Distributed Middleware Enforcement of Event Flow Security Policy

Matteo Migliavacca[1], Ioannis Papagiannis[1], David M. Eyers[2], Brian Shand[3], Jean Bacon[2], and Peter Pietzuch[1]

[1] Imperial College London, {migliava, ip108, prp}@doc.ic.ac.uk
[2] University of Cambridge, {*firstname.lastname*}@cl.cam.ac.uk
[3] CBCU/ECRIC, National Health Service, brian.shand@cbcu.nhs.uk

Abstract. Distributed, event-driven applications that process sensitive user data and involve multiple organisational domains must comply with complex security requirements. Ideally, developers want to express security policy for such applications in data-centric terms, controlling the flow of information throughout the system. Current middleware does not support the specification of such end-to-end security policy and lacks uniform mechanisms for enforcement.
We describe DEFCon-Policy, a middleware that enforces security policy in multi-domain, event-driven applications. Event flow policy is expressed in a high-level language that specifies permitted flows between distributed software components. The middleware limits the interaction of components based on the policy and the data that components have observed. It achieves this by labelling data and assigning privileges to components. We evaluate DEFCon-Policy in a realistic medical scenario and demonstrate that it can provide global security guarantees without burdening application developers.

Keywords: multi-domain distributed applications, security policy, information flow control, event-based middleware

1 Introduction

Distributed systems that span multiple organisational or administrative domains are increasingly common in many areas, yet the associated security challenges remain largely unsolved. In the public sector from healthcare to public security, the integration of separate agencies and departments into federations enables the free flow of information and promises better services for citizens. Global companies are moving away from monolithic organisations towards a more dynamic business ecosystem that is reflected in the distributed nature of their software infrastructures. To achieve the necessary degree of integration between software components in such applications, they are often implemented as *event-driven architectures* [1], in which components, potentially belonging to different domains, process and exchange data in the form of event messages.

Multi-domain, event-driven applications process and exchange personal, often sensitive, data belonging to different users. As a result, organisations have

to abide by information handling policies, frequently stemming from data protection laws. Such policies often refer to the *flow of sensitive data* within the system. For example, the Department of Health in the UK stipulates that any access to a patient's electronic health record must be controlled by strict protocols.[4] *An open problem is how to encode and enforce such flow-based security policies in the context of multi-domain event-driven applications.* In particular, there is an impedance mismatch between high-level policies governing the handling of confidential data and low-level technical enforcement mechanisms.

Enforcing security policies in multi-domain, event-driven applications is challenging for several reasons: (1) due to the complexity and scale of applications, there is a risk that software faults may render policy checks ineffective. A single component that omits an access control check may reveal a confidential patient record to the outside world; (2) the integration of third-party components and libraries, often without source code auditing, may mean that necessary policy checks are omitted altogether. For example, a third-party developer may have a different interpretation of a security policy and decide incorrectly that revealing a patient record to an insurance provider is acceptable; (3) application deployments across multiple administrative domains introduce issues of trust and legal responsibility when enforcing security policies. For example, a hospital domain may trust an insurance provider with billing-related data but not with a complete patient health history. All these factors put the managed data at risk—security violations of organisations' private data may be disastrous; violations of third-party user data may make them liable to lawsuits.

Traditional event-based and message-oriented middleware[5] leaves the task of enforcing security policy to application developers. They have to include appropriate *policy enforcement points* (PEPs) in application components that carry out checks before executing sensitive operations. The middleware may provide support for the implementation of PEPs in the form of access control models such as access control lists and capabilities. However, these are low-level mechanisms that require the configuration of permissions at the granularity of individual operations. This makes it hard to realise a high-level security policy correctly, in particular, in distributed, multi-domain environments with different levels of trust between organisations.

To address the above security concerns, we argue that it should be possible to express high-level security policy as *constraints on permitted data flows* throughout a distributed, event-driven application. It should be the responsibility of the middleware to enforce such security policy uniformly across domains by taking a *data-centric view*—policy enforcement should occur automatically when data flows cross boundaries between components and domains, instead of carrying out access control checks over individual operations.

In this paper, we describe DEFCon-Policy, an event-based middleware that enforces event flow security policy in distributed, multi-domain applications. Flow policy is expressed in the DEFCon *Policy Language* (DPL) as high-

[4] See `http://www.nigb.nhs.uk/guarantee/2009-nhs-crg.pdf`

[5] For example, IBM's WebSphere MQ product: `http://www.ibm.com/webspheremq`

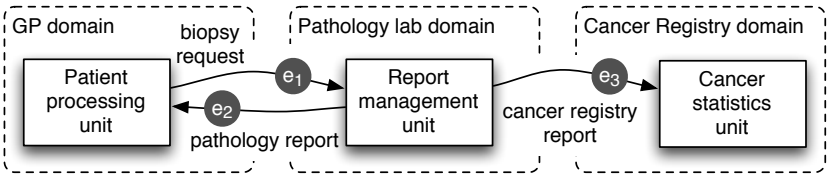

Fig. 1. Multi-domain healthcare scenario. Events are exchanged between processing units belonging to multiple domains.

level constraints on permitted data flows between components. The middleware takes a DPL specification and translates it into an equivalent assignment of *security labels*, which are associated with data flows, and *privileges*, which enable components to send and receive labelled data. To prevent event flows that would violate the security policy, DEFCon-Policy *sandboxes* components after they have received sensitive data controlled by the policy.

This approach makes policy enforcement transparent to application developers and does not add complexity to the application implementation. We evaluate DEFCon-Policy with a case study of a healthcare scenario with multiple distributed domains. Our results show that DEFCon-Policy can realise a complex, realistic security policy and enforce it with an acceptable runtime overhead.

In summary, the paper makes the following main contributions:

- a high-level event flow policy language for expressing constraints on distributed event flows in multi-domain, event-driven applications;
- the design and implementation of a middleware that can enforce event flow policy by monitoring event flows between application components and restricting them to ensure policy compliance;
- the evaluation of this approach as applied to security policy enforcement in the context of a realistic medical scenario.

The rest of the paper is organised as follows. In §2, we explore security problems in multi-domain, event-driven applications. We present DPL, our event flow policy language, in §3. In §4, we describe how the DEFCon-Policy middleware enforces event flow policy in a distributed setting by controlling the flow of events between distributed components. Evaluation results are presented in §5. The paper finishes with related work (§6) and future work and conclusions (§7).

