Defined-Trust Transport (DeftT) Protocol for Limited Domains

Abstract

This document describes a broadcast-oriented, many-to-many Defined-trust Transport (DeftT) framework that makes it simple to express and enforce application and deployment specific integrity, authentication, access control and behavior constraints directly in the protocol stack. DeftT's communication model is one of synchronized collections of secured information rather than one-to-one optionally secured connections. DeftT is part of a Defined-trust Communications approach with a specific example implementation available. Combined with IPv6 multicast and modern hardware-based methods for securing keys and code, it provides an easy to use foundation for secure and efficient communications in Limited Domains (RFC8799), in particular for Operational Technology (OT) networks.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 2 April 2024.

Copyright Notice

Copyright (c) 2023 IETF Trust and the persons identified as the document authors. All rights reserved.
1. Introduction

Decades of success in providing IP connectivity over any physical media ("IP over everything") has commoditized IP-based communications. This makes IP an attractive option for Internet of Things (IoT), Industrial Control Systems (ICS) and Operational Technologies (OT) applications like building automation, embedded systems and transportation control, that previously required proprietary or analog connectivity. For the energy sector in particular, the growing use of Distributed Energy Resources (DER) like residential solar has created interest in low cost commodity networked devices but with added features for security, robustness and low-power operation [MODOT][OPR][CIDS]. Other emerging uses include connecting controls and sensors in nuclear power plants [DIGN] and carbon capture monitoring [IIOT].

While use of an IP network layer is a major advance for OT, current Internet transport options are a poor match to its needs. TCP generalized the Arpanet transport notion of a packet "phone call" between two endpoints into a generic, reliable, bi-directional bytestream working over IP's stateless unidirectional best-effort delivery model. Just as the voice phone call model spawned a global voice communications infrastructure in the 1900s, TCP/IP's two-party packet sessions are the foundation of today's global data communication infrastructure.

Yet "good for global communication" isn't the same as "good for everything". A significant number of OT uses can be characterized as Limited Domains [RFC8799]: localized and communication-intensive with a primary function of coordination and control and communication patterns that are many-to-many. Implementing many-to-many applications over two-party transport sessions changes the configuration burden and traffic scaling from the native media's $O(n)$ to $O(n^2)$ (see Section 1.2). Further, as OT devices have specific, highly prescribed roles with strict constraints on "who can say what to which", the opacity of modern encrypted two-party sessions can make it impossible to enforce or audit these constraints.

This memo describes Defined-trust Transport (DeftT) for Limited Domains [RFC8799] in which multipoint communications are enabled through use of a named collection abstraction and secured by an integrated trust management engine. DeftT employs multicast (e.g., IPv6 link-local [RFC4291]), a distributed set reconciliation communications model, a flexible pub/sub API, chain-of-trust membership identities, and secured rules that define the local
DeftT's intended use is for communications in the constrained environments that can be characterized as Limited Domains (LDs). Though the properties enumerated in [RFC8799] are necessary, we do not claim they are sufficient.

![Figure 1: DeftT's place in an IP stack](image)

DeftT is IP-compatible but not Internet-compatible (e.g., not routable). In contrast with IETF standards track protocols like the client-server COAP [RFC7252], DeftT is intended to serve the communication needs of a closed community with common objectives, a zero-trust Limited Domain (trust domain). Foremost among those needs is the ability to enforce community-specific policy constraints ("who can say what to which"). ABAC (Attribute-Based Access Control) [NIST] provides a model sufficient to express and enforce these constraints but a fundamental architectural choice remains to either:

(a) Start with Internet-based communication protocols then "harden them" by layering an ABAC framework on top, or

(b) Start with an ABAC framework that verifiably enforces the policy constraints then augment it with the minimum necessary communication primitives needed to function in a community's deployment environment.

Existing IETF protocols use approach (a) and, given how few enforceable security policies are possible on the open Internet, it's a reasonable choice. For LDs, approach (a) imports all the (otherwise unneeded) Internet abstraction maintenance machinery...
(DHCP, DNS, CAs, PDPs/PIPs, routing, address plans, etc.). When communication is expressed in terms of Internet abstractions (e.g., a TLS connection between two IP endpoints), there needs to be a translation layer to map between these abstractions and the community's entities, requirements and objectives. All this machinery is configuration intensive and recent history has demonstrated that it's all prime attack surface. DeftT has been created as a self-contained ABAC framework where the PEP and PDP are in the transport narrow pub/sub waist and embeds the PIP function in certificate signing chains so it's self-authenticating and self-distributing. Further, DeftT's efficient use of its communications schema obviates the ABAC expectation that "the more granular the controls, the higher the overhead."

Like COAP/OSCORE nodes, DeftT members start with a pre-existing identity obtained out of band which means that existing and evolving bootstrap and enrollment protocols and methodologies can be used. But DeftT identities are more than a single key pair and only convey membership in a specific Trust Domain that is using a particular set of rules and a particular trust anchor (TA). Member identities are in the form of certificate chains containing all relevant attributes or roles with a private key corresponding to a unique identity cert at the chain's leaf. As with MUD, members of a trust domain have specific capabilities and permitted communications that are explicitly specified. Unlike MUD, each member gets the communications rules for the domain distributed in binary form in a cert signed by the same trust anchor that is at the root of the member identity. This schema specifies the format for membership identity chains as well as the format of all legal communications and the attributes required to issue them. Each DeftT has an integrated trust management engine that makes use of the schema at run-time. DeftT enrollment consists of configuring a device with identity bundles that contains the trust anchor certificate, a compact and secured copy of the communication rules, and a membership identity (for domain communications) which comprises all the certs in its signing chain (used to confer attributes) terminated at the trust anchor. The private key corresponding to the leaf certificate of the identity should be securely configured (i.e., not exposed to any third party) while the security of the identity bundle can be deployment-specific (i.e., the public certificates it contains may optionally be protected from third parties). The identity chains of all communicating members share a common trust anchor as do the rules that define the permissible signing chains of a Domain, so the bundle suffices for a member to authenticate and authorize communication from peers and vice-versa. The identity bundle, DeftT's trust management engine, and a trust domain's certificate collection (obtained via DeftT as members initiate connection) allow new members to join and communicate with no specific knowledge of other members, thus obviating labor
intensive and error-prone device-to-device association configuration. (More on certificates and identities in DeftT in Section 2.6 and Section 4.)

In synchronized collections, members communicate about their local version of the collection state and send additions to the collection the other members are missing. Along with the out-of-band identity bundle, DeftT makes use of both its synchronized collections and its integrated trust management engine to securely join a particular Trust Domain. As understanding how a DeftT joins a Trust Domain should be helpful in understanding DeftT and how it differs from other approaches, its certificate distributor module is briefly described here. A state diagram of the joining process is Figure 2.

During configuration, the private identity key should always be stored so it is both private and secure. The identity key is not used for signing DeftT packets, but for signing a cert that is locally created so that signing certs can be updated more often without the need for update of the identity. In addition, the identity key can remain within protected hardware like a TPM for signing while the signing key is intended for use in the communications path where we can tradeoff the possibility of more exposure vs the need for speed. Once the signing pair is created and a cert signed, the DeftT starts the process of joining the Trust Domain by subscribing to the Domain certificate collection whereupon it receives the state of the collection. The local signing chain is added to the local copy of the cert collection then the local collection state can be compared to the received collection state and any certs that are not already in that received state will be sent on the network to be added to the Domain collection. Note this will always include the identity cert and the signing cert of a new member, but other certificates of the chain may already have been added by previously joined members. A DeftT does not consider itself joined until it receives a collection state from the network that contains all of its certs, indicating that at least one other member will be able to receive its signed packets. Whether joined or not, the cert distributor handles all certs received from the network, adding them to its local collection when an entire validated signing chain is received.
Figure 2: DeftT certificate distributor enables joining Trust Domain

OSCORE [RFC8613] adds object security to COAP specifically to get around the vulnerability of using only DTLS/TLS with proxies. OSCORE uses pre-shared keys either acquired out-of-band or via a key establishment protocol. OSCORE encrypts/signs a COAP message and carries it as payload in a COAP message with the OSCORE option. A Security Context is between two endpoints, specific to sender ID and recipient ID. Sender IDs may be establish o-o-b. As Internet compatible protocols, COAP/OSCORE/ACE [RFC9200] use 1) cleartext options in their headers and 2) trusted third parties or resource servers, both of which can be exploited. A DeftT PDU uses a hash of its compiled rules cert to identify its trust domain with no options. In the Internet, PDU headers tell nodes how the packet should be handled. In a DeftT trust domain, the hash in the PDU identifies the packet as part of the domain whose rules will be enforced by any receiver. These are very different architectures both for communicating and for securing communications and are expected to serve different roles although the application spaces may overlap. Further, DeftT and Defined-trust Communications are early-stage work compared to COAP/OSCORE and other IETF work, but deployments are underway by Operant and Pollere.

1.1. Environment and use

Due to physical deployment constraints and the high cost of wiring, many OT networks preferentially use radio as their communication medium. Use of wires is impossible in many installations (untethered Things, adding connected devices to home and infrastructure networks, vehicular uses, etc.). Wiring costs far exceed the cost of current System-on-Chip Wi-Fi IoT devices and the cost differential is increasing [WSEN][COST]. For example, the popular ESP32 is a 32bit/320KB SRAM RISC with 60 analog and digital I/O channels plus complete 802.11b/g/n and bluetooth radios on a 5mm die that consumes 70uW in normal operation. It currently costs $0.13 in small quantities while the estimated cost of pulling cable to retrofit nuclear power plants is presently $2000/ft [NPPI].

Many OT networks are Limited Domains having a defined membership and communications that are often local, have a many-to-many pattern, and use application-specific identifiers ("topics") for rendezvous. This fits the generic Publish/Subscribe communications model ("pub/ sub") and, as table 1 in [PRAG] shows, nine of the eleven most widely used IoT protocols use a topic-based pub/sub transport. For example MQTT, an open standard developed in 1999 to monitor oil pipelines over satellite [MQTT][MHST], is now likely the most widely used application communication protocol in IoT (https://mqtt.org/use-cases/). Microsoft Azure, Amazon AWS, Google Cloud, and Cloudflare all offer hosted MQTT brokers for collecting and
connecting sensor and control data in addition to providing local pub/sub in buildings, factories and homes. Pub/sub protocols communicate by using the same topic but need no knowledge of one another. These protocols are typically implemented as an application layer protocol over a two-party Internet transports like TCP or TLS which require in-advance configuration of peer addresses and credentials at each endpoint and incur unnecessary communications overhead Section 1.2.

1.2. Transporting information

The smart lighting example of Figure 3 illustrates a topic-based pub/sub application layer protocol in a wireless broadcast subnet. Each switch is set up to do triple-duty: one click of its on/off paddle controls some particular light(s), two clicks control all the lights in the room, and three clicks control all available lights (five kitchen plus the four den ceiling). Thus a switch button push may require a message to as many as nine light devices. On a broadcast transmission network each packet sent by the switch is heard by all nine devices. IPv6 link-level multicast provides a network layer that can take advantage of this but current IP transport protocols cannot. Instead, each switch needs to establish nine bi-lateral transport associations in order to send the published message for all lights to turn on. Communicating devices must be configured with each other's IP address and enrolled identity so, for \( n \) devices, both the configuration burden and traffic scale as \( O(n^2) \). For example, when an "all" event is triggered, every light's radio will receive nine messages but discard the eight determined to be "not mine." If a device sleeps, is out-of-range, or has partial connectivity, additional application-level mechanisms have to be implemented to accommodate it.

