Abstract—Aspect Oriented Programming is increasingly being used for the practical coding of cross-cutting concerns woven throughout an application. However, most existing AOP point-cut definition languages don’t distinguish in their application between different systems across a network. For network security there is a need to apply different aspects depending on the role a piece of code has within the larger networked system, and a new approach for this is therefore required. In this paper we present a formalism for how this might be approached, proposing a way to capture distributed point-cuts for applying different aspects in different parts of the network. The method is based on templates that match properties within the code, and a set of flexible relationships that can be defined between them.

I. INTRODUCTION

While the majority of modern, mainstream programming languages have adopted object-oriented programming (OOP) approaches, it’s nonetheless widely acknowledged that the OOP paradigm fails to adequately reflect the full breadth of structural features applicable to software development. In particular, while objects are a great way to compartmentalise functionality into reusable components, they can make it difficult to introduce functionality spanning multiple components. Aspect-oriented programming (AOP) has become established as an effective way for addressing this.

Although AOP can be achieved statically at compile-time, the most successful AOP technologies have drawn upon the reflective (introspection and introcession) capabilities of modern languages (e.g. those built using virtual machine abstractions, such as Java, .Net, etc.) to provide dynamic aspects that can be introduced and removed at run-time.

From a security perspective, AOP techniques are an exciting development that offer considerable promise. Along with logging, redundancy and Quality of Service (QoS), security features are invariably considered as canonical examples of cross-cutting concerns particularly amenable to AOP approaches. However, security concerns present particular challenges, not least because the most pressing and relevant cases occur when components are distributed across a network. In contrast, most AOP approaches still focus on individual pieces of software, with aspects woven into the code in multiple places, but without considering how concerns cut across multiple interacting networked systems.

While security features should therefore be ideal AOP candidates, the need to introduce them across distributed networked systems creates both conceptual and practical challenges. For example, how can we describe concerns to allow reasoning across multiple heterogeneous components? Having established a suitably expressive language, how can aspects be applied simultaneously and so as not to inadvertently introduce new security vulnerabilities?

In this paper we focus on the question of describing suitable patterns. While a number of AOP methodologies have been developed with distributed systems in mind, including several that consider security in particular, we believe there is still scope for improvements in terms of their expressiveness, and that these improvements are needed to allow the wider variety of security requirements found in networked systems to be adequately described.

The remainder of this paper is structured as follows. In the next section we will consider a selection of the existing proposed AOP approaches applicable to distributed systems. We build on this in Section III by considering examples issues that these systems do not yet address, but which are needed for security patterns in networked systems. Section IV turns these into a set of requirements we aim to fulfil, with the resulting point-cut definition language described in Section V, followed by applications in Section VI. Finally we conclude and suggest future work in Section VII.

II. RELATED WORK

Broadly speaking, AOP involves the introduction of aspects into a piece of software. These aspects represent cross-cutting concerns, so that the same (or similar) functionality can be added in the same way across multiple components (e.g. objects or methods) within the same piece of software.

There are already a number of well-established AOP platforms, including AspectJ, PROSE, JAsCo, LOOM.Net and DSAW. One of the important characteristics that distinguishes these different platforms is the selection of join-points each supports. A join-point represents the possible types of location where aspects can be introduced into an existing body of code. Common examples of join-points

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant no 257930 (Aniketos).
include before, after or around a method call (where the latter potentially allows the call to be replaced entirely). In addition, some platforms will allow new attributes to be added to objects, or alteration of attributes through changes to the relevant getter or setter methods. Other language features, such as operator overloading, ensure even a relatively restricted set of join-points can nonetheless offer powerful and flexible program extensions.

Join-points provide a set of potentials which are turned into actuals through the use of point-cuts and advice. A point-cut is a description of a precise set of join-points, restricted for example by method name (often using wildcards) or parameter types. Each is associated with an advice, which constitutes the code to be injected at each of the join-points satisfying the point-cut description.

These existing techniques are well-suited to individual pieces of software, but require extension in order to be applied in distributed environments. For example, Pawlak et al. [1] developed A-TOS, a “general purpose aspect-oriented reflective middleware” to introduce such extensions. In A-TOS aspects are encapsulated in aspect components, which describe both wrappers that are applied to – and respond to events from – the existing distributed objects, and centralised code that provides a global state and semantics available to all the individual wrappers.

This provides a clean way for describing distributed aspects, and the authors also apply it to security in particular, demonstrating how Kerberos-like authentication can be achieved using the technique. However, while the system utilises the network to offer a global state to the aspect code spread across multiple components, it provides limited means for reasoning about how the wrappers should be applied (e.g. refining the choice of join-points based on the nature of the components).