2 Security in Multi-domain, Event-driven Applications

In this section, we describe the problem of providing security guarantees in distributed systems that involve multiple organisational domains. We introduce a sample healthcare workflow and show how event-based middleware can naturally support it but struggles to cover security challenges. Based on this analysis, we propose to enforce high-level security policy by the middleware.

Figure 1 gives an example of a data processing workflow in a healthcare scenario with multiple domains. It is representative of the types of workflows found

in the UK National Health Service (NHS). The figure shows domains (dashed regions) and software components within those domains. In this scenario, a patient is worried about a lump on his arm, and visits his general practitioner (GP) for a consultation. The GP takes a skin biopsy, which she sends, together with an electronic biopsy request containing information about the patient, to an NHS pathology laboratory for testing (edge e_1). There, a pathologist analyses the sample and produces an electronic pathology report. The lab sends this pathology report back to the GP (edge e_2), and also sends a copy to the regional cancer registry (edge e_3), but only if there is any evidence of cancer.

2.1 Event-based middleware.

A software system that supports the above workflow can be effectively realised as an *event-driven architecture* [1]. In this architectural model, data is presented as structured event messages or **events**. The software components of an application are implemented as a set of event processing **units** that receive, process and emit events. Events are usually transformed by a sequence of units. Units are hosted on physical machines and belong to a given organisational or administrative **domain**. An *event-based middleware* dispatches events between units, either locally or involving network communication between machines.

In the above scenario, the GP, the pathology lab and the cancer registry each form a domain. There are three types of events flowing between them: the biopsy request event e_1, the pathology report event e_2 and the cancer registry report event e_3. These events are exchanged between units belonging to the different domains: a patient processing unit in the GP domain, a report management unit in the pathology lab domain and a cancer statistics unit in the cancer registry.

2.2 Information security

The above workflow has events that contain confidential patient data, which makes information security important. Their propagation between the different parts of the system is regulated by corresponding data protection legislation.[6] It states that the following security guarantees must be maintained at all times:

1. Pathology reports may be sent only to the requesting GP or a cancer registry.
2. Cancer registries may only receive cancer-related pathology reports.
3. Only doctors in the pathology lab may view sensitive patient data.

In general, the security goals in such scenarios are to prevent *leakage of data* to unauthorised units or third-parties and to ensure the *integrity of data* that units use for input. For example, the pathology report event should not be sent to any other units outside of the GP and Cancer Registry domains. In addition, the GP domain should only accept genuine pathology reports as input.

[6] `http://www.nhs.uk/choiceintheNHS/Rightsandpledges/NHSConstitution/Documents/COI_NHSConstitutionWEB2010.pdf`

In terms of a threat model, information security can be violated in a number of ways: software bugs in the implementation of units can leak sensitive events or make units accept bogus input data; malicious developers can include back-doors in unit implementations to obtain unauthorised access to data; and units operating in different domains may handle security policy differently due to inconsistent interpretations at implementation time. All of these problems are enabled by the fact that security concerns are distributed across the implementation of many units and are disconnected from the global security requirements. Any unit in the system can potentially violate information security.

Security policy. To ensure that a multi-domain, event-based application guarantees information security, we argue that a policy administrator should first express security concerns in a high-level policy language. By separating security policy from unit implementations, the policy administrator can focus on the high-level security goals of a multi-domain application, without being overwhelmed by implementation details.

A key observation when expressing security policy is that the required security guarantees, as the ones described above, usually pertain to the event data and, more specifically, focus on the flow of events through the system. An *event flow security policy* should therefore control the propagation of event flows through the system. This is in contrast to fine-grained security policy found in access control systems that usually governs permitted operations. For example, when the biopsy request event is received by the report management unit, it is trusted to manage the data appropriately. Any operations carried out by the report management unit internally do not need to be checked, as long as interactions of the unit with other units are controlled.

Policy enforcement. Current implementations of multi-domain, event-driven applications leave the overall enforcement of security policy to the developers of units. A frequent approach is to introduce an access control layer around units, which carries out ad hoc policy checks at the input and output of events. This is not only error-prone but also makes it challenging to enforce security properties that rely on the behaviour of a sequence of event processing units.

In contrast, we want to enforce event flow policy by the middleware itself, independently of the implementation of processing units. For this, the middleware must track the flow of events between components in order to provide end-to-end security guarantees that do not depend on the correct implementation of each individual unit. This assumes that the middleware implementation is correct and can be trusted to enforce event flow policy. In practice, this is a reasonable assumption because only a small part of the middleware implementation is involved with policy enforcement.

Information flow control. Since event flow policy expresses limitations on the flows of events throughout the system, the middleware must be able to prevent invalid flows. This idea of *information flow control* has been successfully applied in different domains for achieving security guarantees, including operating systems [2, 3], programming languages [4], web applications [5, 6] and high-performance event processing [7].

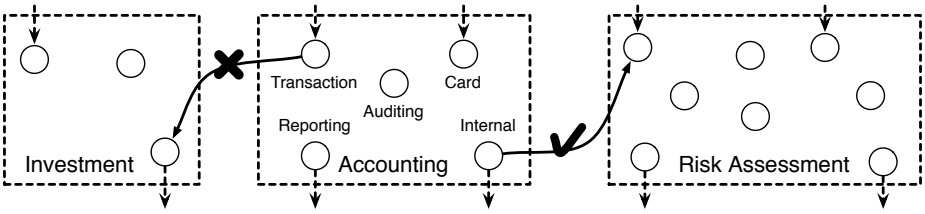

Fig. 2. Protection of flow categories within a bank. Flow categories define boundaries around data and specify which units can pass data over those boundaries. `Transaction`, as an input unit, cannot send data to the investment flow, while `Internal`, an output unit, can extract data from the flow.

In previous work, we proposed a Decentralised Event Flow Control (DEFC) model [7] for controlling the flow of events in event-driven applications. DEFC associates events with **labels**, that "contaminate" units that receive them: data output by a contaminated unit must include the labels of the events that contaminated it. As detailed in §4.2, this mechanism is key for implementing mandatory tracking of security properties for data processed in the system. Units may bypass constraints imposed by labels associated with events only when they possess **privileges** over them. As we show in §4, the DEFC model provides an appropriate low-level enforcement mechanism for event flow policy. However, it must be extended to support multiple domains that may enforce policies differently.