Figure 3: Smart lighting use of Pub/Sub

MQTT and other broker-based pub/sub approaches mitigate this by adding a broker where all transport connections terminate (Figure 4). Each entity makes a single TCP transport connection with the broker and tells the broker the topics to which it subscribes.
Thus the kitchen switch uses its single transport session to publish commands to topic kitchen/counter, topic kitchen or all. The kitchen counter light uses its broker session to subscribe to those same three topics. The kitchen ceiling lights subscribe to topics kitchen ceiling, kitchen and all while den ceiling lights subscribe to topics den ceiling, den and all. Use of a broker reduces the configuration burden from $O(n^2)$ to $O(n)$: 18 transport sessions to 11 for this simple example but for realistic deployments the reduction is often greater. There are other advantages: besides their own IP addresses and identities, devices only need to be configured with those of the broker. Further, the broker can store messages for temporarily unavailable devices and use the transport session to confirm the reception of messages. This approach is popular because the pub/sub application layer protocol provides an easy-to-use API and the broker reduces configuration burden while maintaining secure, reliable delivery and providing short-term in-network storage of messages. Still the broker implementation doubles the per-device configuration burden by adding an entity that exists only to implement transport and traffic still scales as $O(n^2)$, e.g., any switch publishing to all lights results in ten (unicast) message transfers over the wifi network. Further, the broker introduces a single point of failure into a network that is richly connected physically.

Figure 4: Brokers enable Pub/Sub over connection/session protocols

Clearly, a transport protocol able to exploit a physical network's broadcast capabilities would better suit this problem. (Since unicast is just multicast restricted to peer sets of size 2, a multicast transport handles all unicast use cases but the converse is not true.)

More general solutions for this communications paradigm are possible by moving the view of the problem from message exchange to the concept of coordinating shared objectives. In the distributed systems literature, communication associated with coordinating shared objectives has long been modeled as *distributed set reconciliation* [WegmanC81][Demers87]. In this approach, each domain of discourse is a named set, e.g., *myhouse.iot*. Each event or
action, e.g., a switch button press, is added as a new element to the instance of myhouse.iot at its point of origin then the reconciliation process ensures that every instance of myhouse.iot has this element. In 2000, [MINSKY03] developed a broadcast-capable set reconciliation algorithm whose communication cost equaled the set instance differences (which is optimal) but its polynomial computational cost impeded adoption. In 2011, [DIFF] used Invertible Bloom Lookup Tables (IBLTs) [IBLT][MPSR] to create a simple distributed set reconciliation algorithm providing optimal in both communication and computational cost. DeftT uses this algorithm (see Section 2.2) and takes advantage of IPv6's self-configuring link local multicast to avoid all manual configuration and external dependencies. This restores the system design to Figure 3 where each device has a single, auto-configured transport that makes use of the broadcast radio medium without need for a broker or multiple transport associations. Each button push is broadcast exactly once to be added to the distributed set. (See Section 2.5 to see how members fill in missing elements.)

1.3. Securing information

Conventional session-based transports combine multiple publications with independent topics and purposes under a single session key, providing privacy by encrypting the sessions between endpoints. The credentials of endpoints (e.g., a website) are usually attested by a third party certificate authority (CA) and bound to a DNS name; each secure transport association requires the exchange of these credentials which allows for secure exchange of a nonce symmetric key. In Figure 4 each transport session is a separate security association where each device needs to validate the broker's credential and the broker has to validate each device's. This ensures that transport associations are between two enrolled devices (protecting against outsider and some MITM attacks) but, once the transport session has been established there are no constraints whatsoever on what devices can say. Clearly, this does not protect against the insider attacks that currently plague OT, e.g., [CHPT] description of a lightbulb taking over a network. For example, the basic function of a light switch requires that it be allowed to tell a light to turn on or off but it almost certainly shouldn't be allowed to tell the light to overwrite its firmware (fwupd), even though "on/off" and "fwupd" are both standard capabilities of most smart light APIs. Once a TLS session is established, the transport handles "fwupd" publications the same way as "on/off" publications. Such attacks can be prevented using trust management that operates per-publication, using rules that enable the "fwupd" from the light switch to be rejected. Combining per-publication trust decisions with many-to-many communications over broadcast infrastructure requires per-publication signing rather than session-based signing.
Securing each publication rather than the path it arrives on deals with a wider spectrum of threats while avoiding the quadratic session state and traffic burden. In OT, valid messages conform to rigid standards on syntax and semantics [IEC61850][ISO9506MMS][ONE][MATR][OSCAL][NMUD][ST][ZCL] that can be combined with site-specific requirements on identities and capabilities to create a system's communication rules. These rules can be employed to secure publications in a trust management system such as [DLOG] where each publisher is responsible for supplying all of the "who/what/where/when" information needed for each subscriber to prove the publication complies with system policies.

Instead of vulnerable third-party CAs [W509], sites employ a local root of trust and locally created certificates. When the communication rules are expressed in a declarative language [DLOG], they can be validated for consistency and completeness then converted to a compact runtime form which can be authorized and secured via signing with the system trust anchor. This communication schema can be distributed as a certificate, then validated using on-device trusted enclaves [TPM][HSE][ATZ] as part of the device enrollment process. In DeftT's publication-based transport, the schema is used to both construct and validate publications, guaranteeing that all parts of the system always conform to and enforce the same rules, even as those rules evolve to meet new threats (more in Section 3.1). DeftT embeds the trust management mechanism described above directly in the publish and subscribe data paths as shown below:

Figure 5: Trust management elements of DeftT.

This approach extends LangSec's [LANGSEC] "be definite in what you accept" principle by using the authenticated common rules of the schema for belt-and-suspenders enforcement at both publication and subscription functions of the transport. If an application asks the Publication Builder to publish something and the schema shows it
lacks credentials, an error is thrown and nothing is published. Independently, the Publication Validator ignores publications that:

* don't have a locally validated, complete signing chain for the credential that signed it
* the schema shows its signing chain isn't appropriate for this publication
* have a publication signature that doesn't validate

Note that since an application's subscriptions determine which publications it wants, only certificates from chains that can sign publications matching the subscriptions need to be validated or retained. Thus a device's communication state burden and computation costs are a function of how many different things are allowed to talk to it but not how many things it talks to or the total number of devices in the system. In particular, event driven, publish-only devices like sensors spend no time or space on validation. Unlike most 'secure' systems, adding additional constraints to schemas to reduce attack surface results in devices doing less work.

1.4. Defined-trust Communications Domains

A Defined-trust Communications Limited Domain (or simply, trust domain) is a Limited Domain where all the members communicate via a DeftT (Figure 6) and are configured with the same trust anchor and schema as well as an individual schema-conformant DeftT identity cert chain that terminates at the trust anchor and the private key corresponding to the identity chain's leaf cert. The particular rules for any deployment are application-specific (e.g., Is it home IoT or a nuclear power plant?) and site-specific (specific form of credential and idiosyncrasies in rules) which DeftT accommodates by being invoked with a ruleset (schema) particular to a deployment. We anticipate that the efforts to create common data models (e.g., [ONE]) for specific sectors will lead to easier and more forms-based configuration of DeftT deployments.

A trust domain is perimeterless and may operate over one or more subnets, sharing physical media with non-member entities. Domain member entities' DeftTs publish and subscribe using Publication Builders and Validators as shown in Figure 5. Publications become the elements of a set, or named collection, that is synchronized across each subnet. DeftT uses a distributed set reconciliation protocol on each collection and each subnet independently. Every DeftT maintains at least two collections: pubs for application information Publications and cert where identity signing chains are published.
Trust domains are extended across physically separated subnets, subnets using different media and/or subdomains on the same subnet (see Section 2.7) by using relays that have a DeftT in each subnet and pass Publications between subnets as long as they are valid at the receiving DeftT Figure 7. Since set reconciliation does not accept duplicates, relays are powerful elements in creating efficient configuration-free meshes. The subnets of the figure could be different colocated media (e.g. bluetooth, wifi, ethernet) or may be physically distant. The triangle relay-only subnet can be carried over a unicast link. The set reconciliation protocol ensures that items only transit a subnet once: an item must be specifically requested in order to be transmitted. Any part of a verifiable defined-trust identity can be used in the delineation of subdomains, e.g. specific component(s) of identity names for all DeftTs of a subdomain can be constrained to be the same so that Publications are effectively only relayed to a particular "group" as identified by those components and this is enforced via the secured schema (non-"group" Publications will not validate). Relay discussion is in Section 2.7 and Section 5.

1.5. Current status

An open-source Defined-trust Communications Toolkit [DCT] with an example implementation of DeftT is maintained by the corresponding author's company. [DCT] currently has examples of using DeftT to implement secure brokerless message-based pub/sub using multicast.
UDP/IPv6 and unicast UDP/TCP and include extending a trust domain via a unicast connection or between two broadcast network segments.

Massive build out of the renewable energy sector is driving connectivity needs for both monitoring and control. Author King's company, Operant, is currently developing extensions of DeftT in a mix of open-source and proprietary software tailored for commercial deployment in support of distributed energy resources (DER). Current small scale use cases have performed well and expanded usage is underway. Pollere is also working on home IoT uses. The development philosophy for DeftT is to start from solving useful problems with a well-defined scope and extend from there. As the needs of our use cases expand, the Defined-trust communications framework will evolve with increased efficiencies. DeftT's code is open source, as befits any communications protocol, but even more critical for one attempting to offer security. DCT itself makes use of the open source cryptographic library libsodium [SOD] and the project is open to feedback on potential security issues as well as hearing from potential collaborators.

The well-known issues with 802.11 multicast [RFC9119] can make DeftT less efficient than it should be. Target OT deployments primarily use smaller packet sizes and DeftT's set reconciliation provides robust delivery that currently mitigates these concerns. DeftT use may become another force for improved multicast on 802.11, joining the critical network infrastructure applications of neighbor discovery, address resolution, DHCP, etc.

Cryptographic signing takes most of the application-to-network time in DeftT. Though not prohibitively costly (e.g., under 20 microseconds on a Mac Studio), increased use of signing in transports may incentivize creation of more efficient signing algorithms.

2. DeftT and Defined-trust Communications

DeftT synchronizes and secures communications between enrolled members of a Limited Domain [RFC8799]. DeftT's multi-party synchronized collections of named, schema-conformant Publications contrast with the bilateral session of TCP or QUIC where a source and a destination coordinate with one another to transport undifferentiated streams of information. DeftTs in a trust domain may hold different subsets of the collection at any time (e.g., immediately after entities add elements to the collection) but the synchronization protocol ensures all converge to holding the complete set of elements within a few round-trip-times following the changes.
Applications use DeftT to add to and access from a collection of Publications. DeftT enforces "who can say what to which" as well as providing required integrity, authenticity and confidentiality. Transparently to applications, a DeftT both constructs and validates all Publications against its schema's formal, validated rules. The compiled binary communications schema is distributed as a trust-root-signed certificate and that certificate's thumbprint (see Section 2.3.1.4 and Section 7) uniquely identifies each trust domain. Each DeftT is configured with the trust anchor used in the domain, the schema cert, and its own credentials for membership in the domain. To communicate, DeftTs must be in the same domain. Identity credentials comprise a unique private identity key along with a public certificate chain rooted at the domain's trust anchor. Certificates in identity chains are specified in the schema and contain the attributes granted to the identity. Thus, attributes are stored in the identity not on an external server.

As illustrated in Figure 2, each member publishes its credentials to the certificate collection in order to join the domain. DeftT validates credentials as a certificate chain against the schema and does not accept Publications without a fully validated signer. This unique approach enables fully distributed policy enforcement without a secured-perimeter physical network and/or extensive per-device configuration. DeftT can share an IP network with non-DeftT traffic as well as DeftT traffic of a different Domain. Privacy via AEAD (Authenticated Encryption with Associated Data) is automatically handled within DeftT if selected in the schema.

Figure 8: DeftT's interaction in a network stack

Figure 8 shows the data flow in and out of a DeftT. DeftT uses its schema to package application information into Publications that are added to its local view of the collection. Application information is packaged in Publications which are carried in collection addition (cAdd) PDUs that are used along with collection state (cState) PDUs to communicate about and synchronize Collections. cStates report the state of the local collection; cAdds carry Publications to other members that need them. These PDUs are broadcast on their subnet (e.g., UDP multicast).
2.1. Inside DeftT

DeftT's example implementation [DCT] is organized in functional library modules that interact to prepare application-level information for transport and to extract application-level information from packets, see Figure 9. Extensions and alternate module implementations are possible but the functionality and interfaces must be preserved. Internals of DeftT are completely transparent to an application and the example implementation is efficient in both lines of code and performance. The schema determines which modules are used. A DeftT participates in two required collections and may participate in others if required by the schema-designated signature managers. One of the required collections, pubs, contains application Publications. The other required collection, cert, contains the certificates of the trust domain. Specific signature managers may require group key distribution in descriptively named collection keys.

