Analysis and Detection of Access Violations in Componentised Systems

David Llewellyn-Jones, Madjid Merabti, Qi Shi, Bob Askwith
School of Computing and Mathematical Sciences
Liverpool John Moores University
Email: {D.Llewellyn-Jones, M.Merabti, Q.Shi, R.Askwith}@ljmu.ac.uk

***Abstract*—Componentised systems present significant security difficulties, especially relating to how access control can be enforced across multiple systems without imposing overbearing restrictions on the flow of data. In this paper we present a practical approach to analysing data flow between components using the MATTS composition analysis tool, for the purpose of detecting potential access control violations across the entire composed system. As a corollary, we also present experimental evidence to show that refining the algorithm to accommodate component structure information can reduce an exponential time check to one requiring only linear time.**

I. Introduction

With movement towards a Ubiquitous Computing world the distribution of devices and data across a network is becoming predominant. The consequences for access control are significant. Traditional access control mechanisms have been centralised in their nature, with access being granted or restricted at the location that the data is held in the network. Examples of such systems include Taos [1] or local access control mechanisms. A number of distributed access control mechanisms have also been proposed that aim to allow the distribution of data throughout a network, whilst maintaining access control integrity. Examples of these include Chothia *et al.* [2], DSS DACS [3], and DSI [4]. Such systems often utilise certificate based authentication to control access to data. For distributed systems, CORBA also has some access control functionality in the form of the CORBASec specification. This applies access decision criteria at both the client and server sides of a communication link based on caller privileges, privilege controls, the requested operation and the target access policies [5]. Such access control mechanisms can be considered atomic, in that they take account only of relationships between pairs of components, rather than of the wider context or structure that a component forms a part of.

In contrast with these systems we present a method for testing whether a particular configuration of components is likely to cause problems from an access control perspective. This can be useful for a number of reasons. The aforementioned techniques are only able to produce runtime warnings, whereas our testing mechanism can be used to discover inconsistencies as the links between component systems are configured. Once a system has been tested its safety can be ensured and no further checks are require until a system is restructured. Moreover, we employ formal code analysis techniques to guarantee compliance, reducing the trust that must be placed in individual components to conform to an access control policy. Such formal analysis techniques would be overbearing for a large system, but by analysing each of the components individually, we show that we can reduce the exponential analysis process to one requiring only linear time.

Access control mechanisms for componentised systems are becoming increasingly important due to the proliferation of massively networked systems and a move towards Ubiquitous Computing without centralised control. The need for a practical model of data flow through a system is therefore of paramount importance.

We can understand the difficulties of applying access control in componentised systems by first considering access control as it applies locally. On a Unix-like system, access to data is determined based on the user's ID (UID) and the read and write permissions of the file (for simplicity we ignore execute and other permissions). Different access permissions can be set for the owner of a file (who ultimately maintains full control over it, given that they are also able to alter the access permissions), members of the owner's group, and all others. The access rights and file owner are all stored as meta-data in the file's inode.

In theory access restrictions are straightforward. The user can only access a file where the access rights allow it. In practice the user can only access the file through an intermediary program. Consequently each executable must also have access rights associated with it. In the simplest case a program will inherit access permissions from the user or process that spawned it, achieved through the association of the parent's UID with the process. However, in some cases (such as for an SUID process) a program can take on a different UID (this becomes the process's *effective* UID or EUID). This is often necessary so that the program can perform operations that it would otherwise not have the rights to undertake. For example, a password changer requires access to the passwords database, and a mail daemon may need to store emails in a variety of users' directories. Such programs must be carefully designed to prevent users gaining unauthorised access to data.

This requirement to elevate privileges has been the source of many vulnerabilities, allowing access to data for users who do not have the necessary rights. At least 72 such reports were added to the US-CERT Vulnerability Notes database in the 12 months to March 2007 (amounting to 24% of the reports up from 18% in 2005).

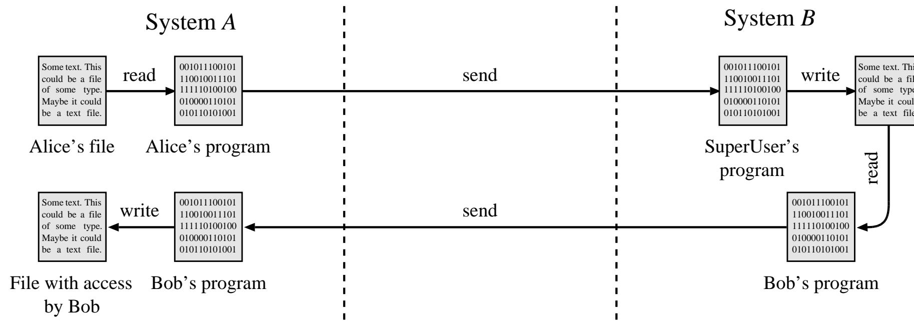


Fig. 1. Elevating file access across a network

In a distributed or networked environment, this can make access control particularly precarious. For example in a networked environment, the domain of an operating system's control may extend only to a single device. On a remote device, there is no reason to assume that access control meta-data will be adhered to. The local OS may not have complete control over this meta-data. Moreover it may be necessary to rely on the administrator of multiple machines to ensure executables with special access rights do not provide a back-door to gaining unauthorised access.