3 Event Flow Policy

In this section, we introduce the DEFCon Policy Language (DPL), our language for event flow policy specification. Based on the previous analysis of security in multi-domain, event-based applications, the design of DPL aims to satisfy the following set of requirements:

- Security policies should take a data-centric view, providing end-to-end guarantees for confidentiality and integrity of event flows in the system.
- Security policies should be independent of the functional implementation of processing units and be supported across legacy processing units.
- Security policies should be separate from the details of the enforcement mechanism at the middleware level.
- Security policies should be enforceable efficiently, without resulting in an unacceptable degradation of event processing performance.

We describe DPL with reference to a financial scenario, as illustrated in Figure 2. Within the processing system of a bank, several functions exist: from investment activities to accounting on behalf of clients to internal risk assessment. Flows of information are exchanged within each function and among different functions. We consider the case of a bank that wants to improve the security of the software component that processes customer account information. The goal is to ensure that account information cannot be corrupted or leaked by software faults or malicious behaviour of components.

The events in this scenario can be divided into *event flow categories*. An event flow category, such as `accounting_flow`, is used to identify events with distinct security requirements, for example, by pertaining to data containing customers' account details. Alternatively, a flow category could group all data belonging to a single user.

3.1 Event flow constraints

Our event flow policy provides security guarantees through the definition of *event flow constraints* on flow categories. We focus on two ways that policy specification can distinguish flows of information by applying flow constraints and we name them *vertical* and *horizontal* flow separation. Vertical separation relates to flow constraints that should hold across the end-to-end processing of events, from input to output. Horizontal separation is used to isolate the processing at one stage from the processing being done in another, and is typically used to achieve security guarantees related to functional transformations such as data cleaning, auditing and anonymisation.

In DPL, event flows constraints have the following syntax:

flow_constraint ::= ⟨*flow_name*⟩ `':' '{'` flow_part (`','` flow_part)* `'}' '.'`
flow_part ::= [`'->'`] ⟨*processing_context*⟩ [`'->'`]

As an example, the following DPL specification encodes the flow constraints in the above banking scenario:

```
accounting_flow: {
  -> transaction, -> card,
  auditing,
  reporting ->, internal -> }.
```

All flow constraints must name an event flow category (i.e. `accounting_flow` above), and state whether the *flow parts* (i.e. `card`, `transaction`, etc.) can receive or produce events within the flow. Flow parts indicate the processing context, which can be a unit that sees and alters the event flow.[7] The inclusion of a unit as a flow part, without any further annotation, means that the unit is *sandboxed* within the context of the specified event flow. In other words, such units are isolated so that they can only input and output events from and to other units within that event flow.

Parts of event flows contain annotations to indicate that they are able to cause events to flow in or out of the event flow. Input units, preceded by a `->` prefix, such as `transaction` and `card` are constrained to output events only within the flow, but can receive events from the outside of the flow. Units with a `->` suffix, such as `reporting` can, in addition to sending events into the flow, take events from the flow and let them leave the protected environment created by the flow constraint. Units can also both input and output events to and from

[7] To simplify discussion, we assume for now that a processing context is a single unit; we relax this assumption in §3.2, when we address the general case.

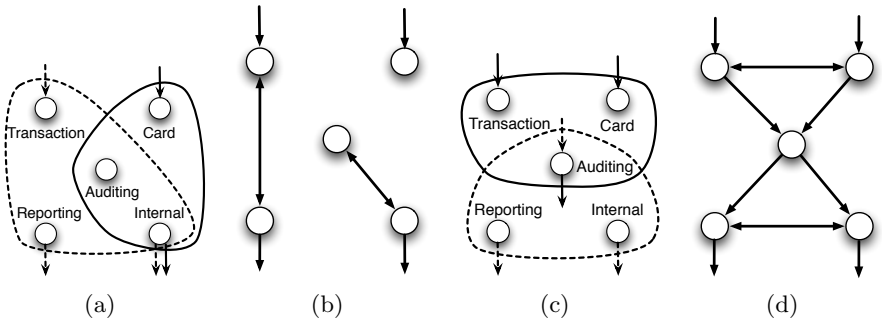

Fig. 3. Vertical and horizontal separation. In 3(a), two constraints are defined to separate the data of the transaction from the details of the credit card; flows permitted by both constraints are shown in 3(b). In 3(c), horizontal separation is used to force events to pass through an `Auditing` unit as shown in 3(d).

a flow. Note that input and output units have a certain degree of freedom in their actions: input units can choose to receive input events from "inside" the flow, "outside" the flow or both; similarly output units can specify where to send new produced events with respect to the flow constraint.

Specification of flow constraints protects both the confidentiality and the integrity of event flows. For example, a policy administrator can ensure that a flow prevents units written by third parties from tampering with accounting data (integrity), or even receiving it in the first place (confidentiality). A sand-boxed unit cannot leak events, or allow unauthorised modifications. As a consequence, the amount of code to be trusted is limited for integrity to the units that are flow inputs and for confidentiality to the units that are flow outputs.

Vertical flow separation. Flow categories isolate and control the diffusion of events with different security requirements. For example, the accounting data above may contain credit card information that should be prevented from reaching the reporting unit. In DPL, this can be expressed by defining two constraints:

```
transaction_flow:
   { -> transaction,
   auditing,
   reporting ->, internal -> }.
```

```
card_flow:
   { -> card,
   auditing,
   internal -> }.
```

The personal transaction flow and the credit card flow intersect as part of the `auditing` and `internal` units. When two flows intersect, we have some events that are part of only the first flow, some events that are only part of the second flow and some events that are part of both flows. A unit that is present in both flows (e.g. the `auditing` unit in our example) can only receive events that are accepted in both flows by their respective flow constraints. This means that events in the intersection of the two flows can be created only by units that act as inputs for both flows, or units in one flow that are also inputs for the other flow. As this never happens for personal and card flows, the two units in the intersection are effectively isolated from input events. This can be illustrated

using the possible interactions between units shown in 3(b). DPL specifications can be checked for such inconsistencies by the middleware, alerting the policy administrator to such problems.