A shim serves as the translator between application semantics and the named information objects (Publications) whose format is defined by the schema. The syncps module is the set reconciliation protocol used by DeftT (see Section 2.2). New signature managers, distributors, and face modules may be added to the library to extend features. More detail on each module can be found at [DCT] in both code files and documents.

The signing and validation modules (signature managers) are used for both Publications and cAdds. Following good security practice, DeftT's Publications are constructed and signed early in their creation, then are validated (or discarded) early in the reception process. The schemaLib module provides certificate store access throughout DeftT along with access to distributors of group keys, Publication building and structural validation, and other functions of the trust management engine. This organization of interacting modules is not possible in a strictly layered implementation.
2.2. syncps: a set reconciliation protocol

DeftT requires a method or protocol that keeps collections of Publications synchronized. Required functionality for such a protocol can be understood through the example of the syncps protocol included in the example implementation. The syncps protocol uses IBLTs \[DIFF\][IBLT][MPSR] to solve the multi-party set-difference problem efficiently without the use of prior context and with communication proportional to the size of the difference between the sets being compared. The state of a local collection is encoded in an IBLT. A syncps announces its local collection state (set of currently known Publications) by sending a cState (Section 2.3.1.1) that also serves as a query for additional data not reflected in its local state. Receipt of a cState performs three simultaneous functions: (1) announces new Publications, (2) notifies of Publications that member(s) are missing and (3) acknowledges Publication receipt. The first may prompt the recipient to share its cState to get the new Publication(s). The second results in the recipient sending a cAdd Section 2.3.1.2 containing all the locally available missing Publications that fit. The third is used optionally and may result in a progress notification sent to other local modules so anything waiting for delivery confirmation can proceed.

On broadcast media, syncps uses any cStates it hears to reduce (suppress) sending excess cStates and listens for cAdds that may add to its collection. This means that one-to-many Publications cause sending a single cState and a single cAdd independently of the number of members desiring the Publication (the theoretical minimum possible for reliable delivery). The digest size of a cState can be controlled by Publication lifetime, dynamically constructing the digest to maximize communication progress [Graphene][Graphene19] and, if necessary for a large network, dynamically adapting topic specificity.

A cAdd with new Publication(s) responds to a particular cState as per (Section 2.3.1.2 item 1). Any DeftT that is missing a Publication (due to being out-of-range, asleep, channel errors, etc.) can receive it from any other DeftT. A syncps will continue to send cAdds as long as cStates are received that are missing any of its active Publications. This results in reliability that is subscriber-oriented, not publisher-oriented, kept efficient with protocol features that prevent multiple redundant broadcasts. The example implementation of syncps prevents redundant broadcasts by having originating publishers send their responding Publications immediately while others delay before supplying missing Publications, canceling if a responding cAdd is overheard. Other approaches are possible.
The collection synchronization work of a syncps module is shown as a state diagram in Figure 10. When a new syncps is started, it always sends its local cState (starts unsuppressed) on the network and sets an expiration timer for the cState. If this timer expires, the "new local cState" actions are repeated and the cState may be suppressed (thus not sent). For most collections, the initial cState will show an empty collection (certificate collections will have the local identity chain). The events that can move the collection forward are (1) the arrival of a cState from the network, (2) the arrival of a cAdd from the network whose csID matches a hash value stored from a previously received or sent cState, or (3) arrival of a new Publication from its shim. For an arriving cAdd, each Publication is extracted, validated, and passed to any registered subscriber callback(s). Non-validating packets are silently discarded (may optionally set alerts or count discards). Reception of new Pub(s) may cause an application process to create and add new Publications to the local collection. Sending of new Pubs is deferred until the entire cAdd has been processed. If there are no new Pubs to send, syncps moves to its "set sendCStateTimer" state where a cancelable sendCState timer is set to the estimated dispersion delay of this local subnet. (Dispersion should be << cState lifetime. More on dispersion delay in Section 2.5.)

Since new Publications are always eligible to send, if any were created while in "process cAdd", the next state is "process Pubs to send" with csID set to the csID field of the cAdd on entry to "process cAdd". In "process Pubs to send" any pending sendCState will be canceled and eligible Pubs are packaged as content for a new cAdd. Packaged Publications are subject to a hold time (during which they are ineligible to send) of twice the dispersion delay to avoid responding to cStates sent before reception of a cAdd containing the Pub. If there are Pubs to send, a cAdd with that content and the passed in csID is sent, then the set sendCStateTimer state is entered; when there are no Pubs to send, the action moves there directly.
Another path to exit the "wait for event" state is reception of a cState which moves to "process cState" where incoming cStates are recorded. If this cState matches the local cState, the syncps returns to the wait state. Otherwise, the IBLT is extracted from the cState, an IBLT is computed on the local Pub collection, and they are peeled to find the ones the received cState has that are not in the local collection ("needs") and the ones that are in the local collection and not in the received cState ("haves"). Syncps enters the "process Pubs to send" state with csID set to the hash of the cState's name. Eligible Pubs are "haves" that do not have a hold time set and locally generated Publications are sent preferentially. Publications obtained from others are not immediately eligible to send; members delay to give the originator time to respond, sending these when further cStates indicate a member continues to need them.

A syncps also exits the wait state when the attached shim has a Publication to send. Since a new Pub will not appear in any previously issued cState, any one can be used, including one issued locally. In "get stored cState", the best one (the most recent cState from the network if available) is retrieved and passed to the "peel IBLT values" state where its cState.csID is used for the csID value. There will always be at least the one new Publication to send in this case.

This state diagram is intended to capture the major functionality of a syncps module while excluding excessive detail. In particular, Figure 10 does not show "housekeeping" tasks on the collection, e.g., removal of expired Publications.

2.3. DeftT formats

All DeftT Information is represented using TLV (Type, Length, Value) tuples. Types can either be containers (they contain a concatenated sequence of TLVs) or leaves (they contain a single non-TLV value with well-defined semantics and serialization). All TLVs have a boolean 'valid()' method that returns 'true' if and only if their content satisfies all the constraints associated with the TLV's type. For container types this means, at minimum, that the sum of all the enclosed TLV Lengths and header sizes exactly equals the Length of the container and that the valid() method of each of the enclosed TLVs returns true. Most container types have additional constraints on the type, ordering and value of the enclosed TLVs that are described below.

2.3.1. Top level container TLVs

As shown in Figure 9 there are two kinds of top level containers: PDUs which are exchanged with the system-provided transport and carry Pubs, the other top level container, which are the elements of
the set synchronization protocol. PDUS and Pubs have similar structure and share most of their code but are designed to be unambiguously distinguishable. As indicated in Figure 8 and Figure 9, syncps uses a pub/sub model for both its shim facing and network facing interfaces. Thus the first TLV in any top level container is a Name container comprising the topic name used to mediate the pub/sub rendezvous. The other TLVs in the top level container depend on its kind.

There are two kinds of PDU containers: cState and cAdd and two kinds of of Publications, pubs and certs. All four are described in the following section.

2.3.1.1. cState PDUs

A cState PDU (TLV type 5) announces the items a member holds in a specific collection of a specific trust domain subnet. It must contain the following three TLVs and they must be in this order:

1. Name TLV containing exactly three type Generic (aka, byte array or binary blob) components:

   C1: Trust domain id consisting of the first 8 bytes of the SHA-256 thumbprint of the domain's schema cert.

   C2: Collection name

   C3: Run-length compressed IBLT of the items in the publisher's instance of the collection (see Section 2.2 for more information on IBLTs).

2. Nonce (TLV type 10, leaf) whose value must be 4 random bytes chosen by the publisher at the time the cState is built. Duplicate cStates can arise from multiple members announcing the same Name because they hold the same items or because the network doesn't handle multicast well and lets PDUs loop. The nonce allows these two cases to be distinguished so looping cStates can be dropped.

3. Lifetime (TLV type 12, leaf) whose value is the lifetime (measured in milliseconds since this PDU's arrival) serialized as an unsigned big-endian integer with all leading zero bytes suppressed. A member receiving the cState and capable of publishing into the collection can hold onto the cState for this lifetime. If the member has an item to publish before the end of the cState's lifetime, the Publication can be sent immediately in a responding cAdd.

For example, the initial PDU sent by the home IoT "gate controller" sample app (in examples/hmIot of [DCT]) looks like:
2.3.1.2. cAdd PDUs

Note: cAdds, Publications and Certificates all share the same Data (TLV type 6) container format but are distinguished by its Metainfo TLV. They all contain the same five TLVs in the same order but each has different constraints on the value of those TLVs.

A cAdd PDU (TLV type 6) supplies one or more Pubs in response to some cState. It must contain the following five TLVs and they must be in this order:

1. Name TLV derived from the cState's Name: the first two components, domain id and collection name, are the same but the IBLT component is replaced by a csID (TLV type 35, leaf) whose value must be the 32 bit big-endian Murmurhash of the cState's entire Name TLV. (This is done because "Repeat the question and append the answer" is the common strategy for matching responses to requests in multicast protocols but an IBLT can be hundreds of bytes which would drastically reduce the cAdd's payload space so "the question" is replaced with a compact hash proxy.)

2. Metainfo (TLV type 20) saying the PDU's ContentType is cAdd (42), i.e., contains one or more Pubs and nothing else so it must be 'structurally validated' on arrival.

3. Content container (TLV type 21) which must contain one or more complete, valid Pubs. The Pubs must NOT already be in the cState's IBLT. I.e., the Pubs must be newly created on the cAdd publisher or in the 'need' set when the difference between the publisher's IBLT and the cState's IBLT is 'peeled' (see [DIFF] and the DeftT example implementation's handleCState code for details).
4. SigInfo container (TLV type 22) which must contain a SigType (TLV 27, leaf) containing a valid keyed or unkeyed signature type from the types listed in Section 2.3.2. If and only if the signature type is keyed (i.e., validation requires the public key cert of a public/private keypair), the SigInfo must contain KeyLocator (TLV 28) containing a KeyDigest (TLV 29, leaf) of length 32 bytes containing the thumbprint of the cert needed for validation. The SigType must match the type of the PDU signature validator associated with the collection.

5. SigValue (TLV type 23, leaf) containing the result of signing the cAdd PDU with using the algorithm and key, if any, specified by the SigInfo. The Length of the TLV must match the length used by the signature type as per Section 2.3.2. The PDU signature validator must successfully validate the signature.

For example, what follows is the frontdoor's cAdd responding to the cState shown above. The collection being synced here is the certificate distributor which can't use any of the signature types that depend on keys since it's responsible for obtaining the keys that would be needed to validate a PDU's signature. Thus it is the only collection allowed to use an unkeyed [RFC7693] BLAKE2 MAC to integrity check the PDU. Since the content of the cAdd is self authenticating public key certs, this doesn't cause security issues.

6 (Data) size 561:
| 7 (Name) size 22:
| | 8 (Generic) size 8: 55d5 7f99 7d8d ba91
| | 8 (Generic) size 4:  cert
| | 35 (csID) size 4:  f6d7 3d84
| 20 (MetaInfo) size 3:
| | 24 (ContentType) size 1:  42 (CAdd)
| 21 (Content) size 489:
| ... (489 bytes of Content elided)
| 22 (SigInfo) size 3:
| | 27 (SigType) size 1:  9 (RFC7693)
| 23 (SigValue) size 32: af8e 1412 e659 103f 5237 f1e1 0e7b 0af8 9c..

