Secure Event Types in Content-Based, Multi-Domain Publish/Subscribe Systems

Lauri I.W. Pesonen
University of Cambridge, Computer Laboratory
JJ Thomson Avenue, Cambridge, CB3 0FD
United Kingdom
lauri.pesonen@cl.cam.ac.uk

Jean Bacon
University of Cambridge, Computer Laboratory
JJ Thomson Avenue, Cambridge, CB3 0FD
United Kingdom
jean.bacon@cl.cam.ac.uk

ABSTRACT

Publish/subscribe research has so far been mostly focused on efficient event routing, event filtering, and composite event detection. The little research that has been published regarding security in publish/subscribe systems has been tentative at best. This paper presents a model for secure type names, and definitions for type-checked, content-based publish/subscribe systems. Our model provides a cryptographically verifiable binding between type names and type definitions. It also produces self-certifiable type definitions that guarantee type definition authenticity and integrity. We also consider type management in a large-scale publish/subscribe system and present a way for delegating management duties to type managers by issuing SPKI authorisation certificates. We feel that secure names are a prerequisite for most other security related work with publish/subscribe systems.

1. INTRODUCTION

Publish/subscribe has emerged as a popular communication paradigm for Internet-wide, large-scale distributed systems. Modern large-scale publish/subscribe systems are based on a peer-to-peer network of brokers that routes events from publishers to subscribers in an efficient manner. A peer-to-peer network re-balances itself dynamically in case of node joins and leaves, as well as node and network link failures. This makes a peer-to-peer network very fault-tolerant and scalable. This paper focuses specifically on type-checked, content-based publish/subscribe systems with a peer-to-peer broker network, where events are instances of predefined event types and subscriptions may include filter expressions that filter events based on their content.

We envision a large-scale, multi domain publish/subscribe system where multiple domains co-operate together in forming a shared broker network, as seen in Fig. 1. The incentive for domains to join the network is twofold: on the one hand domains are interested in implementing shared applications with other domains, e.g. one domain produces events while another one consumes them. On the other hand, even with domain-internal applications, the increased number of brokers increases the reliability and coverage of the broker network, i.e. a broker network with more nodes is increasingly fault-tolerant and the increased coverage enables the domain to reach a larger geographic area with lower infrastructure investments.

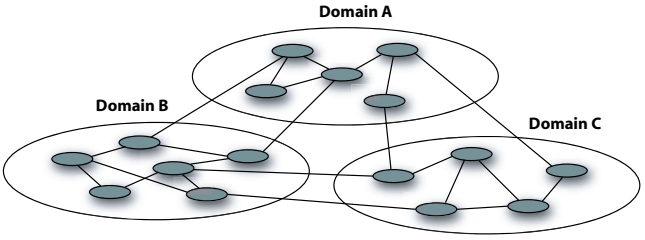

Figure 1: Three domains co-operate in order to form a shared broker network.

Most of the research in publish/subscribe systems has concentrated on efficient event routing, event filtering, and composite event detection. Very little work has been done pertaining to the security aspects of publish/subscribe systems. We feel that secure type definitions and names are a basic building block for more complex security features for publish/subscribe systems, e.g. access control, event integrity and confidentiality etc.

We propose an event type definition format for content-based publish/subscribe systems which binds the type name and type definition to each other in a secure fashion, and guarantees type authenticity and integrity by using public-key cryptography: a *secure event type*. This prevents: (i) forged types, where a malicious party tries to disrupt the data flow by introducing a type definition to the system with the same name as another legitimate type; (ii) tampered types, where a malicious party changes an existing type definition; and (iii) accidental name collisions, where two parties both introduce a new type to the system with the same name. In addition to solving the above problems, secure event types form a basis for, for example, access control where access control policy can refer securely and unambiguously to type and attribute names.

The rest of the paper is organised as follows. Sect. 2 introduces the reader to publish/subscribe and decentralised trust management. Our model for secure event types is presented in Sect. 3. Sect. 4 describes the changes that were

SEM 2005 September 2005 Lisbon, Portugal

made to the Hermes publish/subscribe middleware in order to support secure event types. We discus related work in Sect. 6 and conclude the paper with Sect. 7 by outlining possible future work and summarising our research contributions.

2. BACKGROUND

This section provides a brief introduction to type-checked, content-based publish/subscribe systems and decentralised trust management. Our work is specific to type-checked, content-based publish/subscribe systems where published events are instances of predefined event types and subscriptions can include filter expressions that filter events based on their content. The publish/subscribe middleware performs type-checking of requests during publication and subscription time to verify that the publication/subscription filter conforms to the specified event type. Decentralised trust management concepts are used in delegating event type management duties to other principals.