There are two ways of fixing the above issue. The first is to restructure the system to split a unit into two units e.g. in the auditing case, one unit to monitor for suspicious card numbers, the other for suspicious transactions above a given threshold. An alternative solution is to add `auditing` and `internal` as input units to `card_flow` and `transaction_flow`. This is acceptable as card data entering from the `card` unit would still be constrained to flow only to the `internal` unit. The weakening of the flow constraints would simply allow `auditing` and `internal` to receive additional events as input.

Horizontal flow separation. So far, we have explored the case, in which flows are defined to protect data of a given security category from other categories (vertical separation). There is, however, another use for flow constraints: to constrain the processing within a specific flow (horizontal separation). For example, in our accounting flow, the policy administrator may want to ensure that all transactions and card usage are audited. We can enforce this by separating the auditing flow horizontally into two subflows:

```
unaudited_flow: {                audited_flow: {
   -> transaction, -> card,         -> auditing,
   auditing -> }.                   reporting ->, internal -> }.
```

The two flows intersect each other again (see Figure 3(c)): the `auditing` unit is common to both event flows. However, the intersection does not cause problems for these two flows (see Figure 3(d)). The case when outputs of one flow are inputs for the other is actually beneficial: `auditing` is an output of the "top stage" of Figure 3(c), it can thus only receive events from the top stage. In general, it could output without constraint, except that being an input to the bottom stage means that it can only present its events to the bottom event flow. Thus, a protected data transfer is forced from one flow category to the next.

Parameterisation. Some policies require separation vertically of many flow categories with the same structure, e.g. to protect data individually by client or patient. To support such constraints, DPL allows *parameterisation* of flow constraints, supporting the inclusion of parameters that appear both in flow categories and in processing units. For example, the above `transaction_flow` can be parameterised by client to prevent transactions from one client to affect a report for another client:

```
transaction_flow[client]: {
   -> transaction[client],
   auditing[client],
   reporting[client] ->, internal[client] -> }.
```

3.2 Abstracting processing context

So far, we illustrated the use of DPL in small-scale contexts. We assumed that policy administrators had global knowledge of all processing units that participate in event flows. In such scenarios, it is possible to link policy fragments in the

form of flow constraints directly to the units that are constrained by these fragments. However, such an approach is not feasible in larger deployments where the policy spans multiple domains and no domain has control over the details of event processing in other domains. For example in our accounting scenario, including units from a third party would tie in the policy with the units' design: changes to the design would require changes to the event flow specification.

To apply event flow control in a large distributed setting, it is therefore necessary to abstract the relationship between flow policies and units. We achieve this by introducing hierarchical names that correspond to *event processing contexts*. The hierarchical nature of processing contexts facilitates support for multi-domain use of event flow policy: we can map the organisational structure of domains to the hierarchical structure of processing contexts. Also, by using a federated naming service analogous to the domain name system (DNS), the control over subcontexts can be delegated to the domains themselves.

Processing context names provide a common, consistent naming structure to correlate processing units and policies belonging to different organisations. Flow constraints can refer to processing context names, which then map to actual event processing units. This relaxes the previous assumption in §3.1 that each unit maps to exactly one processing context. When a flow constraint states a processing context, the constraint applies to all units that are directly part of the context and to all units that are part of any sub-contexts.

We illustrate processing contexts with two examples. As the first example, we refine the flows that are internal to the previously introduced `reporting` context from Figure 2 by specifying the following flow constraint:

```
anon_reporting_flow: {
   -> reporting.anonymiser,
   reporting.stats -> }.
```

This flow names two sub-contexts of `reporting`: an `anonymiser` and `stats`. The `stats` context is reachable only through the `anonymiser`. All units assigned to `reporting.stats` can only receive data from `reporting.anonymiser`, while units in `anonymiser` or directly in `reporting` are still constrained by any flow constraint mentioning the `reporting` context. As a multi-domain example, we can consider the following version of the `accounting_flow`:

```
policy uk.co.ebank
accounting_flow: {
  -> transaction, -> .uk.co.curr_quotes.ebank,
   local_processing,
   internal ->, .uk.gov.soca.auditing.ebank -> }.
```

In this policy, processing contexts not starting with a dot are treated as relative to the domain specified in the policy header, while fully-qualified names can refer to arbitrary contexts. This example has an external provider of quotes for foreign currencies and the UK Serious Organised Crime Agency (SOCA) as an output context. The two organisations with control over these processing context names define processing units operating in these domains, and authorise foreign organisation (`ebank` in this case) to define policies relating to these contexts.

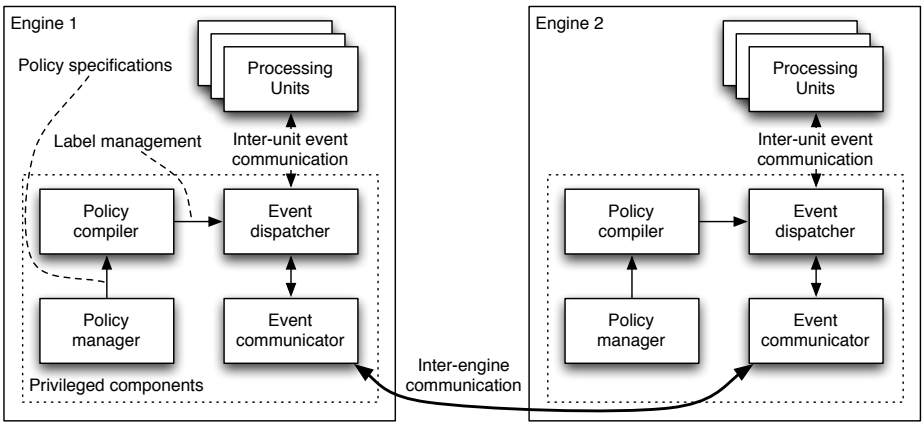

Fig. 4. DEFCon-Policy architecture. Multiple engines house processing units that communicate using message passing while information flow is tracked using labels.

4 Distributed Event Flow Middleware

Given a set of policies described in DPL, we enforce them using the DEFCon-Policy middleware. It implements distributed DPL policies by translating them into local communication constraints on processing contexts and enforces them as units execute. The architecture of the DEFCon-Policy middleware is shown in Figure 4 and consists of the following parts:

Engines. Engines are responsible for policy enforcement in one or more processing contexts. Each engine hosts processing units and isolates them from each other to be able to sandbox units. Engines also control communication channels to and from other engines and outside systems. Engines manage the internal flow of events using the DEFC security model (cf. §4.2).