Except for this collection, PDU and Pub signature types are specified in the schema. PDUs typically use AEAD with a locally elected cover key distributor to protect the content privacy. Pubs typically use EdDSA to provide provenance and ABAC attributes via the signing chain or a combined AEAD and EdDSA signature type (AEADSGN) to constrain content disclosure to some limited group. All encrypted content must remain encrypted, in motion or at rest, from point of origin to point(s) of use. The syncps subscribe upcall may decrypt a piece of content for ephemeral use but the callee must NOT retain the plaintext form.
2.3.1.3. Publications

As noted above, a Publication must be in a Data TLV containing the same five TLVs in the same order as cAdds and Certificates. Publications are distinguished by having a Metainfo ContentType of Blob (0).

A Publication (TLV type 6) must contain the following five TLVs and they must be in this order:

1. Name TLV which must contain at least three components and the first component's length must be non-zero. The schema specifies the format of the Name including number and type of components, allowed values, allowed signers, etc. Implementations must construct and sign Pubs so that they are consistent with the schema. (The example implementation's applications show that this can be done automatically with minimal application involvement, e.g., see the phone app in the office control example.) Implementations must fully validate Publications both cryptographically and against the schema before adding them to the collection. Implementations must NOT add a Publication to a collection that already contains it.

2. Metainfo (TLV type 20) saying the Publication's ContentType is Blob (0), i.e., contains arbitrary bytes that can't be 'structurally' validated (but are always cryptographically validated for integrity and authorization by the signature check).

3. Content container (TLV type 21) containing Length bytes. Length may be zero.

4. SigInfo container (TLV type 22) which must contain exactly two TLVs: a SigType (TLV type 27, leaf) containing a valid keyed signature type from the types listed in Section 2.3.2 followed by a KeyLocator (TLV type 28) containing a KeyDigest (TLV type 29, leaf) of length 32 bytes containing the thumbprint of the cert needed to validate the signature. The SigType in the Publication must match the collection's Publication validator which must match the #pubValidator specified in the schema.

5. SigValue (TLV type 23, leaf) containing the result of signing the Publication using the algorithm and key specified by the SigInfo. The Length of the TLV must match the length used by the signature type as per Section 2.3.2. The collection's publication signature validator must successfully validate the signature.

For example, what follows are two consecutive Publications made to the pubs collection. First, operator alice publishes a command for
all lock devices to lock themselves (similar to the multiple subscriptions per-light shown in Figure 3, the schema requires that all lockable devices subscribe to the iot1/lock/command/all prefix in pubs):

Three milliseconds later, the gate publishes that it has locked itself:
As described in Section 8, these Publications' schema is designed for spoofing and replay protection. Section 2.2 notes that the per-publication EdDSA signature prevents spoofing or modification. Since all collections ignore duplicates of an existing publication, replays of anything in the collection will be ignored. To keep collections from growing without bound, publications are removed after a collection-dependent lifetime but arriving are pubs are ignored as "expired" if their timestamp (name component 9) plus a collection-dependent "expiry time" is after the node's local time. "lifetime" is substantially larger then "expiry time" to account for clock skew so the combination of these two mechanisms prevents all replay.

2.3.1.4. Certificates

As noted above, a Certificate must be in a Data TLV containing the same five TLVs in the same order as cAdds and Publications. Certificates are distinguished by having a Metainfo ContentType of Key (2) and by having a Validity Period specified according to a more rigorous subset of the rules in [RFC1422] section 3.3.6 as described in item 5 below.
A Certificate (TLV type 6) must contain the following five TLVs and they must be in this order:

1. Name TLV which must contain at least five components and the first component's length must be non-zero. The schema specifies the format of the Name including number and type of components, allowed values, allowed signers, etc. Implementations must construct and sign certs so that they are consistent with the schema. (Tools to do this are supplied with the example implementation.) Implementations must fully validate certs both cryptographically and against the schema before adding accepting them. "Fully validating" requires that the cert's signer has been accepted thus a cert cannot be accepted until its entire signing chain has been accepted.

2. Metainfo (TLV type 20) saying the Cert's ContentType is Key (2), This means the container has no TLV structure to validate.

3. Content container (TLV type 21) containing Length bytes. Length must equal the size of the public key associated with the cert’s SigInfo SigType

4. SigInfo container (TLV type 22) which must contain exactly two TLVs: a SigType (TLV type 27, leaf) containing a valid keyed signature type from the types listed in Section 2.3.2 followed by a KeyLocator (TLV type 28) containing a KeyDigest (TLV type 29, leaf) of length 32 bytes containing the thumbprint of the cert needed to validate the signature. The KeyDigest must be followed by a Validity Period (TLV 253) containing a NotBefore (TLV 254, leaf) containing a valid 15 character ISO 8601-1:2019 format GMT timepoint followed by a NotAfter (TLV 255, leaf) containing a valid 15 character ISO 8601-1:2019 format GMT timepoint. The cert must be ignored if the NotBefore value is >= the NotAfter value, if the NotAfter value is < the current time or if the validity period is not completely contained within its signing cert's validity period. The SigType in the Cert must match the #certValidator type specified in the schema.

5. SigValue (TLV type 23, leaf) containing the result of signing the Cert using the algorithm and key specified by the SigInfo. The Length of the TLV must match the length used by the signature type as per Section 2.3.2.

For example, what follows is the frontdoor's identity cert used in the home IoT example:
2.3.2. Leaf TLVs

Most of DeftT's leaf TLVs were described above but there are two important enumeration types, name components and signature types, with particular constraints and implications.

There are four types of components allowed in a Name (TLV 7):

<table>
<thead>
<tr>
<th>Type</th>
<th>TLV</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>8</td>
<td>Arbitrary blob of bytes</td>
</tr>
<tr>
<td>csID</td>
<td>35</td>
<td>32-bit murmurhash of cState name (number)</td>
</tr>
<tr>
<td>Timestamp</td>
<td>36</td>
<td>GMT time point in microseconds (number)</td>
</tr>
<tr>
<td>SequenceNum</td>
<td>37</td>
<td>unsigned 64-bit integer (number)</td>
</tr>
</tbody>
</table>

Table 1: Name Component Types

"Number" types are encoded in big-endian order (MSB first) with all leading zero bytes suppressed. Thus their length can be zero to eight bytes. For example, a SequenceNum of 0 would be [37, 0], 100 would be [37, 1, 100] and 1,000,000 would be [37, 3, 15, 66, 64].

There are five types of signature allowed in a SigType (TLV 27) and each requires the SigValue (TLV 23) in a Data with that SigType have a particular size:
### Table 2: Signature Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
<th>SigValue length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>stSHA256</td>
<td>0</td>
<td>32</td>
<td>SHA256 data integrity</td>
</tr>
<tr>
<td>stAEAD</td>
<td>7</td>
<td>40</td>
<td>[RFC8103] content privacy plus full data integrity</td>
</tr>
<tr>
<td>stEdDSA</td>
<td>8</td>
<td>64</td>
<td>Ed25519 provenance and full data integrity</td>
</tr>
<tr>
<td>strRFC7693</td>
<td>9</td>
<td>64</td>
<td>[RFC7693] full data integrity</td>
</tr>
<tr>
<td>stAEADSGN</td>
<td>13</td>
<td>104</td>
<td>[RFC8103] content privacy with Ed25519 provenance and data integrity</td>
</tr>
</tbody>
</table>

#### 2.3.3. TLV header details

All TLV headers use the same format. They occupy either 2 or 4 bytes, depending on the value of $L$. $L$ specifies the length in bytes of $V$. Lengths in the range 0 to 252 occupy one byte. A length of zero is allowed and indicates there are no $V$ bytes. Lengths in the range 253 to 65535 occupy three bytes: a 'flag byte' of 253 followed by the two bytes of the 16 bit length in big endian order. Lengths greater than 65535 (deliberately) can not be represented so a DeftT object can be no larger than 65535+4 = 65539 bytes. (Objects of arbitrary size can be handled by a segmentation/reassembly layer above DeftT such as dct/shims/mbps.hpp in the example implementation.)

$L$ must use the minimum description length coding. For example, a length of 0 must be encoded as the single byte [0], not as the 3 bytes [253, 0, 0], 252 is encoded as [252], 253 as [253, 0, 253], 256 as [253, 1, 0] and 65535 as [253, 255, 255].

$T$ specifies the type of data in the container. It occupies one byte, must be an element of the valid types set defined below, and must conform to that element's rules.

#### 2.3.4. Design rationale

DeftT's Publication, PDU and serialization formats were strongly influenced by the [LANGSEC] observation that most security issues are due to improper input handling. For example, Part II of [LangSecErr] found that this class of errors accounted for 75% of the 47 OpenSSL security vulnerabilities reported in the 18 months following 2015-1-1. Also, as of 2023-7-5, all 25 of the protobuf CVEs listed in the NIST National Vulnerability Database are of this class.

[LangSecErr] suggests these vulnerabilities could have been avoided by designing the protocol following three rules:
The acceptable input to a program should be:

1. well-defined (i.e., via a grammar)
2. as simple as possible (on the Chomsky scale of syntactic complexity)
3. fully validated before use (no "shotgun parsing")

2.3.4.1. Making DeftT 'well-defined'

A DeftT domain's "acceptable inputs" are specified using its communication rules declarative language (see Section 3.1) then compiled by an LALR parser into a compact binary "schema" that avoids any need for runtime parsing -- given the schema, the DeftT runtime can construct or validate any legal domain input in constant time. The compiler will fail to construct a schema if the domain communication rules are incomplete or inconsistent.

After successful compilation, the schema is authorized, authenticated and integrity protected by cryptographic signing using the domain's trust anchor. This signed schema is supplied to every member as part of their identity bundle (Section 4.2) and the SHA-256 thumbprint (see Section 2.3.1.4) of the schema is the first component of every PDU's topic name. This ensures not only that the rules are well defined but also that all publishers and subscribers are playing by the same rules.

2.3.4.2. Making DeftT 'as simple as possible'

All DeftT Information is represented using TLV (Type, Length, Value) tuples for the reasons noted by Dan Berstein [netstrings][tnetstrings]:

*Unlike delimiter-based approaches like XML or JSON, TLVs are resistant to buffer overflow and false pairing attacks.
*TLVs are self-describing and trivial to parse or validate.
*They can be used recursively -- containers can contain other containers.
*TLVs are fast, cache friendly and not resource intensive.
*TLVs make no assumptions about contents and can store binary data without escaping or encoding.
*TLVs are transport agnostic.

Attackers regard the 'seams' between protocol layers as prime attack surface since a lower layer can pass up partial information that it later finds to be inconsistent or invalid (an anti-pattern known as shotgun parsing [LangSecErr]). DeftT deliberately reuses a small set of formatting conventions to construct its TLV containers in contrast to the Internet convention of constructing its PDUs in separate layers with rules chosen by different committees. For example, DeftT PDUs, Publications and Certs have essentially the
same format so they can all be structurally validated (e.g., the contents of a container are the type expected in the order expected and exactly fill their container) by one simple, generic, recursive descent validation pass over each arriving PDU performed at the point where it arrives.

As described in Section 1.3, DeftT validates every Publication and PDU both cryptographically and syntactically using the domain's communications rules to enforce who-can-say-what-to-which-where-when. DeftT does both serialization and validation using rules bound at runtime (Figure 5) not compile time. It can do this at rates competitive with protobufs by taking advantage of the "definiteness" of local-domain communication:

*Since the same rules are used both to produce and validate Publications/PDUs, encoding order is fixed and known in advance. Thus every top-level object can be validated by a single sequential pass through it.

*Every party to the communication is guaranteed to be using the same rules so there are no options and no negotiation thus no combinatorial explosion of variants to check.

*Communication rules can be extended and amended at any time and the resultant binary schema published to members with no changes to their code. Thus the current ruleset should always be the minimum necessary to support existing applications and policies, not the open-ended monster needed to support any possible future.