Consider the example shown in Figure 1. In this case the access policies of system A are set to prevent Bob from accessing Alice's restricted files. However, on system B, where no secure data is stored, Bob is able to access the restricted data using a program with elevated access. On each system considered individually, the access policies are adhered to in the system. However, when the two are used as a composed system, due to the differences in the configurations of the two systems, Alice's access policies may be violated.

Ideally, it should be possible to consider such interactions to establish whether such problems are likely to occur. It is this that we hope to achieve using our method.

Although we have talked about access control in Unix systems, our approach looks primarily at data flow through a system. It can therefore be applied more generally to any related access control model. In this way it has similarities with other composition work, such as non interference [6] and related results [7]. These also consider data flow, although concentrate on different areas of security such as covert channels. We also go slightly beyond many of these results by presenting a fully automated testing method based on formal analysis techniques.

We anticipate that distributed access controls will be of particular importance in Ubiquitous Computing environments, where data is distributed amongst many devices, with users having simultaneous access through identities that apply equally on all devices. Data restrictions between devices are anathema to the Ubiquitous Computing paradigm.

Extending our description of traditional access control models from above, we see that they have tended to concentrate on the relationship that exists between data, as the artefact of restriction, and processes, as the means for users to access it. More specifically, the control point in traditional systems is placed between the data and the process requesting access to the data.

Such a system works well where data is centralised, inter-process communication is limited or processes with elevated access rights can be trusted. However, in a Ubiquitous Computing environment, where data is distributed across many devices and interaction between devices and processes is routine, such solutions may not be practical. We believe an approach based on testing that can be enforced before a component or device is granted access to a network, will have greater applicability.

The novelty of the process therefore lies in two areas. First we present a new algorithm for the automated analysis of access control across a componentised system which can detect potential problems before they occur, rather than throwing runtime exceptions. We believe this process to be unique in the way it can perform an automated analysis of components.

Second we present results to show that by combining composition analysis and formal analysis, significant improvements in efficiency can be achieved. In effect, the injection of structural knowledge into the analysis process can reduce checking time from exponential to linear.

We demonstrate the process be implementing it using MATTS, a framework for composition analysis based on mobile agents. Each agent represents a component within the system able to communicate with other agents via the network. Our algorithm combines composition analysis between components along with **slicing** to allow data flow within components to be considered. Both elements are combined within the framework to produce a fully automated analysis process.

The structure of this paper is as follows. In the next section we discuss the generalization used to describe access control systems in our work. Section III then presents our algorithm and implementation for analysing access control data flow using composition analysis and slicing. In Section IV we present the results of applying the analysis to a set of interacting mobile agent components. Finally, we discuss future work and conclude in Section V.

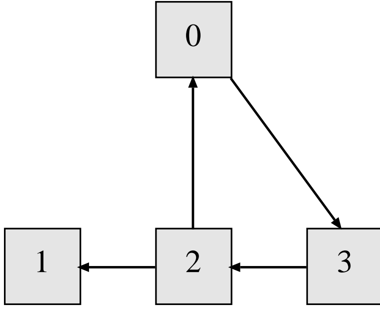


Fig. 2. A simple component structure

II. Generalised Approach

We consider componentised structures to be comprised of multiple components that interact together across specified directed links between pairs of components. A simple four component structure is shown in Figure 2, with arrows representing data links between components.

Each component is assigned four data structures to determine access rights. The first pair, u_R and u_W, represent the effective user ID for read and write operations of the process respectively. The second pair, d_R and d_W, represent the access attributes of the data (files) actually read from and written to by the process. In general, all we suppose is that there exist mappings $f_R, f_W : U \times D \to \{0, 1\}$ where $u_R, u_W \in U$ and $d_R, d_W \in D$. These functions determine whether access should be granted or not.

For example, suppose authorisation is based on access control lists [8]. In this case, associated with each file we would have two lists of users: one dictating the users able to read from the file and the other dictating the users able to write to the file. A process may be required to access multiple files, each of which may have different access lists. To ensure compliance we would therefore take the intersection of all of these lists to provide the set of users that can safely have access to the same data as that process. To model this scenario we can therefore take U to be the set of user IDs, and $D = \mathscr{P}(U)$. Then for $i \in \{R, W\}$ we have

$$f_i(u_i, d_i) = \left\{ \begin{array}{l} 1 \text{ if } u_i \in d_i, \\ 0 \text{ otherwise.} \end{array} \right.$$

The framework can determine the u_R and u_W level by the levels of the user or process that initiated or spawned the component process, unless the component itself relinquishes privileges or is an SUID process. The d_R and d_W attributes are based on the files a component accesses during execution, which in many cases can be automatically determined by the security algorithm we present. Access to resources is then controlled by the comparison metrics $f_R, f_W : U \times D \to \{0, 1\}$ so that a component is granted read access to a resource if and only if $f_R(u_R, d_R) = 1$ and write access if and only if $f_W(u_W, d_W) = 1$.