Event dispatcher. The event dispatcher supports asynchronous communication between units in the form of *publish/subscribe* communication. Publish/subscribe allows units to express their interest in events that match a subscription filter. Events are dispatched in compliance with security labels.

Policy manager. As part of each engine, a policy manager is responsible for locally checking and authorising DPL policies constraining processing contexts local to that engine. In a small-scale deployment, policies can be checked and deployed manually; in more complex deployments, the policy managers of different engines coordinate to set-up policies (cf. §4.1).

Policy compiler. The policy compiler translates DPL policies into security labels and privileges in the DEFC model used for enforcement (cf. §4.1).

Event communicator. In each engine, the event communicator is responsible for securely propagating protected events between engines. It guarantees that events are labelled correctly in each engine trusted by the policy when they are exported and imported over the network (cf. §4.2).

4.1 Distributed policy management

To support large multi-domain deployments, DEFCon-Policy needs to handle many processing contexts deployed in many engines. In such a scenario, policy set-up and management needs support from the middleware. To set up a policy, before enforcement can begin, DEFCon-Policy performs a series of steps:

1. Context to engine resolution. After a new DPL policy has been submitted, the DEFCon-Policy middleware first resolves engines responsible for processing contexts mentioned in the policy, thus locating the deployed units to constrain. The resolution from processing contexts to engines is performed through a distributed directory system. Such a directory service can be federated so that each organisation owns a part of the namespace and can delegate subparts to other organisations.

2. Engine trust verification. Engines have to verify that remote engines involved in a policy can be trusted to enforce event flow constraints defined in the policy. This is important because remote engines may belong to independent administrative domains. For example `uk.co.ebank.transactions` from §3.2 might map to a local engine `defcon.ebank.co.uk`, while `uk.co.curr_quotes.ebank` might map to engine `defcon.curr_quotes.co.uk` externally hosted.

In the most general case, each domain, such as `ebank`, can specify the set of DEFCon-Policy engines that it trusts for enforcement of its policies. These can be specified per organisation, per policy or per flow. We assume that units are deployed on engines with sufficient trust, such as the local engine, to support their execution. Referring to processing contexts by a fully-qualified name is an assertion of trust in the remote policy enforcement of that domain.

3. Policy deployment and authorisation. Once engines are verified, the policy is deployed on all relevant engines. The policy managers on each engine check if the deployed policy is authorised with respect to the contexts involved. Such authorisation may be implemented by using PKI infrastructure[8] for example. Digitally signed policies, and information about the signing certificates, can be integrated with the directory service exploited in step 1.

4. Policy checking. Before a policy is enforced, DEFCon-Policy checks that the new policy is not inconsistent (cf. vertical separation example in §3). An inconsistent policy may violate liveness properties by leading to units that are unable to receive or send any events because of policy constraints.

To check the policy, the policy manager recursively retrieves policies related to the processing contexts specified in the new policy. It then performs a graph traversal to check if, for all units, there exists at least one event flow path from the external world to their input (reachability) and at least one path from their output to the external world (observability).

This policy checking algorithm can be formalised as follows. Let P be the set of all possible processing contexts. The goal of the algorithm is to check the compatibility of a set of flow constraints F where $F \subseteq 2^P \times 2^P \times 2^P$. We can

[8] SPKI would meet our needs: `http://www.ietf.org/rfc/rfc2693.txt`

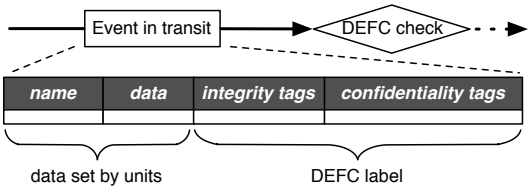

Fig. 5. An event with a DEFC label. DEFC labels are not controlled by units, but are used to enforce event flows.

first compute whether two units can send an event to each other according to F:

$$\begin{aligned}\text{canSendTo}(x, y) \Leftrightarrow \forall(F_{\text{in}}, F_{\text{sand}}, F_{\text{out}}) \in F : F_{\text{all}} = F_{\text{in}} \cup F_{\text{sand}} \cup F_{\text{out}} \\ \Big(x \in (F_{\text{in}} \cup F_{\text{sand}}) \Rightarrow y \in (F_{\text{sand}} \cup F_{\text{out}})\Big) \wedge \\ \Big(\big(x \in F_{\text{out}} \vee x \notin F_{\text{all}}\big) \Rightarrow \big(y \in F_{\text{in}} \vee y \notin F_{\text{all}}\big)\Big)\end{aligned}$$

and based on that, infer reachability via multiple hops:

$$\begin{aligned}\text{canReach}(x, y) \Leftrightarrow \exists\, n > 0, \forall i \in [1, \ldots, 2n], z_i \in P \\ \wedge\, (z_1 = x \wedge z_{2n} = y) \wedge \text{canSendTo}(z_i, z_{i+1})\end{aligned}$$

Then we consider a unit that is external to all flow constraints of F, i.e. operates in an unconstrained context, modelling the external world ϕ. Given $\phi \in P : \forall(F_{\text{in}}, F_{\text{sand}}, F_{\text{out}}) \in F, \phi \notin F_{\text{in}} \cup F_{\text{sand}} \cup F_{\text{out}}$, the following definitions hold:

$$\begin{aligned}\text{isObservable}(x) &\Leftrightarrow \text{canReach}(x, \phi) \\ \text{isReachable}(x) &\Leftrightarrow \text{canReach}(\phi, x)\end{aligned}$$

The check succeeds if and only if, given F,

$$\forall x \in P : \text{isObservable}(x) \wedge \text{isReachable}(x)$$

at which point the policy is ready for enforcement.

4.2 Enforcement of event flow constraints

After policies have been distributed to engines, the DEFCon-Policy middleware sets up runtime enforcement within an engine and between engines.