2.1 Publish/Subscribe

We have implemented secure event types on a Hermes-like [13, 12] publish/subscribe middleware called Maia. Maia is based on Hermes, but includes security features that were not part of the original Hermes design.

Hermes is a content-based publish/subscribe middleware with strong event typing. It is built on a peer-to-peer routing substrate to provide scalable event dissemination and fault tolerance in case of node or network failures.

Hermes systems consists of *event brokers* and *event clients*, the latter being *publishers* and/or *subscribers*. Event brokers form an application-level overlay network which performs event propagation by means of a content-based routing algorithm. Event clients publish and/or subscribe to events in the system. An event client connects to a *local broker*, which then becomes *publisher-hosting*, *subscriber-hosting*, or both. An event broker without connected clients is called an *intermediate broker*.

A feature of Hermes, that this work relies on, is support for *event typing*: every published event (or *publication*) in Hermes is an instance of an *event type*. An event type defines a *type name* and a set of *attributes* that consist of an *attribute name* and an *attribute type*. Supported attribute types typically depend on the types supported by the language used to express subscription filters. For example, XPath 2.0[20], which can be used to define filters for XML documents, defines primitive types like string, boolean, decimal, float, double, duration, time, and date etc.

Another Hermes-specific addition to traditional content-based publish/subscribe is support for *event type hierarchies*. In Hermes event types can be organised into inheritance hierarchies, where an event type inherits all of the attributes defined and inherited by its ancestor. In addition to making defining new types easier, type hierarchies enable a subscriber to subscribe to a supertype in an event hierarchy and receive notification of events of that specific type as well as all its subtypes. Our work is compatible with event type hierarchies, but the proposed design would require minor additions in order to support them fully. For example, `extend` should be added to the set of type management operations (see Sect. 3.3). In general we have tried to keep the design as widely compatible with type-checked content-based publish/subscribe systems as possible.

2.2 Decentralised Trust Management

Decentralised trust management was first introduced as a concept by Matt Blaze et al. in [4]. Blaze's *PolicyMaker* was later followed by *KeyNote*[3], the simpler incarnation of PolicyMaker, and the *Simple Distributed Security Infrastructure* (SDSI)[14], which was later integrated with the *Simple Public-Key Infrastructure* (SPKI)[6, 7] to create SPKI/SDSI 2.0[5].

The central idea in decentralised trust management is to decentralise access control policy management, decision making, and credential management. This is achieved by implementing a capability-based approach to access control where the *owner* of an *object* is responsible for access control policy and credential management for that particular object. Distributing management responsibilities over all of the principals results in an extremely scalable access control system.

In SPKI the decentralisation is based on *certificate loops*. A typical certificate loop is depicted in Fig. 2 where the owner, P_a of an object O, grants an SPKI authorisation certificate, C_{ab}, with access rights, A_{ab}, to P_b. P_b then further delegates access rights A_{bc}, where $A_{bc} \subseteq A_{ab}$, to P_c by granting P_c another delegation certificate C_{bc}. Now, when P_c wants to access O, she shows P_a both certificates C_{ab} and C_{bc}. P_a is now able to form a certificate chain from P_c to P_b via C_{bc} and from P_b to itself via C_{ab}. Finally P_c authenticates herself to P_a by proving ownership of the key-pair P_c[1]. P_c does this by executing a public-key challenge-response protocol with P_a. This completes the certificate chain which now forms a *certificate loop* flowing from P_a to P_b to P_c and back to P_a again. P_a has now verified that P_c is authorised to access O within the privileges granted by $A_{ab} \cap A_{bc}$[2]. Typically the verification is performed by an access control service rather than P_a.

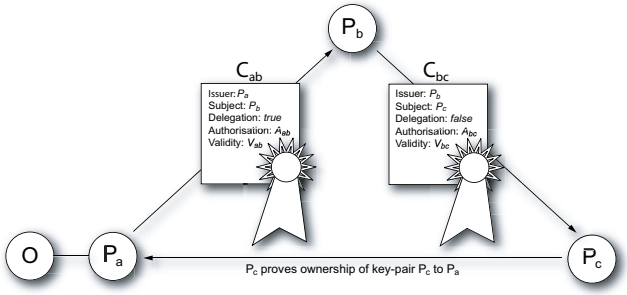

Figure 2: An SPKI authorisation certificate loop with three principals and two level delegation.

An SPKI authorisation certificate is a 5-tuple containing the following five items: *Issuer*, *Subject*, *Delegation*, *Authorisation*, and *Validity*. *Issuer* is the public-key (or a hash of the public-key) of the issuer of the certificate. *Subject* is the public-key of the entity the certificate is issued to. *Delegation* is a boolean value specifying whether the *Sub-*

[1]In SPKI a *principal* is a cryptographic key, capable of generating digital signatures. Therefore, SPKI treats principals and public-keys as synonyms.

