it builds upon a wide variety of exiting work. However, various existing tools achieve our goals partially, but there is no single tool that combines all the desired features of STS. On one end of the spectrum, we have SSL/TLS that provide a secure communication channel over insecure channel. On the other end, we have systems and tools for storing data securely on untrusted infrastructure.

At the very core, STS is an Authenticated Data Structure [12]. Our data structure design is loosely based on such previous work, while being able to achieve properties such as (1)

publish-subscribe, (2) streaming support, (3) truncation of old data, (4) confidentiality (encryption/privacy/etc), (5) tunability for performance, (6) Data freshness. (7) easy to implement in a distributed fashion.

We also borrow ideas from other related systems such as Oceanstore, CryptDB, SUNDR, Plutus, Encrypted file-system, etc. However, these systems are focused on providing secure storage on untrusted infrastructure. However, we propose to unify storage and communication. There have been some efforts to demonstrate the performance benefits because replication of effort is avoided [8]. We take this to a step further and provide a unified tool that can be generalized and put to a much wider use. **need some more words to show enough value addition.**

Another related area of research is indirection of communication and sensor-virtualization. Systems such as i3 [11] have proposed redirection of communication as a tool to achieve mobility, while still keeping the same end-point address. The idea of using sensors and actuators as a virtual device, or sensor as a service has been proposed by others [7, 9]. However, we provide an end-to-end solution from a sensor to service to actuator. In addition, the previous work in this field is focused mostly for enabling heterogeneous infrastructure or achieving mobility. However, our virtual device model goes a step beyond by using the STS for a variety of other potential purposes such as intrusion-detection, firewall, access-control, etc all on the software service. We also propose it as a DDOS prevention mechanism. **Do we?**

6 conclusion

1. Presented the STS interface and how it can help alleviate the IoT security challenges, 2. Security analysis of communication and data storage aspects, 3. Emphasize that even though the design may not be perfect, the philosophy is sound.

References

[1] Internet of Things Top Ten: OWASP. `https://www.owasp.org/images/7/71/Internet_of_Things_Top_Ten_2014-OWASP.pdf`, 2014.

[2] Hackers Remotely Kill a Jeep on the Highway, With Me in It. `http://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/`, 2015. [Online; accessed 27-March-2016].

[3] RSA vs ECC Comparison for Embedded Systems. `http://www.atmel.com/Images/Atmel-8951-CryptoAuth-RSA-ECC-Comparison-Embedded-Systems-WhitePaper.pdf`, 2015.

[4] Samsung smart fridge leaves Gmail logins open to attack. `http://www.theregister.co.uk/2015/08/24/smart_fridge_security_fubar/`, 2015. [Online; accessed 27-March-2016].

[5] The government just admitted it will use smart home devices for spying. `http://www.theguardian.com/commentisfree/2016/feb/09/internet-of-things-smart-devices-spying-surveillance-us-government`, 2016. [Online; accessed 27-March-2016].

[6] Internet of Things security is hilariously broken and getting worse. `http://arstechnica.com/security/2016/01/how-to-search-the-internet-of-things-for-photos-of-sleeping-babies/`, 2016. [Online; accessed 27-March-2016].

[7] Alam, S., Chowdhury, M. M., and Noll, J. Senaas: An event-driven sensor virtualization approach for internet of things cloud. In *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on* (2010), IEEE, pp. 1–6.

[8] Bagci, I. E., Raza, S., Chung, T., Roedig, U., and Voigt, T. Combined secure storage and communication for the internet of things. In *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2013 10th Annual IEEE Communications Society Conference on* (2013), IEEE, pp. 523–531.

[9] Evensen, P., and Meling, H. Sensewrap: A service oriented middleware with sensor virtualization and self-configuration. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2009 5th International Conference on* (2009), IEEE, pp. 261–266.

[10] Ghena, B., Beyer, W., Hillaker, A., Pevarnek, J., and Halderman, J. A. Green lights forever: analyzing the security of traffic infrastructure. In *8th USENIX Workshop on Offensive Technologies (WOOT 14)* (2014).

[11] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. Internet Indirection Infrastructure. In *ACM SIGCOMM Computer Communication Review* (2002), vol. 32, ACM, pp. 73–86.

[12] Tamassia, R. Authenticated data structures. In *Algorithms-ESA 2003*. Springer, 2003, pp. 2–5.

[13] Whitten, A., and Tygar, J. D. Why johnny can't encrypt: A usability evaluation of pgp 5.0. In *Usenix Security* (1999), vol. 1999.