DEFC model. Event flow constraints specified in DPL are enforced at runtime according to the Decentralised Event Flow Control (DEFC) model [7]. In this model, as shown in Figure 5, events are structured messages that consist of (1) a named *data* part that units can manipulate and (2) a *DEFC label*. The data part contains the payload of the event whereas the DEFC label restricts its flow.[9]

A DEFC label (S, I) is composed of a *confidentiality label* (S) and an *integrity label* (I). Labels are sets of *tags*, each representing a concern over confidentiality or integrity of the event. A tag is implemented as a unique bit sequence.

[9] We ignore multiple data parts here; see [7] for more detail on the DEFC model.

Processing units are also assigned a label L_p that represents the confidentiality and integrity of information contained in their state. Units can create events or process events that they receive, provided that their label can flow to the labels of the events. Intuitively, a unit cannot read data that is "more confidential" than its label, or write data that has "higher integrity" than itself. More precisely, an event flow is only allowed if the source label L_s and the destination labels L_d satisfy a partial order *can-flow-to* relation $\preceq$:

$$L_s = (S_s, I_s) \preceq L_d = (S_d, I_d) \iff S_s \subseteq S_d \wedge I_s \supseteq I_d.$$

Units possess *privileges* that allow them to change labels. Adding a tag t to a label requires the t^+ privilege, which for integrity is called an "endorsement privilege"—it endorses the unit's state, allowing it to produce higher integrity data. Removing t from a confidentiality label requires the t^- privilege, called a "declassification privilege"—it declassifies the unit's state allowing it to produce unclassified data. Privileges held by units also determine if a unit can communicate externally. Only a unit that holds endorsement privileges t^+ for all tags in I and declassification privileges t^- for all tags in S can freely exchange events with the outside world.

DEFC also controls the *delegation* of privileges between units. Only if a unit possess the *privilege-granting* privileges t^+_{auth} and t^-_{auth}, it is permitted to delegate its endorsement and declassification privileges t^+ and t^- to other units.

Policy translation to DEFC. The DEFC model is used to enforce event flow constraints specified in DPL policy specifications. Each flow constraint f is associated with a tag pair (c_f, i_f) to protect flow confidentiality and integrity. Tags and privileges are assigned to units in the following way:

(1) *Sandboxed and output units* have i_f in their integrity label, constraining them to receive only events that also contain i_f. (2) *Sandboxed and input* units also have c_f in their confidentiality label, thus being constrained to have their output contain c_f. (3) *Input units* are given i^+_f and therefore can produce events with i_f, even if they do not have i_f in their label. This means that they can receive events form "outside of the flow". (4) *Output units* are given c^-_f to produce data without c_f in their label, even if their confidentiality label contains c_f to be able to receive data from the flow. When a unit is mentioned in multiple flow constraints, its label is the conjunction of all listed constraints.

In the multi-domain example from §3.2, the policy manager on the engine `defcon.ebank.co.uk`, responsible for the `transactions`, `local_processing` and `internal` subdomains of `uk.co.ebank`, would create i, c tags to represent the flow `uk.co.ebank.account_flow`. The manager would then instantiate the processing unit parts of those contexts with a label $L = (\{c\}, \{i\})$, it would also bestow i^+ in `transactions`, and c^- in `internal`, keeping i^+_{auth} and c^-_{auth} for itself. These privileges can be used by the policy manager in case of a policy change that would require a reconfiguration of privileges.

Inter-engine communication. Tags and privileges that are allocated by policy managers in engines have local meaning. Engines use these to restrict communication between units in the local engine. However, units that are able to process

data of a given flow should be able to exchange events even if they are located on different engines. This requires the exchange of events between two engines over the network. It cannot be achieved by just giving units endorsement or declassification privileges because this would enable them to communicate with units outside of the flow, without event flow control.

To address this problem, DEFCon-Policy provides a trusted proxy unit, called an *event communicator*. The event communicator is delegated endorsement and declassification privileges for a given event flow and can then transfer events to the event communicator in another engine. As part of this process, the tags associated with events are translated at the receiver's engine by the event communicator to equivalent tags for local enforcement. This mapping between tags on different engines is set up on demand, on the basis of the policy specifications shared between the engines during policy deployment (§4.1).

We illustrate inter-engine communication in the context of the example above. The policy manager on the engine `defcon.curr_quotes.co.uk`, which is responsible for the `uk.co.curr_quotes.ebank` context, can allocate tags i, c to represent the `uk.co.ebank.account_flow` flow. It initialises units in the context to $L = (\{c\}, \{i\})$, granting the i^+ privilege to them. The units from the `uk.co.curr_quotes.ebank` context cannot communicate directly with units in `uk.co.ebank.local_processing`, which are sandboxed in the accounting flow.

Each policy manager instantiates an event communicator, granting it the i^+ and c^- privileges. When a unit in the `uk.co.curr_quotes.ebank` context sends an event to `uk.co.ebank.local_processing`, the event is received by the communicator, which exports it from the accounting flow by exercising its declassification privilege. The event is then transmitted securely, for example, using an encrypted transport-level connection, to the other event communicator. The second event communicator possesses the i^+ privilege, which enables it to insert the received event in the accounting flow in the other engine.

5 Evaluation

We evaluate the effectiveness of DPL and DEFCon-Policy with respect to specifying and enforcing security policy in a multi-domain, event-driven application. We focus on the ease-of-use from a software developer's standpoint and also experimentally evaluate the performance impact of the middleware.

5.1 Healthcare case study

In our case study, we examined an NHS policy involving GPs, Pathology Laboratories, a Primary Care Trust (PCT), the UK Office of National Statistics (ONS) and a Cancer Registry. Figure 6 shows an extract of the policy that enforces the guarantees introduced in §2.