[2]The verification process will also consider optional validity conditions (e.g. expiration dates) for each certificate in the chain. An invalid certificate will break the chain and thus render the whole chain invalid.

ject is permitted to further propagate the *Authorisation* in this certificate. In a certificate chain all certificates bar the last one must have *Delegation* set to *true*, otherwise the certificate chain is not valid. *Authorisation* is an application specific representation of access privileges granted to the *Subject* by the *Issuer*. And finally, *Validity* defines the date range when the certificate is valid as well as optional on-line validity tests, e.g. *certificate revocation lists* (CRLs).

Two certificates are reduced to a single 5-tuple as follows:

$$< I_1, S_1, D_1, A_1, V_1 > + < I_2, S_2, D_2, A_2, V_2 > \Rightarrow < I_1, S_2, D_2, A_1 \cap A_2, V_1 \cap V_2 >$$
$$\text{iff } A_1 \cap A_2 \neq \emptyset,\ V_1 \cap V_2 \neq \emptyset,\ S_1 = I_2 \text{ and } D_1 = true$$

This 5-tuple reduction rule is applied recursively to the certificate loop in order to collapse the loop to a single 5-tuple which will then be evaluated.

Our work relies on SPKI authorisation certificates to propagate type management authorisation from a *type issuer* to other *type managers* in a decentralised and scalable fashion. We use *type issuer* to refer to the principal that created a new type, while *type manager* refers to a principal that has been authorised to manage an existing type. We refer to the SPKI authorisation certificates as delegation certificates in order to emphasise the fact that they are used to delegate type management duties. For example, a type issuer is able to authorise a type manager to add, remove, and rename attributes in an event type by issuing a delegation certificate to the type manager with the access rights `addAttribute`, `renameAttribute`, and `removeAttribute`. See Sect. 3.3 for a detailed description of delegation certificates in type management and possible type management operations.

3. SECURE EVENT TYPES

We approach the problem of secure event type definitions in a publish/subscribe system by defining a secure name space for type names. We propose incorporating the type issuer's *public-key* to the type name. A public-key is globally unique [7], thus it defines a globally unique name space. Type issuer specific name spaces will prevent accidental name collisions in the publish/subscribe system.

We also propose digitally signing the type definitions with the private-key corresponding to the public-key used to define the name space. Because the private-key used to sign the type definition corresponds to the public-key incorporated in the name of the type definition, the type name is bound to the type definition. Both the authenticity and integrity of the type definition can be verified by verifying the type definition signature with the public-key in the type name. If the signature verifies correctly, it means that the issuer is the owner of the name space defined by the public-key (authenticity), and that the type definition has not been tampered with since it was issued (integrity). Only the owner of the name space is able to issue new types for that name space, because the digital signature on the type definition must be bound to the public-key defining that name space.

A secure event type definition consists of six items as depicted in Fig. 3 where *ti* signifies the *type issuer*.

The items can be divided into three groups based on their function in the event type definition: the first three items form the *name tuple*, which identifies the secure type definition uniquely in the system; the fourth item is the set of

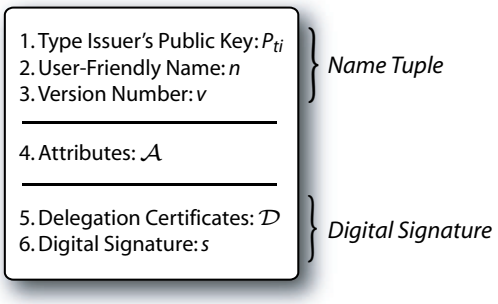

Figure 3: The contents of a secure event type definition.

attribute definitions which is the actual payload of the secure type definition; and the remaining two items are relevant in verifying the *digital signature* of the secure event type definition, which ensures its authenticity and integrity. The following sections will describe each group in more detail.

3.1 Name Tuple

The name of an event type is used to reference a type definition from publish/subscribe messages (i.e. advertisement, subscriptions, and publications). It is crucial that a specific *type name* references the same *type definition* for all clients and messages; the name tuple must identify a type definition unambiguously.

We replace the traditional type name with a 3-tuple consisting of the *type issuer's public-key*, a *user-friendly name*, and a *version number*. The name tuple defines a unique name for the type definition and creates a secure one-to-one mapping between type name and type definition.