A different approach is taken by Duzan et al. [2], by focussing on the boundary between a component and the middleware used to abstract the network functionality (e.g. CORBA). This work allows advice to be added across client-server connections, for example to allow different data to be sent from the server depending on the network conditions at the client end. The work introduces the concept of regions, which allow different code to be executed depending on dynamic properties such as QoS parameters defined using a contract. The work focusses tightly on QoS, but the use of contracts to define dynamic behaviour dependent on external conditions could potentially be adapted for security properties.

The DiaAspect Architecture Description Language [3] also focusses on distributed middleware, but in this case as an extension of existing modelling languages such as UML. The DiaAspect models can be translated into more specific implementations based on technologies such as RMI or Web Services. A carefully tailored join-point model allows point-cuts to be defined based on a variety of messaging patterns that include commands, publish/subscribe events, network sessions, component registration/discovery, and so on. Advice is injected at both the message emitter and receiver ends. As a modelling language DiaAspect is potentially well-suited to the abstraction of patterns, and the authors demonstrate its use for both SSL certificate management and access control.

At the implementation end, Horie et al. [4] take synchronisation of advice injection as a cross-cutting concern itself. They show how aspects can be used to bootstrap the synchronisation of other aspects by first injecting code to separate deployment from activation, before using this to simultaneously activate further aspects. A method is proposed for dynamically introducing encryption/decryption that avoids packet loss that would occur if the encryption code were to be activated before the decryption code.

III. AOP SECURITY CHALLENGES

While all of these techniques tackle important issues related to applying distributed aspects, one area which is left open is that of reasoning about how aspects should be applied across multiple distributed components. Invariably a binary client/server-style relationship is assumed. While in many cases aspects can be easily introduced across connections (i.e. different coordinated code at the client and server ends of a connection) there is no coordination across multiple components.

We therefore explore how patterns can be defined to allow aspects to be woven into multiple components straddling multiple network connections. To understand the requirements of this approach we first consider a number of example scenarios.

A. Secure Data Forwarding

A very simple example of a security requirement is where data should be accessible by one system but not another (e.g. as a result of data classified at different security levels, as shown in Figure 1.a). This can be enforced using AOP at network endpoints, by designing aspects to add metadata and encryption at one end of a network connection, then check
the metadata and decrypt it at the other end. This scenario can be captured using aspects applied to pairs of atomic services and can already be represented using existing (e.g. client-server) methods.

However, this scenario can be extended, where for example data might be allowed to travel through a system not entitled to access it, as long as decryption doesn’t take place at the intermediate systems (See Figure 1.b). This requires a more complex pattern description than a simple binary relation.

B. End-to-end Security

Extending the example in Section III-A, we might consider a case where endpoint security must be applied using aspects at the source and destination node, but tackled differently at intermediate nodes (see Figure 2).

In this case, a pattern capturing the chain of nodes, but where the length of the chain is indeterminate, is needed. Current methods have difficulty capturing this case adequately.

C. Separation of Duty

Not all patterns involve linear chains and it may be necessary to match different branch-types within a network graph. For example, a separation of duty requirement might allow component A in Figure 3 to access B only if it had not yet accessed C (or vice versa). Aspects could be woven into A, B and C to enforce this, but would need to be applied only where such a split occurs.

IV. REQUIREMENTS

In order to allow the above concepts to be formalised through pattern descriptions, we formulate a number of requirements that we believe a distributed point-cut definition language must fulfil.

1) Patterns must capture sequences of more than two systems.
2) Patterns may match multiple (potentially infinite) sets of networked system. Flexibility is needed in both the depth (length of sequences) and breadth (number of links entering/leaving a system) in the network.
3) Patterns must not be indeterminate. That is, while there may be multiple arrangements that satisfy a pattern, the question of whether a particular pattern matches a particular set of systems should be decidable.
4) Pattern matches should be based on both the contents of the system code (obtained through introspection) and the relationship between systems (the structure).
5) It should be possible to relate the aspect code with the distributed point-cut and associated code. For example a different aspect is needed at either end of an asymmetric encrypt-decrypt network link.

V. POINT-CUT DEFINITION LANGUAGE

In order to fulful the above requirements we define a formalism for specifying patterns that allows system properties to be combined with network interactions. Our intention is to allow a language to be developed from this for the flexible definition of distributed AOP point-cuts and advice.

In essence, we aim to specify a graph, where each node in the graph represents an atomic system (a piece of code to which an aspect can be applied), and the links between nodes represents possible interactions between code across the network.