An overarching NHS policy (lines 1–3) specifies a high-level constraint that sensitive data are controlled and partitioned by GPs. Partitioning is enforced by the use of a constraint parametrised by `gp`. Data processing in the laboratory is

```
1  policy uk.nhs
2  sensitive[gp]: { -> GP[gp].sensitive ->, -> lab.doc[gp] ->,
3  lab.sensitive[gp], cancer_registry.sensitive -> }. {...}
4
5  policy uk.nhs.GP[gp]
6  patient_data_flow: { -> sensitive.patient_data,
7     sensitive.patient_data.anonymiser ->,
8     sensitive.pathology.patient_data ->,
9     -> sensitive.pathology.incoming_reports }.
10 anonymised_data: { -> sensitive.patient_data.anonymiser,
11    statistics.anonymised_data -> ,
12    performance.anonymised_data -> }.
13 path_request: { -> sensitive.pathology.test_requests,
14    -> sensitive.pathology.patient_data,
15    .uk.nhs.lab.doc[gp].pathology.request ->,
16    .uk.nhs.lab.sensitive[gp].pathology.patient_data -> }. {...}
17
18 policy uk.nhs.lab
19 path_report[gp]: { -> doc[gp].pathology.report,
20    -> sensitive[gp].pathology.patient_data,
21    sensitive[gp].pathology.cancer_registry_reporting ->,
22    .uk.nhs.GP[gp].sensitive.pathology.incoming_reports -> }.
23 tumour_report: { -> sensitive[gp].pathology.cancer_registry_reporting,
24    .uk.nhs.cancer_registry.sensitive.pathology.incoming -> }. {...}
```

Fig. 6. Extract of the healthcare policy scenario in DPL. Constraints not related to managemet of sensitive medical data are omitted.

also partitioned by GP and only doctors within the lab can see and contribute to confidential information (lines 2–3). Finally, Cancer Registries can receive sensitive information for computing statistics about tumours (line 3).

GPs specify their own local policy (lines 5–16) to refine and extend the global policy. In this example, all GPs have the same policy: patient data can be transformed by an anonymiser into anonymised data (line 7), which in turn can be used for computing statistics by the ONS and measuring performance by the PCT (lines 10–12). Alternatively, patient data can be used to generate pathology requests that are to be sent to a lab (lines 8 and 13–16), while reports received from the lab can be combined with patient data (line 9). The lab pathology reports can either be sent back to the same GP (line 22) or included in tumour reports (lines 23–24) by specific reporting units (line 21).

Policy enforcement in DEFC. After the policy is specified in DPL, it is compiled into tags that are used for DEFC enforcement. We now show how the resulting tag assignment enforces the guarantees presented in §2. Each flow constraint is enforced by a tag pair, which we represent symbolically as (i_x, c_x) where x is the line at which the constraint is defined in Figure 6. Pathology reports are produced within the `lab.doc[gp].pathology.report` context, specific to each *GP*, that is tainted by c_{19}^{gp}. The declassification privilege for this tag, c_{19}^{gp+}, is held by the cor-

responding `GP[gp].sensitive.pathology.incoming_report` context and not by any other context under a different GP. The only other context with declassification privilege for the report tag is `lab.sensitive[gp].pathology.cancer_registry_reporting`. It is, however, tainted by tag c_{23}, which can be removed only by `cancer_registry.pathology.incoming`. This completes the enforcement of the first guarantee protecting sensitive pathology reports.

As the `cancer_registry.pathology.incoming` context is tainted by i_{23}, `lab.sensitive[gp].pathology.cancer_registry_reporting`, holding the i_{23}^{+} privilege, is the only context that can send data to it. Furthermore, as units in this context drop reports not classified as cancerous, the second guarantee on Cancer Registry input is enforced.

To enforce the third guarantee, the `GP[gp].sensitive.patient_data` context is protected by c_2^{gp} and c_6^{gp}. Only units under `lab.doc[gp].pathology.request` can reveal sensitive data to authenticated doctors within the lab because, as a sub-context of `GP[gp]`, they have the c_2^{gp-} privilege. While these units do not hold privileges for c_6^{gp}, units in `GP[gp].sensitive.pathology.patient_data` can exchange c_6^{gp} with c_{13}^{gp} for which units in `lab.doc[gp].pathology.request` have the c_{13}^{gp-} privilege.

The policy fragment consists of 24 lines. It generates $10n + 2$ tags and distributes $37n + 1$ privileges where n is the total number of GPs in the system. Assuming that one unit is instantiated in every context, at least $14n + 2$ units must be initialised with correct taints. To provide this initial set-up manually, a programmer would have to call the low level DEFC API at least $24n + 4$ times. Instead, these calls, the creation of tags and the distribution of privileges are automatically carried out by DEFCon-Policy.

5.2 Performance overhead

In this section, we present an experimental evaluation of the performance impact of enforcing event flow policy using DEFCon-Policy. We measure overhead as a micro-benchmark in terms of (1) the end-to-end event propagation latency between a set of units and (2) the throughput of event processing. For these experiments, we deploy the following simple security policy:

```
policy secure_Policy
sensitive_data: { -> context_a ->, context_b }
```

This policy specifies that only units in `context_a` can cause events to flow in or out of the `sensitive_data` flow. Units in `context_b` can perceive and process such events without the ability to disclose them. A single unit A and a single unit B are instantiated in each context, respectively. We compare processing latency and throughput while varying the following parameters:

1. **Number of engines.** The units/contexts are deployed in a single engine or in two different engines.
2. **Network encryption.** When network communication is involved, Transport Layer Security (TLS) can be used to encrypt data.

Configuration			Throughput	Penalty	Latency	Penalty
Engines	TLS	Policy	(Events/sec)		(ms)	
1	n/a	✗	99,723	–	0.028	–
1	n/a	✓	82,334	17.4%	0.030	7.1%
2	✗	✗	62,215	–	0.268	–
2	✓	✗	42,344	31.9%	0.283	5.6%
2	✓	✓	37,500	39.7%	0.294	9.7%

Table 1. Performance overhead of DEFCon-Policy middleware.

3. Policy enforcement. The engines enforce that events are propagated according to `secure_policy`.

Our experiments are conducted on two Intel Core 2 Duo E6850 3 GHz machines with a maximum of 1 GiB of heap memory allocated per engine. We use Sun's unmodified JVM 1.6.0.06 on Ubuntu 8.04. The average network round trip-time between the machines is 0.18 ms. Each event contains a single integer.

Table 1 shows the average throughput and the 95^{th} percentile of latency for events sent from unit A to unit B and back to A. As this experiment does not involve actual event processing, it mainly stresses the event dispatching mechanism. In the single-engine configurations, DEFCon-Policy enforcement introduces an overhead of 17.4% for throughput and 7.1% for latency. This is the result of storing, propagating and checking tags at runtime.