Public-Key. The type issuer's public-key defines a globally unique name space while at the same time specifying the owner of that name space. Because the name space is globally unique, accidental name collisions are impossible[3]. A type definition within a name space is valid only if the signature of that type definition can be verified and linked back to the public-key in the name tuple. Because the signature can only be created with the type issuer's private-key the malicious users are not able to forge that link, and therefore are unable to introduce forged type definitions into the system. See Sect. 3.3 for more details on how the digital signature is linked back to the name space.

User-Friendly Name. Where the public-key defines a globally unique name space, the user-friendly name can be used to build naming hierarchies within that name space. For example, the reverse-DNS naming scheme produces hierarchical names like `uk.ac.cam.cl.ActiveOffice.DoorEvent`. Naming hierarchies enable the event type owner to express a semantic structure among multiple related event types, e.g. all type definitions related to a single publish/subscribe application can be grouped together. Also, including information about the type issuer in the type name will be beneficial for the application developer (cf. Java package naming conventions[16]).

[3]We assume that the name-space owner is able to avoid accidental name collisions within her own name space.

Version Number. We must assume that type definitions in a large-scale publish/subscribe system need to evolve during their lifetime. Thus a type manager must be able to update a type definition while it is in use without disrupting running clients (see the end of Sect. 3.3 for possible type management operations).

We propose including a version number in the name tuple, which allows multiple versions of a single event type to coexist in the publish/subscribe system. A new version number is created for each new version of a type definition. Since the version number is part of the name tuple, changing it effectively renames the event type. This means that introducing an updated event type into the system does not interfere with existing clients since they are subscribed to and/or publish events of a different type name. Existing clients will continue publishing and receiving events of the old type until they have been modified to support the new type. New clients, implemented against the updated type definition, can use the new type immediately. We assume that clients have to be manually modified to be able to use an updated type definition. The manual modification might be as simple as adding the new type definition to the client's configuration files, or it might be a complete re-implementation of the client.

The version number is created based on the UUID (Universally Unique Identifier) specification[10]. A UUID is a 128-bit value which is guaranteed to be unique from all other UUIDs until the year 3400. We use UUIDs as version numbers because they are collision resistant, i.e. multiple type managers can create UUIDs at the same time without accidentally creating the same version number. If the version number was a simple integer that was incremented after each type definition update, we would have to synchronise between type managers updating the same event type at the same time to avoid version number collisions. Event type versioning will be discussed in more detail in Sect. 4.4.

3.2 Attributes

Attributes define the event type structure that publications must conform to. An attribute definition consists of a user-friendly name, a unique identifier, and a type. Supported types depend on the subscription filter language.

An attribute definition defines a mapping between the attribute name and the unique identifier. The name has to be unique within the context of that particular version of the type definition whereas the unique identifier has to be unique within the context of all versions of the type definition. A unique identifier therefore identifies a single attribute among all the attributes defined for that event type in all its different versions.

The unique identifier is implemented as a UUID, because of the collision-resistance properties of UUIDs. Again, we want to avoid having to synchronise multiple type updates in order to avoid identifier collisions.

The publish/subscribe clients refer to attributes using attribute names, whereas the intermediate brokers use the unique identifiers. Before routing the event through the broker network the publisher-hosting broker translates the names to UUIDs based on the publisher's version of the type definition. The subscriber-hosting broker translates the UUIDs back to attribute names based on the subscriber's version of the type definition before delivering the event to the subscriber. This enables multiple versions of an event type to co-exists in the publish/subscribe system at any given time. Type version translation is covered in more detail in 4.4.

3.3 Digital Signature

The digital signature on an event type guarantees authenticity and integrity of the type definition. The signature is calculated over all of the items in the type definition except the signature field (see items 1 through 5 in Fig. 3), i.e. it provides protection for all items in the type definition.

Because the signature includes the name and version number of the type definition, and it is linked to the public-key in the name tuple, it binds the type definition to the name tuple.

Delegation Certificates. We assume that in an Internet-scale publish/subscribe system event types are long-lived and therefore the management of an event type cannot be the responsibility of a single principal. Unfortunately the security of our secure event types is based on the fact that the public-key in the name tuple verifies the digital signature on the type definition. Thus, only the owner of the public-key in the name tuple, i.e. the type issuer, is able to create a digital signature which is verifiable with that public-key. This means that only the type issuer is able to issue updated versions of the type definition.

Delegation certificates enable the type issuer to delegate type management duties to type managers. The digital signature and the public-key in the name tuple have to be linked together so that a verifier can trust the authenticity of the type definition. If a type manager would edit the event type, sign it, and reintroduce it to the system this link would be broken, because the type manager is unable to sign the type definition with the type issuer's private-key. In order to maintain the link in a self-certifiable package we add a set of delegation certificates which link the type manager's key pair to the type issuer's public-key. Now the type manager's digital signature can be linked to the public-key in the name tuple by following the certificate chain from the signature to the type issuer's public-key, and the authenticity and integrity of the type definition can be verified (see Sect. 2.2 and Fig. 2).