2.4. Application and network interface

Figure 8 and Figure 9 show the blocks and modules application information passes through in DeftT. Refer to those figures for this discussion of how application information originates at a trust domain member and progressing to a Publication in a collection that is sent in a PDU via the system network layer to be received by other members of the domain. (For more detail, see the library at [DCT].) DeftT uses a shim to interface with the application's model of information exchange. The only currently available shims in the example implementation [DCT] provides a message-based publish/subscribe (mbps) model to the application, although it should be possible to construct a shim that provides a different model (e.g., streaming). All the necessary DeftT startup is kicked off when an mbps object is instantiated by the application. After startup, the pub syncps of each member will maintain a cState containing the IBLT of its view of the collection. (In the stable, synchronized state, all members of a collection will have the same IBLTs.)
Applications use an mbps *subscribe* method to either subscribe to all messages or to a subset by topic, passing a callback function to handle matching items. These application-level subscriptions are turned into syncps subscriptions via mbps. When the application has new information to communicate, topic items (as parameters) and message are passed to mbps with a *publish* call. Only these topic components and the message, if any, are passed between the application and mbps. The message may be segmented into multiple Publications by mbps, if the message size exceeds Publication content. For each Publication, mbps-specific components are added to the parameter list and the services of *schemaLib* are invoked in order to build and publish a valid Publication according to the schema (no Publication will be built if the correct attributes are not contained in the member's identity chain). The Publication is signed using the *sign* method of the appropriate *sigmgr* and passed to syncps.

**syncps** adds this Publication to its collection and updates its IBLT to contain the new Publication. Since its application just created it, syncps knows this is a new addition to the collection and it is a response to the current cState. Thus the Publication is packaged into a cAdd and signed using the *sign* method of the designated *sigmgr* and passed to the face. The updated IBLT is packaged into a new cState that is handed to the face.

Trust domain members only process cAdds that share their trust domain identifier (Section 2.3.1.1 and Section 2.3.1.2). When a new cAdd is received at a member, the face ensures it matches an outstanding cState and, if so, passes it on to matching syncps(es). Syncps validates (both structurally and cryptographically) the cAdd using the appropriate sigmgr's *validate* and continues, removing Publications, if valid. Each Publication is structurally validated via a sigmgr and valid Publications are added to the local collection and IBLT. syncps passes this updated cState to the local face. If this Publication matches a subscription it is passed to mbps, invoking the sigmgr's decrypt if the Publication is encrypted (Publication decryption is *not* available at Relays.) mbps receives the Publication and passes any topic components of interest to the application along with the content (if any) to the application via the callback registered when it subscribed. (If the original content was spread across Publications, mbps will wait until all of the content is received. The sCnt component of a mbps Publication Name is used for this.)

2.5. Synchronizing a collection

This section has thus far covered implementation of DeftT at a member and the format of its communications. Although DeftT works on unicast (as a special case of multicast) links, it is designed to
take full advantage of a multicast subnet (e.g., link-level IPv6 multicast on broadcast media). This subsection is an introduction to how syncps orchestrates its collection-based communications on a shared channel. A sequence diagram of the interaction of multiple members' syncps modules interacting on a multicast subnet to keep their collections synchronized is shown in Figure 11. Starting with all members connected to the collection (having confirmed publication of their identity credentials) and with an empty pubs collection (i.e., no members have active Publications), member2's application passes content to its DeftT (via an mbps.publish()), which creates and sends a cAdd PDU. The cAdd uses a hash of the shared (empty) cState as its cState identifier (third component of the Name Section 2.3.1.2 item 1) to indicate the Publication(s) it carries are additions to the collection in that state. Member2's new local cState (with the new Publication) is scheduled to be sent at a computed delay of twice the subnet's dispersion time (d) plus a small random value (r). (Dispersion time is an estimate of the expected time for a cAdd to reach every member's collection. It may be a fixed or adaptive estimate and syncps is robust to inaccuracies: an overestimate may lead to longer delays and an underestimate may mean more cState traffic.) Members receive and validate the cAdd, then extract and validate Publication(s), passing it to subscriptions. To avoid excessive cState traffic, each member schedules the acknowledging cState for d+r. (Scheduling a sendCState cancels any pending value.) When the sendCState timer expires, a new local cState is created with an IBLT that contains the new Publication. This cState's expiration time is scheduled (value significantly longer than d) and the member sends the cState unless it is suppressed. syncps suppresses cStates that are identical to one that has been heard twice. If member2 is waiting to confirm the Publication, it can do so with the first of these cStates it receives. In Figure 11, member6 did not receive the cAdd but reception of one of the new cStates shows it there is a new Publication in the collection so it immediately sends its own local cState (which has an empty collection, lacking member2's Publication). Here, all members receive that cState, but member2, as the originator, responds preferentially, sending a new cAdd immediately. All other members set a timer (to 2*d+r) to send a cAdd with the Publication. That timer is cancelled when an overheard cAdd responding to that cState contains the Publication. Meanwhile, member6 receives this new cAdd, adds the Publication to its collection and schedules a new cState for d+r. That cState should be suppressed as it will match those already sent by the other members. Now the distributed collection is synchronized with a state of one Publication (p1). If no other application content is created, cStates will be sent at ~cStateLifetime. On the wire, we will see one cState per ~cStateLifetime since they overlap enough to suppress others. When p1 expires, it will be removed at each local collection and the subsequent cState will show an empty collection.
Although Figure 11 shows one Publication at a time for clarity, the logic works if multiple members are publishing simultaneously or at close intervals (less than $d$ or the cStateLifeTime). The distributed collection is always moving toward synchronization but during periods of intense interaction, times when all members are synchronized may be infrequent; this is not considered problematic.

2.6. Distributors

Distributors implement services a Deft requires for its operation. Distributors optional to general operation are specified in the communications schema.

2.6.1. Certificate distributor

DeftT's certificate distributor is a required module. It implements a collection of all the signing chain certificates in the Domain. When a new DeftT is instantiated, it must publish all the certificates from its identity bundle as well as its locally created signing certificate. This joining process was shown in Figure 2. Since many certificates in a member's chain are shared, that will be reflected in each cState and those certs will not be sent on the subnet. A member DeftT must receive a cState showing its signing chain in another member's local collection before a DeftT can be considered "connected" to the trust domain. This ensures there is at least one other member that can receive the PDUs it sends.

2.6.2. Group key distributors

Group key distributors are optional in DeftT but required, and automatically supplied, if encryption is specified in the schema. When present, they are instantiated after their local certificate distributor has "connected." The example implementation contains two types of group key distributors. A group key distributor handles creation and distribution of a single symmetric key to all members of the Domain to use to encrypt either Publications or PDUs (if both are encrypted, there is a group key distributor for each). A
subscriber group key distributor distinguishes subscribers that can decrypt PDUs and/or Publication and publishers that encrypt PDUs and/or Publications (a member can be both subscriber and publisher). The group key distributor is briefly described here.

A trust domain using group key encryption must have at least one member with the attribute or capability of "keymaker" in its identity chain. Keymaker-capable members of a Domain elect a keymaker that makes a new symmetric encryption key upon winning the election. The non-keymakers publish key requests that the keymaker uses to create a list of current members. Requests and the symmetric key both have limited lifetimes. The keymaker uses each member's signing cert to encrypt a copy of the current key and creates and publishes as many Publications as needed to carry all the encrypted keys. In these Publication, entries are indexed by the thumbprint of associated signing cert and the range of thumbprints is used in the Publication name. Members only accept such Publications from keymaker-capable signers and, in case of conflict, use the key sent by a member whose signing cert thumbprint is the smallest.

If the keymaker receives a new key request in between making new keys, a copy of the key will be encrypted for it and published. There is no explicit revocation but a blacklist can be implemented and either published or passed from an application and a new group key can be made and distributed to non-blacklisted members ahead of the normal schedule.

2.6.3. Other distributors

Distributors may be used for other types of key distribution and for distributing other types of information, e.g. blacklisted members, domain statistics.

2.7. Schema-based information movement

Although the Internet's transport and routing protocols emphasize universal reachability with packet forwarding based on destination, a significant number of applications neither need nor desire to transit the Internet (e.g., see RFC8799). This is true for a wide class of OT application. Further, liberal acceptance of packets while depending on the good sending practices of others leaves critical applications open to misconfiguration and attacks. Internet protocols use header information to tell them how to forward packets. DeftT's header only contains a trust domain id and a collection name. Each DeftT has a trust management engine with a copy of rules for its domain. DeftT only creates and moves its Publications in accordance with that fully specified communications schema and never moves a PDU between subnets. This approach differs in both intent and execution from Internet forwarding and may not be
appropriate for all use cases but offers new opportunities to address the specific security requirements of many Limited Domain use cases.

DeftTs on the same subnet may be in different trust domains and DeftTs in the same trust domain may not be on the same subnet. In some cases, it is useful to define sub-domains whose DeftTs have a compatible, but more limited, version of the trust domain's communications schema (introduced in Section 1.3 and further discussed in Section 3). "Compatible" means there is at least one Publication type and associated signer specification in common or one schema may be a subset of the other. In the case of sub-domains, they may be deployed on the same subnet or on different subnets. (The rules of a sub-domain compiled to a binary schema distributed as a schema cert will have a different thumbprint from that of the full trust domain.) A possible use for multi-subnet trust domains with different sub-schemas is where a unicast link is used to connect two remotely located subnets of the same parent trust domain but only certain types of Publications should go through the unicast link and there may be slightly different rules used at each subnet. Different sub-schemas on the same subnet might be used where certain members have more limited access, either due to the technology of their devices or to limit their access (e.g., guests of a network). Relays could limit Publications by filtering Publications or subscribing to limited Publication subsets but delineating these limits in a schema provides enforcement of Publication movement.

In the case of DeftTs on the same subnet but in different trust domains or different sub-domains, the cState and cAdd PDUs of different domains are differentiated by the domain id (thumbprint of the domain's schema certificate as in Section 2.3.1.1 item C1) which can be used at the face module to determine whether or not to process a PDU. A particular sync collection is managed on a single subnet: cState and cAdds are not forwarded off that subnet nor between DeftTs with different domain ids on the same subnet. Instead, schema-compliant Relays connect Publications between separate sync collections of the same trust domain. Collections are differentiated by both subnet (the physical media) and domain id (a required field of the cState and cAdd PDUs). Consequently, cStates and cAdds are subnet-specific while Publications belong to a trust domain (or sub-domain).

A Relay is implemented [DCT] as an entity running on a device with a DeftT interface on each subnet (two or more) or with multiple DeftT interfaces to the same subnet Figure 12 where each uses a different but compatible version of the others' schema. Each DeftT participates in different sync collections and uses a communication identity valid for the schema used by the DeftT. Only Publications (including certs) are relayed between DeftTs and the Publication
must validate against the schema of each DeftT. Consequently cAdd encryption is unique per collection while Publication encryption holds across the domain.

As Relays do not originate Publications, their DeftT API module (a "shim", see Section 2.1) performs pass-through of valid Publications. The Relay of Figure 12-left is on three separate wireless subnets. If all three DeftTs are using an identical schema, a new validated cert added to the cert store of an incoming DeftT is then passed to the other two, which each validate the cert before adding to their own cert stores (superfluous in this case, but not a lot of overhead for additional security). When a valid Publication is received at one DeftT, it is passed to the other two DeftTs to validate against their schemas and published if it passes.

![Figure 12: Relays connect subnets](image)

A Relay may have different identities and schemas for each DeftT but must have the same trust anchor and schemas must be identical copies, proper subsets or overlapping subsets of the domain schema. Publications that are undefined for a particular DeftT will be silently discarded when they do not validate upon relay, just as they are when received from a face. This means the Relay application of Figure 12-left can remain the same but Publications will only be published to a different subnet if its DeftT has that specification in its schema. Relays may filter Publications at the application level or restrict subscriptions on some of their DeftT interfaces. Figure 12-right shows extending a trust domain geographically by using a unicast connection (e.g., over a cell line or tunnel over the Internet) between two Relays which also interface to local broadcast subnets. Everything on each local subnet shows up on the other. A communications schema subset could be used here to limit the types of Publications sent on the remote link, e.g., logs or alerts. Using this approach in Figure 12-right, local communications for subnet 1 can be kept local while subnet 2 might send commands and/or collect log files from subnet 1.