The overall lower performance achieved in the two-engine configurations is a consequence of the work carried out by the event communicators. Throughput is reduced by 31.9% due to network encryption. On top of this, DEFCon-Policy enforcement introduces a further relative overhead of only 11.4% for throughput and 3.9% for latency. We believe that the overhead of policy enforcement becomes even more marginal for realistic applications with more costly processing.[10]

6 Related Work

Middleware. Messaging middleware, and event-based middleware in particular, such as Sun JMS or IBM WebSphere support efficient exchange of information in large-scale distributed systems. Security in these systems usually focuses on access control at the boundary of the middleware API rather than end-to-end tracking of information. Any component with access to multiple channels can transfer information between them. As such each component needs to be trusted to comply with integrity and confidentiality requirements of messages.

Policy. Most approaches to policy specification focus on actions (i.e. privileges) rather than data, e.g. access control lists and role-based access control. Higher level firewall policy languages [8] facilitate the definition of rules for "allow/deny"

[10] Note that Sun's JVM does not fully enforce unit isolation; the overhead imposed to achieve such isolation was the focus of previous research [7].

actions, but such policies are only enforced locally. To achieve end-to-end security, policies need to be attached to data (i.e. "sticky policies" [9]). A survey and taxonomy of enforcement of sticky policies through distributed systems is provided in [10]. In contrast, our work is a contribution regarding the use of a high-level policy language with a view to translation into distributed, low-level enforcement with security labels.

Information flow control (IFC) originated in the military domain in the setting of Multi-Level Security (MLS) systems, and in that context used a limited number of centrally-defined security labels. Declassification of information was dealt with outside of the model. Myers and Liskov [4] extended IFC to decentralised enforcement allowing unprivileged principals to define and share labels and privilege over those new labels dynamically. More recently OS-level DIFC proposals [2, 3, 11] protect OS processes and resources by using dynamic labels that can be created at runtime. DEFC [7] brings tag-based security to event processing systems, by allowing the labelling of event parts and assigning labels to processing components.

In the past, decentralised IFC has mainly been applied to processes within a single machine. An exception is DStar [12], which automates translation between tags in remote enforcement engines. However, DStar aims to scale to a limited number of machines, e.g. multi-tiered web applications. In contrast, the focus of our work are large-scale distributed applications that contain engines under control of independent administrative domains.

Creating a policy language for decentralised IFC has been explored in Asbestos [13]. They compute tag configurations from pairwise communication patterns between sets of processes. In contrast, DPL supports policies independently authored by multiple policy administrators in the context of multi-domain distributed applications and explicitly addresses policy compatibility checking, policy authorisation and distributed enforcement.

7 Conclusions

Our research is motivated with reference to use cases in complex, multi-domain scenarios found in electronic healthcare and financial services. We have presented DEFCon-Policy, a middleware that achieves end-to-end enforcement of distributed event flow control based on high-level policy. The benefits of strict, mandatory access control are coupled with the expressiveness and independence required by policy specification within multi-domain, distributed systems. We provide details of DPL, our event flow policy language, and sketch its formal semantics. We detail the way in which event flow policies are compiled down to be enforced using a distributed event flow control model. The evaluation of our prototype demonstrates that in both single node and distributed cases, an acceptably low overhead is incurred, while benefitting from the end-to-end, event-based security features.

In future work, we want to explore the interaction of programming languages and flow-based policy enforcement. By integrating flow constraints with pro-

gramming paradigms, we can make it more natural for programmers to remain compliant with flow constraints. In addition, we want to determine the potential for interconnection of our policy and enforcement systems with existing parameterised, role-based access control infrastructures. Finally, we will acquire further experience of using DEFCon-Policy in real-world policy environments. This will allow us to judge better the proportion of common policy requirements that are covered by DEFCon-Policy.

Acknowledgements

This work was supported by grants EP/F042469 and EP/F044216 ("SmartFlow: Extendable Event-Based Middleware") from the UK Engineering and Physical Sciences Research Council (EPSRC).

References

1. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley (2002)
2. Efstathopoulos, P., Krohn, M., VanDeBogart, S., et al.: Labels and event processes in the Asbestos Operating System. In: SOSP '05, ACM (2005) 17–30
3. Zeldovich, N., Kohler, E., et al.: Making information flow explicit in HiStar. In: OSDI '06, Berkeley, CA, USA (2006) 263–278
4. Myers, A., Liskov, B.: Protecting privacy using the decentralized label model. ACM Transactions on Software Engineering and Methodology **9**(4) (2000) 410–442
5. Chong, S., Vikram, K., Myers, A.: SIF: Enforcing confidentiality and integrity in web applications. In: USENIX Security Symposium, Berkeley, CA (2007) 1–16
6. Papagiannis, I., Migliavacca, M., Eyers, D.M., Shand, B., Bacon, J., Pietzuch, P.: Enforcing user privacy in web applications using Erlang. In: Web 2.0 Security and Privacy (W2SP), Oakland, CA, USA, IEEE (2010)
7. Miglivacca, M., Papagiannis, I., Eyers, D., Shand, B., Bacon, J., Pietzuch, P.: High-performance event processing with information security. In: USENIX Annual Technical Conference, Boston, MA, USA (2010) 1–15
8. Bandara, A., Kakas, A., Lupu, E., Russo, A.: Using argumentation logic for firewall policy specification and analysis. In: Distributed Systems: Operations and Management (DSOM), Dublin, Ireland (2006) 185–196
9. Mont, M.C., Pearson, S., Bramhall, P.: Towards accountable management of identity and privacy: Sticky policies and enforceable tracing services. In: Database and Expert Systems Applications (DEXA), Washington, DC, USA (2003) 377–382
10. Chadwick, D.W., Lievens, S.F.: Enforcing "sticky" security policies throughout a distributed application. In: Middleware Security (MidSec), New York, NY, USA, ACM (2008) 1–6
11. Krohn, M., Yip, A., Brodsky, M., et al.: Information flow control for standard OS abstractions. In: SOSP '07, New York, NY, USA, ACM (2007) 321–334
12. Zeldovich, N., Boyd-Wickizer, S., Mazières, D.: Securing distributed systems with information flow control. In: NSDI'08, Berkeley, CA, USA (2008) 293–308
13. Efstathopoulos, P., Kohler, E.: Manageable fine-grained information flow. In: EuroSys European Conference on Computer Systems, ACM (2008) 301–313