Figure 4 depicts three different cases of type management. In the first column the type issuer, P_a, has created or updated the type definition and signed it. The type issuer's signature is directly verifiable with the public-key in the name tuple, so there is no need for a delegation certificate and thus the delegation certificate field in the type definition is left empty. In the second column the type manager, P_b, has updated the type definition. P_b includes in the updated type definition a delegation certificate which links the signature, $S(P_b)$, to P_a. The third column is similar to the second column, except in this case the certificate chain linking $S(P_n)$ to P_a consists of more than one certificate.

The set of delegation certificates is there to link the current signature to the public-key in the name tuple. The type manager updating the type definition always replaces the previous set of delegation certificates with a set of delegation certificates that link her to the name tuple. In the default case where the type issuer issues the type definition and manages all updates, there is no need to include any delegation certificates (see the first column in Fig. 4).

The type issuer is able to authorise type managers to man-

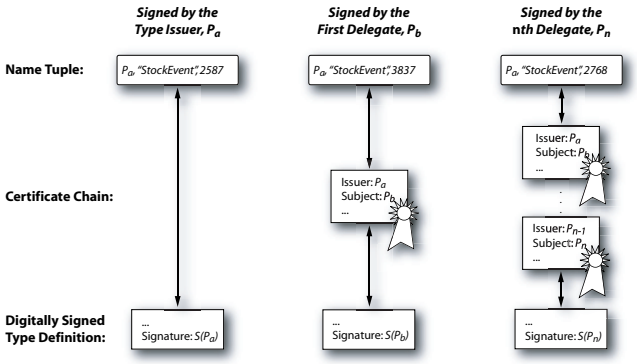

Figure 4: Verifying the name-signature link with and without delegation certificates.

age a type definition by granting them delegation certificates with the appropriate access rights. The delegation certificates are delivered to the type managers out-of-band. Out-of-band delivery enables the type issuer to grant delegation certificates even after the type definition has already been issued. If the delegation certificates were embedded in the type definition, the type issuer would have to update the type definition every time she issued a new delegation certificate. This would become especially cumbersome if type managers were also issuing delegation certificates.

Delegation certificates enable fine grained authority delegation. That is, the type issuer has fine grained control over what management rights are delegated to a type manager, if the type manager is able to further delegate those management rights, and what validity conditions the created certificate has.

In the case of secure event types possible access rights are `addAttribute`, `removeAttribute`, `editAttributeName`, and `editAttributeType`. This can be extended to single attributes, e.g. `editAttributeName`, iff `UUID=9058`. We expect that in most cases limiting the delegate's authority would not be necessary.

4. CHANGES TO HERMES

We made a number of changes to the original Hermes design in order to add support for secure event types. Some of the changes were necessary for Hermes to be able to support secure event types. Other changes were made possible by secure event types and enabled us to simplify the Hermes design. The following sections describe in detail the more important changes that were done.

4.1 Type Storage

The original Hermes design uses a *distributed hashtable* (DHT)[15] to store event types in the broker network. The name of the event type is used as a key when inserting the type definition into the DHT. The unreliable nature of peer-to-peer networks demands that the stored type definition be replicated among multiple nodes. That is, because peer-to-peer nodes may leave the network at any given time, not to mention the possibility of node and network failures, the content stored at a specific node must be replicated to other nodes in order to guarantee the availability of specific content with high probability. Maintaining the DHT thus results in a lot of unnecessary network traffic when content is copied to replica nodes during inserts, and node joins and leaves. Even with replication, the DHT can only provide availability with high probability based on the number of replica nodes. In the worst case the requesting node is cut off from all replica nodes and thus not able to access the content.

The self-certifiability of secure event types enables brokers to verify the authenticity and integrity of type definitions. This allows a local broker to trust a type definition received from a client. That is, we can allow publish/subscribe clients to provide the local broker with the type definition rather than having to store the type definitions in the broker network as is the case with Hermes.

We argue that developers implementing publish/subscribe applications that handle specific event types need to have the definitions of those types available to them during development time. The type definitions would be delivered to the developers out-of-band, e.g. as downloads from a web page or a central type repository. Since the type definitions are part of the development process it would be simple to include them in the packaging of the publish/subscribe clients. The client code would then be able to pass the type definition on to the local broker as a part of an appropriate publish/subscribe request (`advertise` or `subscribe`). Finally the broker would verify the authenticity and integrity of the client-provided type before executing the client's request.