More generally, Relays can form a mesh of broadcast subnets with no additional configuration (i.e., Relays on a broadcast network do not need to be configured with others' identities and can join at any time). The mesh is efficient: Publications are only added to an individual DeftT's collection once regardless of how it is received. Relays with overlapping broadcast physical media will only add a Publication to any of its DeftTs once; syncps ensures there are no
duplicates. More on the applicability of DeftT meshes is in Section 5.

2.8. Congestion control

Each DeftT manages its collection on a single broadcast subnet (since unicast is a proper subset of multicast, a point-to-point connection is viewed as a trivial broadcast subnet) thus only has to deal with that subnet's congestion. As described in the previous section, a device connected to two or more subnets may create DeftTs having the same collection name on each subnet with a Publication Relay between them but DeftT never forwards PDUs between subnets. It is, of course, possible to run DeftT over an extended broadcast network like a PIM multicast group but the result will generally require more configuration and be less reliable, efficient and secure than DeftT's self-configuring peer-to-peer Relay mesh.

DeftT sends at most one copy of any Publication over any fully connected subnet, independent of the number of publishers and subscribers on the subnet. Thus the total DeftT traffic on a subnet is strictly upper bounded by the application-level publication rate. As described in Section 2.2, DeftTs publish a cState specifying the set elements they currently hold. If a DeftT receives a cState specifying the same elements (Publications) it holds, it doesn't send its cState. Thus the upper bound on cState publication rate is the number of members on the subnet divided by the cState lifetime (typically seconds to minutes) but is typically one per cState lifetime due to the duplicate suppression. Each member can send at most one cAdd in response to a cState. This creates a strict request/response flow balance which upper bounds the cAdd traffic rate to (number of members - 1) times the cState publication rate. The flow balance ensures an instance can't send a new cState until it's previous one is either obsoleted by a cAdd or times out. Similarly a cAdd can only be sent in response to the cState which it obsoletes. Thus the number of outstanding PDUs per instance is at most one and DeftT cannot cause subnet congestion collapse.

If a Relay is used to extend a trust domain over a path whose bandwidth delay product is many times larger than typical subnet MTUs (1.5-9KB), the one-outstanding-PDU per member constraint can result in poor performance (1500 bytes per 100ms transcontinental RTT is only 120Kbps). DeftT can run over any lower layer transport and stream-oriented transports like TCP or QUIC allow for a 'virtual MTU' that can be set large enough for DeftT to relay at or above the average publication rate (the default is 64KB which can relay up to 5Mbps of Publications into a 100ms RTT). In this case there can be many lower layer packets in flight for each DeftT cAdd PDU but their congestion control is handled by TCP or QUIC.
3. Defined-trust management engine

OT applications are distinguished (from general digital communications) by well-defined roles, behaviors and relationships that constrain the information to be communicated (e.g., as noted in [RFC8520]). The capabilities and attributes of Things can be characterized in structured abstract profiles that can be machine-readable (e.g., [ONE][RFC8520][ZCL]). Energy applications in particular have defined strict attribute- and role-based access controls [IEC] though proposed enforcement approaches require interaction of a number of mechanisms across the communications stack [NERC]. In Defined-trust Communications, structured profiles and rules strictly define permitted behaviors including what types of messages can be issued or acted on; undefined behaviors are not permitted. These rules, along with local configuration, are incorporated directly into the schemas used by DeftT's integrated trust management engine both to prohibit undefined behaviors and to construct compliant Publications. This not only provides a fine-grained security but a highly usable security, an approach that can make an application writer's job easier since applications do not need to contain local configuration and security considerations.

DCT [DCT] includes a language for expressing the rules of communication, its compiler, and other tools to create the credentials a DeftT needs at run-time. DCT is example code that is provided as open source. It is frequently updated to improve features and performance but may have bugs or unoptimized features in any particular release.

3.1. Communications schemas

Defined-trust's use of communications schemas has been influenced by [SNC][SDSI] and the field of trust management defined by Blaze et. al. [DTM] as the study of security policies, security credentials, and trust relationships. Li et. al. [DLOG] refined some trust management concepts arguing that the expressive language for the rules should be declarative (as opposed to the original work). Communications schemas also have roots in the trust schemas for Named-Data Networking, described in [STNDN] as "an overall trust model of an application, i.e., what is (are) legitimate key(s) for each data packet that the application produces or consumes." [STNDN] gave a general description of how trust schema rules might be used by an authenticating interpreter finite state machine to validate packets. A new approach to both a trust schema language and its integration with communications was introduced in [NDNW] and extended in [DNMP][IOTK][DCT]. In this approach, a schema is
analogous to the plans for constructing a building. Construction plans serve multiple purposes:

1. Allow permitting authorities to check that the design meets applicable codes
2. Show construction workers what to build
3. Let building inspectors validate that as-permitted matches as-built

Construction plans get this flexibility from being declarative: they describe "what", not "how". As noted on paragraph 4 of [DLOG]:

"a declarative trust management specification based on a formal foundation guarantees all parties to a communication have the same notion of what constitutes compliance."

This is a critical part of Defined-trust Communications which uses the more descriptive term communication schema (or schema where its use is clearly with respect to defined-trust communications) for the rules that define the communications of a trust domain. A single schema, securely provided to all members, provides the same protection as dozens of manually configured, per-node ACL rules.

VerSec, an approach to creating schemas, is included with the Defined-trust Communications Toolkit [DCT]. VerSec includes a declarative schema specification language with a compiler that checks the formal soundness of a specification (case 1 above) then converts it to a signed, compact, binary form. The diagnostic output of the compiler (including a digraph listing) can be used to inspect that the intent for the communications schema has indeed been implemented. The binary form is used by DeftT to build (case 2) or validate (case 3) the Publications (format covered in Section 2.3.1.3). Certificates (Section 2.3.1.4) are a type of Publication, allowing them to be distributed and validated using DeftT, but they are subject to many additional constraints that ensure DeftT's security framework is well-founded.

3.2. A schema language

The VerSec language follows LangSec [LANGSEC] principles to minimize misconfiguration and attack surface. Its structure is amenable to a forms-based input or a translator from the structured data profiles often used by standards [ONE][RFC8520][ZCL]. Declarative languages are expressive and strongly typed, so they can express the constructs of these standards in their rules. VerSec continues to evolve and add new features as its application domain is expanded; the latest released version is at [DCT]. Other languages and compilers are possible as long as they supply the features and output needed for DeftT.
A communication schema expresses the intent for a domain's communications in fine-grained rules: "who can say what." Credentials that define "who" are specified along with complete definitions of "what." Defined-trust communications has been targeted at OT networking where administrative control is explicit and it is not unreasonable to assume that identities and communication rules can be securely configured at every device. The schema details the meaning and relationship of individual components of the filename-like names (URI syntax [RFC3986]) of Publications and certificates. A simple communications schema (Figure 13) defines a Publication in this domain as #pub with a six component name. The strings between the slashes are the tags used to reference each component in the structured format and in the run-time schema library. An example of this usage is the component constraint following the "&" where ts is a timestamp (64-bit unix timepoints in microseconds) which will be set with the current time when a Publication is created. The first component gets its value from the variable "domain" and #pubPrefix is designated as having this value so that the schema contains information on what part of the name is considered common prefix. For the sake of simplicity, the schema puts no constraints on other name components (not the usual case for OT applications) but requires that Publications of template #pub are signed by ("=") a mbrCert whose format and signing rule (signed by a netCert) is also defined. The Validator lines specify cryptographic signing and validation algorithms from DCT's run-time library for both the Publication and the cAdd PDU that carries Publications. Here, both use EdDSA signing. This schema has no constraints on the inner four name components (additional constraints could be imposed by the application but they won't be enforced by DeftT). Member identity comes from a mbrCert which allows it to create legal communications (using the associated private key in signing). A signing certificate must adhere to the schema; Publications or cAdds with unknown signers are discarded. The timestamp component is used to prevent replay attacks. A DeftT adds its identity certificate chain to the domain certificate collection (see Section 4.2) at its startup, thus announcing its identity to all other members. Using the pre-configured trust anchor and schema, any member can verify the identity of any other member. This approach means members are not pre-configured with identities of other members of a trust domain and new entities can join at any time.
To keep the communications schema both compact and secure, it is compiled into a binary format that becomes the content of a schema certificate. The \[DCT\] schemaCompile converts the text version (e.g., \[Figure 13\]) of the schema into binary as well as reporting diagnostics (see \[Figure 14\]) used to confirm the intent of the rules (and to flag problems).

Figure 13: An example communication schema in VerSec \[DCT\]

Even this simple schema provides useful security, using enrolled identities both to constrain communications actions (via its \#pub format) and to convey membership. To increase security, more detail can be added to \[Figure 13\]. For example, different types of members can be created, e.g., "admin" and "sensor", and communications privacy can be added by specifying AEAD Validator to encrypt cAdds or AEADSGN (signed AEAD) to encrypt Publications. To make those member types meaningful, a security policy could be employed by defining Publications such that only admins can issue commands and only sensors can issue status. Specifying the AEAD validator for the

binary schema is 301 bytes

Figure 14: schemaCompile diagnostic output for example of \[Figure 13\]
cAddValidator means that at least one member of a subnet will need a key maker attribute, which is conferred via a capability certificate in a member’s signing chain. Since DeftT identities include the member cert and its entire signing chain, adding attributes via capability certificates to a signing chain lets attribute-based security policies be implemented without the need for separate servers accessed at run-time (and the attendant security weaknesses). More on certs will be covered in Section 4.

If AEADSGN is specified for the pubValidator, at least one member of the trust domain will need key maker capability. In Figure 14 key maker capability is added to the signing chain of all sensors. With AEAD specified, a key maker is elected during DeftT start up and that key maker creates, publishes, and periodically updates the shared encryption key. (Late joining entities are able to discover that a key maker has already been chosen.) These are the only changes required in order to increase security and add privacy: neither code nor binary needs to change and DeftT handles all aspects of validators. The unique approach to integrating communication rules into the transport makes it easy to produce secure application code.

adminCert: mbrCert & { _mbrType: "admin" } <= netCert
sensorCert: mbrCert & { _mbrType: "sensor" } <= kmCap
capCert: _network/"CAP"/_capId/_capArg/_keyinfo <= netCert
kmCap: capCert & { _capId: "KM" }
#reportPub: #pub & {topic:"status"} <= sensorCert
#commandPub: #pub & {topic:"command"} <= adminCert
#cAddValidator: "EdDSA"

Figure 15: Enhancing security in the example schema

Converting desired behavioral structure into a schema is the major task of applying Defined-trust Communications to an application domain. Once completed, all the deployment information is contained in the schema. Although a particular schema cert defines a particular trust domain, the text version of a schema can be re-used for related applications. For example, a home IoT schema could be edited to be specific to a particular home network or a solar rooftop neighborhood and then signed with a chosen trust anchor.

4. Certificates and identity bundles

Defined-trust’s approach is partially based on the seminal SDSI [SDSI] approach to create user-friendly namespaces that establish transitive trust through a certificate (cert) chain that validates locally controlled and managed keys, rather than requiring a global Public Key Infrastructure (PKI). When certificates are created, they
have a particular context in which they should be utilized and trusted rather than conferring total authority. This is particularly useful in OT where communicating entities share an administrative control and using a third party to certify identity is both unnecessary and a potential security vulnerability. Well-formed certificates and identity deployment are critical elements of this framework. This section describes certificate requirements and the identity bundles that are securely distributed to trust domain members. (DCT includes utilities to create certs and bundles.)

4.1. Obviate CA usage

Use of third party certificate authorities (CAs) is often antithetical to OT security needs. Any use of a CA (remote or local) results in a single point of failure that greatly reduces system reliability. An architecture with a single, local, trust root cert (trust anchor) and no CAs simplifies trust management and avoids the well-known CA federation and delegation issues and other weaknesses of the X.509 architecture (summarized at [W509], original references include [RSK][NVR]). DCT certificates (see Section 2.3.1.4) can be generated and signed locally (using supplied utilities) so there is no reason to aggregate a plethora of unrelated claims into one cert (avoiding the Aggregation problem [W509]).