However, we must generalise this, so that the nodes represent sets of atomic systems (defined by a set of properties that a system must match). Consequently we distinguish between an actual graph of interacting systems and a template graph. The actual graph represents a set of networked systems. If each atomic system (nodes in the actual graph) fits the properties of a respective node in the template graph, and if the interactions also match, then the template matches the actual. Figure 4.a shows an example of a template, with various properties \( p_1, \ldots, p_5 \), and an actual graph that might match it in Figure 4.b with actual systems \( n_1, \ldots, n_6 \).

This would allow us to test templates against actual graphs, but doesn’t tell us how to apply any aspects. For each node in the template graph we therefore assign a pointcut template and advice to apply to that system. These are
represented by the point-cuts and advice $a_1, \ldots, a_4$ in Figure 4.a.

Unfortunately such a system is still too restrictive. In many cases we want to apply advice to networked systems without being so prescriptive about the interactions between them. For example, in a client-server setting, we may want to specify advice to apply to the client and server separately. We can capture this with the graph shown in Figure 5.a (where $C$ represents client code and $S$ represents server code), but this will then fail to match if the server is managing multiple connections as in Figure 5.b. Flexibility is therefore needed to define which connections are relevant for a template, which can be ignored and which are prohibited (i.e. result in a template that doesn’t match).

In order to fulfill Requirement 2 we must therefore allow flexibility in the way the input and output patterns are defined. Presenting this diagrammatically is difficult: how should we represent a set of potentially infinite distinct input and output patterns on a single diagram? We therefore consider a more formal description.

First we establish some definitions. Let $N$ be the set of actual services in the scope of consideration, with $n_1, \ldots, n_j \in N$ being actual systems as above. Let $P$ be a set of property descriptions and $A$ be the set of point-cut advice definitions with $p_1, \ldots, p_k \in P$ and $a_1, \ldots, a_l \in A$. The $a_i$ can be written in any existing AOP point-cut specification language such as that provided by DSAW [5]. We leave the choice of language for both the property definition and aspects open at this stage.

Let $S$ be a set of states; for simplicity we can assume $S = \mathbb{N}$ (the natural numbers) using some kind of encoding. In practice it may be convenient to give $S$ a more defined structure.

It will be convenient for us to define sets of properties, as well as sets of properties with associated actions (advice and point-cuts). We therefore note that these can be defined as any member from the sets $2^P$ and $2^{P \times A}$ respectively, where $2^X$ represents the power set of $X$. A member $x \in 2^P$ can be any subset of $P$, so takes the form

$$x = \{p_1, \ldots, p_j\}$$

for some $j \in \mathbb{N}$ and $p_i \in P$. A member $y \in 2^{P \times A}$ takes the form

$$y = \{(p_1, a_1), \ldots, (p_k, a_k)\}$$

for some $k \in \mathbb{N}$, $p_i \in P$ and $a_i \in A$.

For a given node satisfying a given set of properties $p \in P$ we want to define a set of possible templates to match it against. A single template is defined as follows.

$$(a, x_1, x_O, \bar{x}_1, \bar{x}_O) \in A \times 2^{P \times A} \times 2^{P \times A} \times 2^P \times 2^P.$$  

We call this an atomic template since it allows a specific set of input and output properties to be specified as a template for a single atomic system. We set $T$ to be the set of all atomic templates:

$$T = A \times 2^{P \times A} \times 2^{P \times A} \times 2^P \times 2^P.$$  

Suppose we have a node $n$ satisfying the properties of $p$. Then in the atomic template for $p$ the set $x_1$ represents the set of properties that any node with inputs to $n$ must satisfy; $x_O$ represents the set of properties that must be satisfied by any node $n$ connects to; $\bar{x}_1$ represents the set of properties that any node with inputs to $n$ must not satisfy; $\bar{x}_O$ represents the set of properties that any node $n$ connects to must not satisfy. Note also that each property in $x_1$ and $x_O$ has an advice associated with it. In case the complete template holds, this advice will be applied to the related actual node satisfying the property, which we will discuss in more detail shortly.

The above defines a template for only one property. We must extend this to allow templates for graphs of properties (that will then be matched against actual systems). We do this by specifying a function $f$ of the following form.

$$f : P \times S \to 2^{T \times S}.$$  

The function $f$ defines a single general template and can be used to match against sets of actual systems. Note that as well as the atomic template, the function also maps to and from an input and output state $S$. This is to allow the same atomic template to apply at different places in the graph in different ways. We will assume that the initial state $s_1$ for any general template must be zero.