Only the local brokers need to do type-checking. The local broker of a publisher type-checks the submitted publication before it is routed through the broker network. Similarly the local broker of a subscriber type-checks the subscription filter before passing the subscription on to the broker network. The intermediate brokers assume that the publications and subscriptions have already been type-checked by the local brokers, thus only the local brokers need to be aware of type definitions.

By relying on publish/subscribe clients to provide type definitions to local brokers we remove the need for maintaining a type repository in a DHT, thus lowering the amount of network traffic and making the broker network design more elegant and simpler in general.

4.2 API Changes

The new approach to type storage presented in the previous section and the introduction of the name tuple result in changes to the Hermes API as shown in Fig. 5. Because type definitions are not installed in the publish/subscribe system any more, as explained above, there is no need for a specific type management API with the operations `addEventType`, `removeEventType`, and `modifyEventType`. Type definitions are provided to the brokers by the clients as parameters to the `advertise` and `subscribe` operations. In both cases the old API operations referred to the event type with the type name, whereas in the updated API the type name has been replaced by the actual secure type definition.

The other operations in the API have been updated to refer to a type definition with a cryptographically secure hash of the type's name tuple instead of the type name. We use a hash of the name tuple in order to save some bytes in network communications. The name tuple consists of a public-key, a name, and a UUID. A DER-encoded 1024-bit RSA public-key is at least 160 bytes long. Adding another 16 bytes for the UUID means that the size of the name tuple is already 176 bytes without the user-friendly name,

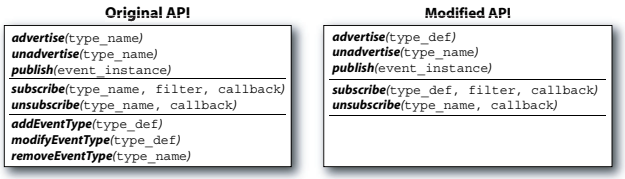

Figure 5: The original and the modified Hermes API.

which can be anything from a single byte to hundreds of bytes. Compressing the name tuple with a hash function results in a shorter identifier depending on the size of the hash function's output, which vary between 128 bits (16 bytes) and 512 bits (64 bytes) in length. In light of the recent breaks related to MD5[19] and SHA-1[18], we suggest using SHA-256[8]. SHA-256 produces a 256-bit (32 byte) digest, which is significantly shorter than what the name tuple would be.

4.3 Message Routing

In addition to using the event type name as a key in the DHT when storing type definitions, Hermes uses the type name as a node id when routing events through the broker network. Hermes chooses a *rendezvous node* from all the broker nodes by hashing the type name in order to create a node id. An event dissemination tree is then created in the broker network by routing advertisement and subscription messages towards the rendezvous node. Because the rendezvous node id is created by hashing the name of the event type, both publishers and subscribers are able to create the node id. Publications are then routed based on the event dissemination tree to subscribers. The Hermes routing algorithm is explained in more detail in [13] and [12].

Simply hashing the name tuple would result in each event type version having a different rendezvous node, because $h(P||n||v_1) \neq h(P||n||v_2)$. This would result in independent event dissemination trees for each type version and, because of this, in unnecessary routing state and sub-optimal routing performance. Instead of hashing the whole name tuple, we ignore the version number and hash only the public-key and the name: $h(P||n)$. This results in a common rendezvous node and optimal routing performance for all versions of an event type.

4.4 Type Version Translation

Another side effect of adding the version number to the name tuple is that subscribers will not receive events from publishers that are publishing events of a different version of the same event type. This presents a real problem when either the publisher or the subscriber updates its event type version unilaterally. In such a case the publisher and subscriber will lose connectivity with each other, because the different event type versions are treated as unrelated types.

Instead of routing events of specific event type versions, we translate the published event to a *transit time event* at the local broker of the publisher after the event has been type-checked by the broker (see Fig. 6). The transit time event is simply a collection of name-value pairs that were copied from the publication. It also includes the public-key and the name from the name tuple, but not the version number.

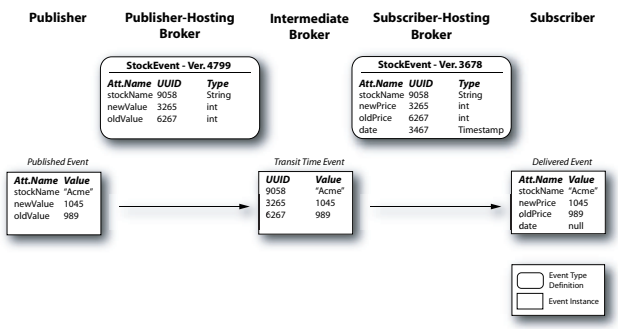

Figure 6: Translation to and from transit time events.