A DCT cert's one and only Subject Name is the Name of the Publication that contains the public key as its content and neither name nor content are allowed to contain any optional information or extensions. Certificates are created with a lifetime; local production means cert lifetimes can be just as long as necessary (as recommended in [RFC2693]) so there's no need for the code burden and increased attack surface associated with certificate revocation lists (CRLs) or use of on-line certificate status protocol (OSCP). Keys that require longer lifetimes, like device keys, get new certs before the current ones expire and may be distributed through DeftT (e.g., using a variant of the group key distributors in DCT). If there is a need to exclude previously authorized identities from a domain, there are a variety of options. The most expedient is via use of an AEAD cAdd or Publication validator by ensuring that the group key maker(s) of a domain exclude that entity from subsequent symmetric key distributions until its identity cert expires (and it is not issued an update). Another option is to publish an identity that supplants that of the excluded member. Though more complex, it is also possible to distribute a new schema and identities (without changing the trust anchor), e.g., using remote attestation via the TPM.

From Section 3, a member cert is granted attributes in the schema via the certs that appear in its member identity chain. Member certs are always accompanied by their full chain-of-trust, both when
installed and when the member publishes its identity to the cert
collection. Every signing chain in the domain has the same trust
anchor at its root and its legal form specified in the schema.
Without the entire chain, a signer's right to issue Publications
cannot be validated. Cert validation is according to the schema
which may specify attributes and capabilities for Publication
signing from any certificate in the chain. For this model to be well
founded, each cert's key locator must uniquely identify the cert
that actually signed it. This property ensures that each locator
resolves to one and only one signing chain. A cert's key locator is
a thumbprint, a SHA256 hash of the entire signer's Publication
(name, content, key locator, and signature), ensuring that each
locator resolves to one and only one cert and signing chain. Use of
the thumbprint locator ensures that certs are not open to the
substitution attacks of name-based locators like X.509's "Authority
Key Identifier" and "Issuer".

4.2. Identity bundles

Identity bundles comprise the certificates needed to participate in
a trust domain: trust anchor, schema, and the member's identity
chain. The private key corresponding to the leaf certificate of the
member's identity chain should be installed securely when a device
is first commissioned (e.g., out-of-band) for a network. The public
certs of the bundle may be placed in a file in a well-known location
or may, in addition, have their integrity attested or even be
encrypted. Secure device configuration and on-boarding should be
carried out using the best practices most applicable to a particular
deployment. The process of enrolling a device by provisioning an
initial secret and identity in the form of public-private key pair
and using this information to securely onboard a device to a network
has a long history. Current and emergent industry best practices
provide a range of approaches for both secure installation and
update of private keys. For example, the private key of the bundle
can be secured using the Trusted Platform Module, the best current
practice in IoT, or secure enclave or trusted execution environment (TEE)
where available. In that case, an authorized configurer adding
a new device can use TPM tools to secure the private signing key and
install the rest of the bundle file in a known location before
deploying the device in the network. Where entities have public-
private key pair identities of any (e.g., non-DCT) type, these can
be leveraged for DeftT identity installation. Figure 16 shows the
steps involved in configuring entities and their correspondence of
the steps to the "building plans" model. (The corresponding tools
available in DCT are shown across the bottom and the relationship to
the "building plans" model is shown across the top.)
In the examples at [DCT], an identity bundle is given directly to an application via the command line, useful for development, and the application passes callbacks to utility functions that supply the certs and a signing pair separately. For deployment, good key hygiene using best current practices must be followed e.g., [COMIS]. In deployment, a small application manager may be programmed for two specific purposes. First, it is registered with a supervisor [SPRV] (or similar process control) for its own (re)start to serve as a bootstrap for the application. Second, it can have access to the TPM functions and the ability to create "short-lived" (~hours to several days) public/private key pair(s) that are signed by the installed (commissioned) private identity key using the TPM. This Publication signing key pair is created at (re)start and recreated at the periodicity of the signing cert lifetime. Since the signing of the public cert happens via requests to the TPM, the identity key (used to sign the cert) cannot be exfiltrated. The locally created signing key is used in the communications path where TPM signing overhead is prohibitive.

The DCT examples and library use configured member identities to sign locally created signing certs (with associated private keys) so the example schemas give the format for these signing cert names. A DefT will request a new signing cert shortly before expiration of the one in use. Upon each signing cert update, only the new cert needs to be published via DefT's cert distributor. Figure 17 outlines a representative procedure.

![Figure 16: Creating and configuring identity bundles](image1)

**Figure 16:** Creating and configuring identity bundles

All DCT certs have a validity period. Publication signing key pairs (with public signing certs) are generated locally so they can easily be refreshed as needed. Trust anchors, schemas, and the member identity chain are higher value and often require generation under hermetic conditions by some authority central to the organization. Their lifetime should be application- and deployment-specific, but
the higher difficulty of cert production and distribution often necessitates lifetimes of weeks to years.

Updating schemas and other certificates over the deployed network (OTA) is application-domain specific and can either make use of domain best practices or develop custom DeftT-based distribution. Changing the trust anchor is considered a re-commissioning. The example here is merely illustrative; with pre-established secure identities and well-founded approaches to secure on-line communications, a trust domain could be created OTA using secure identities established through some other system of identity.

5. Use Cases

5.1. Secure Industrial IoT

IIoT sensors offer significant advantages in industrial process control including improved accuracy, process optimization, predictive maintenance and analysis, higher efficiency, low-cost remote accessibility and monitoring, reduced downtime, power savings, and reduced costs [IIIOT]. Critical Digital Assets (CDA) are a class of industrial assets such as power plants or chemical factories which must be carefully controlled to avoid loss-of-life accidents and where IIoT sensors require tight security. Even when IIoT sensors are not used for direct control of CDA, spoofed sensor readings can lead to destructive behavior. There are real-life examples (such as uranium centrifuges) of nation-state actors changing sensor readings through cyberattacks leading to equipment damage. These risks result in a requirement for stringent security reviews and regulation of CDA sensor networks. Despite the advantages of deploying CDA sensors, adequate security is prerequisite to deploying the CDA sensors. Information conveyed via DeftT has an ensured provenance and may be efficiently encrypted making it ideal for this use.

IIoT sensors may be fixed or mobile (including drone-based); mobility and environmental factors may cause different sensor gateways to receive measurements from a particular sensor over time. A DeftT mesh captures Publications anywhere within its combined network coverage area and ensures it efficiently reaches all members as long as they are in range of at least one member that has received the information. An out-of-service or out-of-range member can receive all active subscribed Publications once it is in range and/or able to communicate. DeftT forms meshes with no additional configuration (beyond DeftT's usual identity bundle and private key) needed to make devices recognize one another in the trust domain. To see how DeftT propagates information throughout a partially connected mesh, consider Figure 18 where sensor S1's signal can
reach devices D1-D4 but not D5 and D6. (Refer to Section 2.2 and Section 2.5.)

Figure 18: Members out-of-range of a Publication's originator can receive from non-originating members

1. S1 sends a cAdd with its latest measurement Publication that is received by D1-D4 and added to their collections after which they synchronize their cStates.

2. Either a device in range of D5 and/or D6 sends a cState that shows the new Publication which results in a cState from D5 and/or D6 that serves as a request for that Publication or D5 and/or D6 send a periodic cState update that lacks the new Publication and is received by at least one of D1-D4.

3. When devices that have received the Publication hear those lacking cStates, they wait a dispersion delay (plus a small random value) so that the originator or some other device might respond, after which they send the Publication in a cAdd unless a cAdd responding to the specific lacking cState is overheard.

The large physical scale of many industrial processes necessitates that expensive cabling costs be avoided through wireless transport and battery power. Wireless sensor deployments in an industrial environment can suffer from signal outages due to shielding walls and interference caused by rotating machinery and electrical generators. In particular, nuclear power plant applications have radioactive shielding walls of very thick concrete and security regulations make any plant modifications to add cabling subject to expensive and time-consuming reviews and permitting. Consider such an industrial setting where LoRa sensors collect a wide range of information (e.g., temperature, movement/vibration, light levels, etc.) they broadcast in layer 2 LoRaWAN messages (see Figure 19). A WiFi network includes fixed displays, mobile tablets, and devices with both a LoRaWAN gateway interface and a WiFi interface (Gateways). Any particular WiFi-enabled device may subscribe to a subset of the information available through DeftT. Privacy of data is ensured by encrypting cAdd PDUs. The presence of several Gateways within a single sensor's broadcast range reduces the number of lost sensor packets and the DeftT WiFi mesh is resilient against transmission outages, further facilitating reliability. Publications
are sent once and heard by all in-range members while Publications missing from one DeftT's set can be supplied by another within range. Figure 19's example deploys WiFi devices near the doorways in shielded walls to connect mesh members. A Controller is sited in a control room and connected via an Ethernet cable to an enhanced Gateway with an Ethernet interface and running a DeftT relay. Mobile WiFi devices can move throughout the site and maintain connection to both the sensors (through the gateways) and the Controller with its longer-term storage of sensor readings.

Figure 19 assumes LoRaWAN server components are integrated with the Gateway devices (as in many existing MQTT-based deployments) but these devices communicate via DeftT over adhoc WiFi. Only one Gateway has the join server capability (via the communications schema). Multiple Gateways can receive sensor messages which they package as Publications with the device identifier (DevAddr) and unique count (uplink FCnt) as part of the name and publish in a collection of sensor measurements. If Publications are encrypted with a group key, the full name will be unique and only those members of the collection who did not receive the broadcast directly from a sensor will obtain it via the DeftT WiFi interface. (Collection synchronization performs the deduplication process.) Otherwise, if packets are signed with the identity of the Gateway, the shim can discard duplicates before passing to the subscribing application. All Gateways participate in a collection for join messages but only the join server originates LoRaWAN join-accept Publications. The join server may distribute the (encrypted) application key to Gateways that display sensor information or to WiFi devices that may perform more sophisticated tasks e.g., a tablet that analyzes and displays historical sensor input. The Controller receives Publications of sensor data via a TCP connection to the TCP face of the DeftT relay.

Figure 19: IIOT meshed gateways connect a single trust domain

In addition to specifying encryption and signing types, the schema rules control which users can access specific sensors. For example, an outside predictive maintenance analysis vendor can be allowed access to the vibration sensor data from critical motors, relayed through the Internet, while only plant Security can see images from on-site cameras.
5.2. Secure access to Distributed Energy Resources (DER)

The electrical power grid is evolving to encompass many smaller generators with complex interconnections. Renewable energy systems such as smaller-scale wind and solar generator sites must be economically accessed by multiple users such as building owners, renewable asset aggregators, utilities, and maintenance personnel with varying levels of access rights. North American Electric Reliability Corporation Critical Infrastructure Protection (NERC CIP) regulations specify requirements for communications security and reliability to guard against grid outages. Legacy NERC CIP compliant utility communications approaches, using dedicated physically secured links to a few large generators, are no longer practical. DeftT offers multiple advantages over bilateral TLS sessions for this use case:

*Security. Encryption, authentication, and authorization of all information objects. Secure brokerless pub/sub avoids single-point broker vulnerabilities. Large generation assets of hundreds of megawatts to more than 1 gigawatt, particularly nuclear power plants must be controlled securely or risk large-scale loss of life accidents. Hence, they are attractive targets for sophisticated nation-state cyber attackers seeking damage with national security implications. Even small-scale DER generators are susceptible to a coordinated attack which could still bring down the electric grid.

*Scalability. Provisioning, maintaining, and distributing multiple keys with descriptive, institutionalized, hierarchical names. DeftT allows keys to be published and securely updated on-line. Where historically a few hundred large-scale generators could supply all of the energy needs for a wide geographic area, now small-scale DER such as residential solar photovoltaic (PV) systems are located at hundreds of thousands of geographically dispersed sites. Many new systems are added daily and must be accommodated economically to spur wider adoption.

*Resiliency. A mesh network of multiple client users, redundant servers, and end devices adds reliability without sacrificing security. Generation assets must be kept on-line continuously or failures risk causing a grid-wide blackout. Climate change is driving frequent natural disasters including wildfires, hurricanes, and temperature extremes which can impact the communications infrastructure. If the network is not resilient communications breakdowns can disable generators on the grid leading to blackouts.