So for example, consider the arrangement from Figure 4.b again. The system $n_1$ satisfies the property $p_1$, hence we first apply the function to $(p_1, 0)$. The function will return a set of associated atomic templates $t_1, \ldots, t_j$ and an updated state $s_2$. We can then test the inputs and outputs of $n_1$ against the properties in each of the $t_i$ to establish whether any of these hold. If they do, the process is repeated recursively, applying the function to each of the atomic systems that $n_1$ connects to/from, along with the updated state $s_j$. Note that at any point it may be possible for multiple of the $t_i$ to match. In this case, each is tested separately. The process completes if either every recursive branch fails to match at some level, or one of the templates matches in full.

Figure 5. Client-server template.
In the event that the template matches in full, the aspects defined in the template should then be applied to their respective systems.

To understand this better, in the next section we will return to our examples from Section III to see how such functions can be defined.

VI. APPLICATION

In order to demonstrate further the application of the pattern descriptions, we consider how they might apply to the examples from Section III. For secure data forwarding the key is to apply four different aspects at different points across the sequence of systems. We can do this by defining the function $f$ to contain the following mappings:

$$(H, 0) \mapsto (\emptyset, \emptyset, \{(L, a_I^H)\}, \emptyset, \emptyset, 0),$$
$$(L, 0) \mapsto (\emptyset, \{(H, a_I^L)\}, \{(L, a_I^H)\}, \emptyset, \emptyset, 0),$$
$$(H, 0) \mapsto (\emptyset, \{(L, a_I^H)\}, \emptyset, \emptyset, \emptyset, 0),$$

with all other inputs mapping to the empty atomic template. This matches only sequences of the form $H \rightarrow L \rightarrow H$. Note that for the above template any other inputs or outputs are ignored, but we could set it to fail if there are any other inputs or outputs by setting the $x_I$ and $x_O$ sets accordingly.

For the case of end-to-end security we require a slightly more flexible function. In this case we define the following mappings (with all others mapping to the empty template).

$$(A, 0) \mapsto (\emptyset, \emptyset, \{(B, a_I^A)\}, \emptyset, \emptyset, 1),$$
$$(B, 1) \mapsto (\emptyset, \{(X_1, a_I^{X_1})\}, \{(X_2, a_I^{X_2})\}, \emptyset, \emptyset, 1),$$
$$(A, 1) \mapsto (\emptyset, \{(B, a_I^A)\}, \emptyset, \emptyset, \emptyset, 2),$$

where $X_1, X_2$ are one of $A, B$ and $Y_1, Y_2$ are one of $B, A$ respectively. This generates a total of six templates. The first requires a sequence $A \rightarrow B$. We then have four mappings that allow any sequence of the form $A \rightarrow B, B \rightarrow B, B \rightarrow A$ or $A \rightarrow A$. Note that the last of these could be left out if we required at least one connection through $B$. The last template stipulates the input requirement of the final $A$ node. This is needed in order to ensure the sequence of templates matches through to the end.

Finally, the separation of duty requirement could be specified using a function $f$ with the following mappings.

$$(A, 0) \mapsto (\emptyset, \emptyset, \{(B, a_I^A)\}, (C, a_O^C)\}, \emptyset, \emptyset, 1),$$
$$(X, 1) \mapsto (\emptyset, \{(A, a_I^{X})\}, \emptyset, \emptyset, \emptyset, 1),$$

where $X \in \{B, C\}$, generating three rules in total.

This case is slightly different, since we have two tuples in the $x_O$ set of $A$: one for $B$ and another for $C$. However, note that the rules for $B$ and $C$ have only a single input, since the template only sees the limited view of the world from the perspective of $B$ or $C$ respectively. Any interactions that requiring a larger world-view must capture this through the state variable.

VII. CONCLUSION

In this paper we have briefly outlined a formalism for applying aspects to networked systems which extends existing techniques in a novel way, thereby providing greater flexibility. The approach allows complex structures to be defined across multiple networked systems using mappings between properties and atomic templates to which actual systems can be matched. While this provides the high-level mechanisms, there are a number of significant gaps in the work. For example, we did not present the point-cut language or explain how properties can be specified. Although an existing point-cut and advice language could be used, a more complete solution would also integrate state values into the advice.

Our approach as presented here is also theoretical, and doesn’t explain how a template function would be specified in practice. All of the examples given here require only a finite set of possibilities for the inputs and outputs from any given system, which allows us to specify them as a finite list of atomic templates. The reason we use a function rather than a list is to allow more flexibility than this, and in general it wouldn’t be possible to represent $f$ using a finite list. In our future work, we therefore aim to demonstrate how these issues can be overcome in practice, using a practical implementation for applying distributed AOP security techniques.

REFERENCES