The transit time event is then routed through the broker network to all subscribers of that event type. The local broker of the subscriber translates the transit time event to an instance of the subscriber's version of the type definition. Attributes that are present in the transit time event, but not in the subscriber's version of the type definition are ignored. Similarly attributes that are not present in the transit time event, but are defined in the subscriber's version of the event type are set to `null` in the event instance.

The transit time event uses UUIDs instead of names to refer to attributes. The unique id numbers guarantee that attributes are always unambiguously identifiable regardless of their name. This allows type managers to add and remove attributes, as well as to rename existing attributes and reintroduce old attribute names while maintaining interoperability between different versions of a type definition. That is, in Fig. 6 the publisher uses the version `4799` of the type `StockEvent` with the name `newValue` for the attribute with UUID `3265`, while the subscriber uses the version `3678` of the same type with name `newPrice` for the same attribute. Although the names for the attribute are different for the publisher and the subscriber, the UUIDs are the same which allows the subscriber's local broker to deliver the attribute to the subscriber as `newPrice`. Similarly a new version of the type definition derived from version `3678` might include a new attribute with the name `newValue`, but with a UUID `3879`. Because the UUID (`3879`) differs from the UUID (`3265`) of the original `newValue` attribute, there is no risk of confusing one attribute with the other when converting instances of the type to and from transit time events.

5. PERFORMANCE

The most significant performance penalty in verifying secure event types is caused by digital signature verification. The other related operations, e.g. SPKI 5-tuple reduction, are very cheap in comparison.

Event types need to be verified when a publish/subscribe client provides an event type to the local broker as a part of an advertisement or a subscription request. Publications refer to an already verified event type and thus do not need to be verified individually.

In a naïve implementation a broker verifies every client-provided event type for every advertisement and subscription request separately. An optimised implementation would cache the verification result of each event type and simply compare the already verified type to the event types in sub-

sequent requests thus avoiding the expensive signature verification.

The broker can also store client-provided event types locally after verification. This enables the broker to load and verify those event types as part of the broker startup sequence. As we can assume that the set of event types in use in a publish/subscribe system is relatively static, i.e. the publish/subscribe clients advertise and subscribe to the same event types most of the time, the bulk of the cost of verifying those types is paid in advance while the broker is starting up. The routing performance of a broker is only affected by new types and type versions introduced to the system that have not been verified yet.

The cost of verifying an event type depends on the length of the certificate chain in that event type. Therefore the impact of event type verification can be reduced even more by the broker caching also verified delegation certificates. The cached certificates can then be used in verifying certificate chains in other event types and event type versions where the certificate chains contain cached certificates. For example, if a type manager introduces a new version of an already cached event type to the system, the broker can use the cached certificates in verifying the certificate chain from the type owner to the type manager.

Caching event type verification results means that only new event types and event type versions have to be verified. Thus the performance impact caused by secure event type verification differs from application to application depending on how many event types the application requires and on how static the set of used event types is. Therefore we felt that a simulation would not be useful, because it could not provide us with universally meaningful results on the performance impact caused by secure event types.

6. RELATED WORK

Wang et al. present in [17] a number security issues in Internet-scale publish/subscribe systems that need to be addressed. The paper covers problems related to authentication, data integrity and confidentiality, accountability, and service availability. We feel that secure event types provide a foundation for solving these problems. The secure name space defined by a principal's public-key provide a basis for event encryption and access control. Without trusted names, implementing either would be very difficult.

Other research on securing publish/subscribe systems includes [9], where Miklós presents an access control mechanism for large-scale publish/subscribe systems, which supports access control decisions based on publication and subscription filter content. Belokosztolszki et al. present a role-based access control architecture for publish/subscribe systems in [2]. In both cases the lack of trusted names renders the scheme infeasible in a multi-domain setting.

Opyrchal and Prakash concentrate on the separate problem of providing confidentiality for events during the last hop from the local broker to the subscribers with as few encryptions as possible [11]. We assume that local brokers have enough resources to maintain secured network connections to clients in which case the efficient encryption of single events for the last hop is not relevant.

7. CONCLUSIONS AND FUTURE WORK

Security is acknowledged to be a crucial concern for the development of publish/subscribe systems, yet much of the research done in this area has been speculative at best.

This paper presents a model for secure event type definitions in type-checked, content-based publish/subscribe systems. The scheme provides self-certifiable type definitions which allow both event clients and brokers to verify the definition authenticity and integrity. Although our design is based on Hermes, it is applicable to type-checked, content-based publish/subscribe systems in general. The design can also be extended to support event type hierarchies by a few minor additions, e.g. by adding an `extend` access privilege to the set of possible access rights.