*Efficiency. Data can be published once from edge gateways over expensive cellular links and be accessed through servers by multiple authorized users, without sacrificing security. For small residential DER systems, economical but reliable connectivity is required to spur adoption of PV compared to
purchasing from the grid. However, for analytics, maintenance and grid control purposes, regular updates from the site by multiple users are required. Pub/sub via DeftT allows both goals to be met efficiently.

*Flexible Trust rules: Varying levels of permissions are possible on a user-by-user and site-by-site basis to tightly control user security and privacy at the information object level. In an energy ecosystem with many DER, access requirements are quite complex. For example, a PV and battery storage system can be monitored on a regular basis by a homeowner. Separate equipment vendors for batteries and solar generation assets, including inverters, need to perform firmware updates or to monitor that the equipment is operating correctly for maintenance and warranty purposes. DER aggregators may contract with a utility to supply and control multiple DER systems, while the utility may want to access production data and perform some controls themselves such as during a fire event where the system must be shut down. Different permissions are required for each user. For example, hourly usage data which gives detailed insight into customer behaviors can be seen by the homeowner, but for privacy reasons might only be shared with the aggregator if permission is given. These roles and permissions can be expressed in the communication rules and then secured by DeftT's use of compiled schemas.

The specificity of the requirements of NERC CIP can be used to create communication schemas that contain site-specifics, allowing applications to be streamlined and generic for their functionality, rather than containing security and site-specifics.

6. Using Defined-trust Communications without DeftT

Parts of the defined-trust communications framework could be used without the DeftT protocol, e.g., to secure structured messages. There are two main elements used in DeftT: the integrated trust management engine and the multi-party communications networking layer that makes use of the properties of a broadcast medium. It's possible to make use of either of these without DeftT. For example, a message broker could implement the trust management engine on messages as they arrive at the broker (e.g., via TLS) to ensure the sender has the proper identity to publish such a message. If a credential is required in order to subscribe to certain messages, that could also be checked. Set reconciliation could be used at the heart of a transport protocol without using defined-trust security, though signing, encryption, or integrity hashing could still be employed.
7. Terms

*certificate thumbprint: the 32 byte SHA256 digest of an entire certificate including its signature ensuring that each thumbprint resolves to one and only one cert and signing chain
*collection: a set of elements denoted by a structured name that includes the identifier of a particular communications schema
*communications schema: defined set of rules that cover the communications for a particular application domain. Where it is necessary to distinguish between the human readable version and the compiled binary version, the modifiers "text" or "binary" will be used. The binary version is placed in a certificate signed by the domain trust anchor.
*DCT: Defined-trust Communications Toolkit. Running code, examples and documentation for defined-trust communications tools, a schema language and compiler, a DeftT implementation, and illustrative examples
*face: maintains tables of DeftT's cState PDUs to manage efficient communications with the system transport in use (UDP multicast, TCP, etc.)
*identity: a certificate signing chain with a particular domain's trust anchor at its root and a unique member certificate as the leaf. The public certificates in the chain contain attributes and capabilities for the leaf member cert. The secret key associated with the public key of the member cert should be securely configured for member use.
*identity bundle - entities in a trust domain are commissioned with an identity bundle of trust anchor, signed communication schema cert, and the signing chain associated with a particular identity
*Publication: a named information object exchanged used by DeftT where name structure and the required identity roles and capabilities for Publications are specified by the schema. Publications are the elements of the sets that are reconciled by DeftT's sync protocol. (Capitalization is used to distinguish this specific use from both the action and more generic use of the term.)
*protocol data unit (PDU): a single unit of information transmitted among entities of a network composed of protocol-specific control information and user data. DeftT uses two types: cState: (from "collection state") and cAdd: (from "collection additions")
*sync protocol: a synchronization protocol that implements set reconciliation of Publications on a subnet making use of cState and cAdd PDUs
*Things: as per [RFC8520], networked digital devices specifically not intended to be used for general purpose computing
*trust anchor: NIST SP 800-57 Part 1 Rev. 5 definition "An authoritative entity for which trust is assumed. In a PKI,
trust anchor is a certification authority, which is represented by a certificate that is used to verify the signature on a certificate issued by that trust-anchor. The security of the validation process depends upon the authenticity and integrity of the trust anchor's certificate. Trust anchor certificates are often distributed as self-signed certificates. In defined-trust communications, a trust anchor is a self-signed certificate which is the ultimate signer of all certificates in use in a trust domain, including the communications schema. From [RFC4949]:

trust anchor definition: An established point of trust (usually based on the authority of some person, office, or organization) which allows the validation of a certification chain.

trust domain: a shorthand form for Defined-trust Communications Limited Domain, a zero trust domain governed by a single trust anchor and communications schema which is enforced at run-time by a library (e.g., DCT) using a signed binary copy of the schema at each member. Nothing is accepted without validation; non-conforming communications are silently discarded. As the schema cert is signed by the trust anchor, a trust domain is uniquely identified by the schema cert's thumbprint. Where context is clear, just domain may be used.

trust-based Relay: (or just Relay) a special-purpose entity that connects a trust domain across different subnets

8. Security Considerations

This document presents a transport protocol that secures the information it conveys (COMSEC in the language of [RFC3552]). Security of data in the application space is out-of-scope for this document, but use of a trusted execution environment (TEE), e.g., ARM's TrustZone, is recommended where this is of concern.

Unauthorized changes to DeftT code could bypass validation of received PDUs or modify the content of outgoing PDUs prior to signing (but only valid PDUs are accepted at receiver; invalid PDUs are dropped by uncompromised member). Although securing DeftT's code is out-of-scope for this document, DeftT has been designed to be easily deployed with a TEE. Revisiting Figure 5, Figure 20 highlights how all of the DeftT code and data can be placed in the secure zone (long-dashed line), reachable only via callgates for the Publish and Subscribe API calls.
Providing crypto functions is out-of-scope of this document. The example implementation uses **libsodium**, an open source library maintained by experts in the field [SOD]. Crypto functions used in any alternative implementation should be of similar high quality.

Enrollment of devices is out of scope. A range of solutions are available and selection of one is dependent on specifics of a deployment. Example approaches include the Open Connectivity Foundation (OCF) onboarding and BRSKI [RFC8995]. NIST NCCOE network layer onboarding might be adapted, treating a communications schema like a MUD URL.

Protecting private identity and signing keys is out-of-scope for this document. Good key hygiene should be practiced, securing private credentials using best practices for a particular application class, e.g. [COMIS][OWASP].

DeftT's unit of information transfer is a Publication. It is an atomic unit sized to fit in a lower layer transport PDU (if needed, fragmentation and reassembly are done in shim or application). All Publications must be signed and the signature must be validated. All Publications start with a Name (Section 2.3.1.3). Publications are used both for ephemeral communication, like commands and status reports, and long-lived information like certs. The set reconciliation-based syncps protocol identifies Publications using a hash of the entire Publication, including its signature. A sync collection can contain at most one instance of any Publication so replays of Publications in the collection are discarded as duplicates on arrival. The current DeftT implementation requires weakly synchronized clocks with a known maximum skew. Publications have a lifetime enforced by their sync collection; their names include a timestamp used both to enforce that lifetime and prevent replay attacks by keeping a Publication in the local collection (but not advertising its existence) until its lifetime plus the skew has passed. (Lifetimes in current applications range from days or years for certs to milliseconds for status and command communications). Publications arriving a skew time before their timestamp or a skew time plus lifetime after their timestamp are discarded.
An attacker can modify, drop, spoof, or replay any DeftT PDU or Publication but DeftT is designed for this to have minimal effect:

1. **Modification** - all DeftT cAdd PDUs must be either signed or AEAD encrypted with a securely distributed nonce group key. This choice is specified in the schema and each DeftT checks at startup that one of these two properties holds for the schema and throws an error if not.

   For signed PDUs each receiving DeftT must already have the complete, fully validated signing chain of the signer or the PDU is dropped. The signing cert must validate the PDU's signature or the PDU is dropped.

   For encrypted PDUs (and Publications) the symmetric group key is automatically and securely distributed using signing identities. Each receiver uses its copy of the current symmetric key to validate the AEAD MAC and decrypt the PDU content. Invalid or malformed PDUs and Publications are dropped.

   cState modification to continually send an older, less complete state in order to generate the sending of cAdds could create a DoS attack but counter measures could be implemented using available DeftT information in order to isolate that entity or remove it from the trust domain.

2. **Dropped PDUs** - DeftT's sync protocol periodically publishes cStates regardless of whether the collection has changed, resulting in (re)sending dropped cAdds (if any). Unlike connection-oriented transports, DeftT can and will obtain any Publications missing from its collection from any member that has a valid copy.

3. **Spoofing** - DeftT uses a trust management engine that validates the signing. Malformed Publications and PDUs are dropped as early as possible.

4. **Replay** - A cAdd is sent in response to a specific cState, so a replayed cAdd must match a current cState and, if so, the cAdd's Publication(s) will be filtered for duplicates and obsolescence as described above. A cAdd that doesn't match a current cState will be dropped on arrival.

Peer member authentication in DeftT comes through the integrated trust management engine. Every DeftT instance is started with an identity bundle that includes the domain trust anchor, the schema in certificate format signed by this trust anchor, and its own member identity chain with a private identity key and the chain signed at the root by trust anchor. Members publish their identity chains
before any Publications are sent. The trust management engine
unconditionally drops any Publication or PDU that does not have a
valid signer or whose signer lacks the role or capabilities required
for that particular Publication or PDU.

DeftT takes a modular approach to signing/validation of its PDUs and
Publications, so a number of approaches to integrity, authenticity,
and confidentiality are possible (and several are available at
[DCT]). Security features that are found to have vulnerabilities
will be removed or updated and new features are easily added.

A compromised member of a trust domain can only build messages that
match the role and attributes in its signing chain. Thus, a
compromised lightbulb can lie about its state or refuse to turn on,
but it can't tell the front door to unlock or send camera footage to
a remote location. Multiple PDUs could be generated, resulting in
flooding the subnet. There are possible counter-measures that could
be taken if some detection code is added to the current DeftT, but
this is deferred for specific applications with specific types of
threats and desired responses.

DeftT's modular structure allows for any cryptographic methods to be
used as sigmgrs. New methods can easily be added to the transport as
long as they present the same API.

The example implementation's encryption modules provide for
encryption on both cAdd PDUs and Publications. The latter must be
signed by the originator in addition to being encrypted. This is not
required for cAdd PDUs, so the specific entity that sent the cAdd
cannot be determined but the Publications it carries must be signed,
even if not encrypted. In DeftT, any member can resend a Publication
from any other member (without modification) so group encryption (in
effect, group signing) is no different. Some other encryption
approaches are provided whose potential vulnerabilities are
described with their implementations and a signed, encrypted
approach is also available [DCT]. [DCT] relies on the crypto library
libsodium and on linux random implementations with respect to
entropy issues. In general, these are quite application-dependent
and should be further addressed for particular deployments.

9. IANA Considerations

This document has no IANA actions.

10. Normative References

[RFC1422] Kent, S., "Privacy Enhancement for Internet Electronic
Mail: Part II: Certificate-Based Key Management", RFC


11. Informative References


al., M. C. E., "Device Management Requirements to Secure Enterprise IoT Edge Infrastructure", April 2021, <https://www.wwt.com/white-paper/device-management-


IEC, "Power systems management and associated information exchange - Data and communications security - Part 8:


Saarinen, M., Ed. and J. Aumasson, "The BLAKE2 Cryptographic Hash and Message Authentication Code


Contributors

Lixia Zhang
UCLA

Email: lixia@cs.ucla.edu

Roger Jungerman
Operant Networks Inc.
Roger contributed significantly to Section 5.

**Authors' Addresses**

Kathleen Nichols  
Pollere LLC  

Email: nichols@pollere.net

Van Jacobson  
UCLA  

Email: vanj@cs.ucla.edu

Randy King  
Operant Networks Inc.  

Email: randy.king@operantnetworks.com