We feel that secure event types are the foundation for a secure publish/subscribe middleware. Other services like access control can then be built on this foundation. For example, in the case of access control we can bind access rights to type and attribute names, because we can trust those names to be unforgeable and unique. Unforgeable and unique names will become even more important when publish/subscribe systems mature and are offered commercially across multiple independently managed domains.

In addition to secure event types, we have also introduced a scheme for managing event types in a large-scale publish/subscribe system. We feel that such a system must be able to run continuously regardless of type management operations. Our approach enables type managers to update existing type definitions transparently without affecting existing clients. We also support the delegation of type management duties, which we see as an equally important feature when considering the expected lifetime of event types in a large-scale, highly decentralised system.

This research is part of a project, EDSAC21, to provide secure middleware for large-scale, widely distributed applications. We are currently working on introducing decentralised infrastructure and application level access control to multi-domain, type-checked, content-based publish/subscribe systems, and enforcing application level access control by encrypting events [1].

8. ACKNOWLEDGEMENTS

We would like to thank David Eyers for his valuable comments. The work of Lauri Pesonen is supported by UK EPSRC (GR/T28164/01).

9. REFERENCES

[1] J. Bacon, D. M. Eyers, K. Moody, and L. I. Pesonen. Securing publish/subscribe for multi-domain systems. In *Proc. of the 6th International Middleware Conference (MW'05)*, Grenoble, France, Nov. 2005. ACM/IFIP/USENIX. To appear.

[2] A. Belokosztolszki, D. M. Eyers, P. R. Pietzuch, J. Bacon, and K. Moody. Role-based access control for publish/subscribe middleware architectures. In *Proc. of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, ACM SIGMOD, San Diego, CA, USA, June 2003. ACM.

[3] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust management for public-key infrastructures (position paper). In *Proc. of the Cambridge 1998 Security Protocols International Workshop*, volume 1550, pages 59–63, 1998.

[4] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proc. of the IEEE Conference on Security and Privacy*, Oakland, CA, USA, May 1996. IEEE.

[5] CIS. SDSI (a simple distributed security infrastructure). `http://theory.lcs.mit.edu/~cis/sdsi.html`, Sept. 2001.

[6] C. Ellison. SPKI requirements. RFC 2692, Internet Engineering Task Force, Sept. 1999.

[7] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylönen. SPKI certificate theory. RFC 2693, Internet Engineering Task Force, Sept. 1999.

[8] FIPS/NSA. Fips 180-2: Secure hash standard (SHS), Aug. 2002.

[9] Z. Miklós. Towards an access control mechanism for wide-area publish/subscribe systems. In *Proc. of the 1st International Workshop on Distributed Event-based Systems (DEBS'02)*, July 2002.

[10] The Open Group. *DCE 1.1: Remote Procedure Call*, 1997.

[11] L. Opyrchal and A. Prakash. Secure distribution of events in content-based publish subscribe systems. In *Proc. of the 10th USENIX Security Symposium*. USENIX, Aug. 2001.

[12] P. R. Pietzuch and J. Bacon. Peer-to-peer overlay broker networks in an event-based middleware. In H. A. Jacobsen, editor, *Proc. of the 2nd International Workshop on Distributed Event-Based Systems (DEBS'03)*, ACM SIGMOD, San Diego, CA, USA, June 2003. ACM.

[13] P. R. Pietzuch and J. M. Bacon. Hermes: A Distributed Event-Based Middleware Architecture. In *Proc. of the 1st International Workshop on Distributed Event-Based Systems (DEBS'02)*, pages 611–618, Vienna, Austria, July 2002. IEEE.

[14] R. L. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession, Oct. 1996.

[15] A. Rowstron and P. Druschel. PAST: A large-scale, persistent peer-to-peer storage utility. In *HotOS VIII*, Schoss Elmau, Germany, May 2001.

[16] Sun Microsystems, Inc. *Code Conventions for the Java^{TM} Programming Language*, Apr. 1999.

[17] C. Wang, A. Carzaniga, D. Evans, and A. L. Wolf. Security issues and requirements in internet-scale publish-subscribe systems. In *Proc. of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)*, Big Island, HI, USA, 2002. IEEE.

[18] X. Wang, Y. L. Yin, and H. Yu. Collision search attacks on SHA1. Technical report, Shandong University, Shandong, China, 2005.

[19] X. Wang and H. Yu. How to break md5 and other hash functions. In R. Cramer, editor, *EUROCRYPT*, volume 3494 of *LNCS*, pages 19–35. Springer, 2005.

[20] World Wide Web Consortium. *XML Path Language (XPath) Version 2.0*, Apr. 2005.