Other access control schemes can also be implemented under this scheme. For example, access control matrices, or the Bell-Lapadula model can both be easily represented using this scheme. In the latter case, we could simply assign numerical access clearance levels to u_R, u_W, d_R and d_W and set f_r and f_w to be the inequalities $u_R \geq d_R$ and $u_W \leq d_W$ respectively.

III. Data Flow Analysis and Slicing

We describe a system to allow for analysis of component interactions based on the access criteria outlined in the previous section. The method involves following the flow of data through the componentised system, ensuring at each stage that the access restrictions for the data are enforced. Intuitively the read access algorithm involves considering each component and keeping track of the access rights for the data leaving the component based on the rights assigned to the data read by that component. This data is followed through the network until it reaches another component, whereupon we apply slicing to the component in order to determine whether the input data is transferred through to any of the output links for that component. This process is continued and at each stage the check f_R or f_W is performed as described above to ensure no access violation occurs.

If we were not interested in the internal structure of components we would have to assume any data that enters a component could potentially be transmitted on any of the output links of the component. In this case we could simply perform a depth first traversal of the component structure to trace where the data could spread through the network [9]. However, this would also result in a highly conservative process that might suggest violations when none were possible, producing a high false positive rate.

Instead, the slicing technique can determine whether data from a particular input is ever transferred to a particular output of a given component. The slicing technique we apply is an automated method of formal bytecode verification allowing a program to be broken down into individual slices. A slice usually represents all elements in a program that relate to a particular variable in the program. The formal notion of slicing was introduced by Mark Weiser in 1981 [10]. We use it to establish correlations between data inputs and data outputs, thereby modelling data flow through a component.

For example, suppose we have a component as shown in Figure 3(a) with access function f_W as described earlier.

Without knowledge of the internal structure of the component, the outputs O_1, O_2 and O_3 would all have to be assigned d_W levels of $a \cap b$, the most restrictive combination of the access levels for I_1 and I_2. However, suppose the internal structure of the component could be considered and that it could be seen to be as in Figure 3(b). In this case only O_2 would have its access d_W set to $a \cap b$. However O_1 outputs data with no portion taken from I_2. Consequently this output need only have d_W set to the less restrictive value a. Similarly O_3 can have a less restrictive access d_W of b. We have incorporated this process into our implementation using simple code analysis based on the method of Floyd [11], built into the MATTS component composition environment.

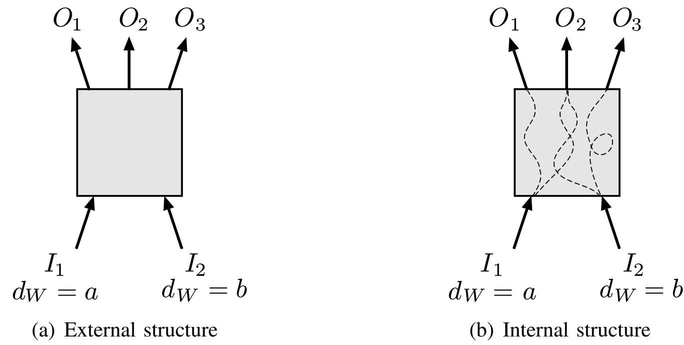


Fig. 3. Slicing a component to establish internal structure

In applying the slicing technique, we hope only to traverse paths for which data flow exists between inputs and outputs. As we enter a node we must keep track of the input channel on which we arrived and analyse the component using slicing based on this, and the output we are considering continuing our journey on. In Figure 4, we suppose that component 3 sends data arriving on channel 1 out on channel 3, and data arriving on channel 2 out on channel 4, but also that no data from channel 1 is sent on 4 and similarly for 2 and 3. In this case, the slicing will establish these paths and whilst the data flow should continue from channel 1 through component 3 on to channel 3 and component 4, there is no need for it to follow channel 1 through component 3 on to channel 4. As we can see in the left hand branch of Figure 4(c), the path from 3 does not continue to component 5.

This is where we must be careful when travelling onwards from component 2. A simple depth first traversal algorithm would not continue on to component 3 in spite of it being connected, due to the fact that it has already been visited, as shown in Figure 4(b). However, in the slicing version, this path now becomes relevant, since the path on to component 5 – whilst not relevant when starting from component 1 – becomes relevant when travelling from component 2. We therefore see in the right hand branch that we should continue on to component 3 and then component 5, despite the fact that component 3 has already been visited.

We devise the algorithm needed to achieve the unravelling in Figure 4(c) by applying a depth first traversal of the network, but rather than traversing components we traverse *links* instead. We generate a new network graph by projecting the links as points onto the new graph with each pair of projected points being connected only if data travels between pairs of links, as determined by the original graph and the slicing. A depth-first traversal of this projected graph provides the structure needed.

Applying this to the example in Figure 4(a) results in the structure shown in Figure 4(c). Using this algorithm we ensure that every unique *pair* of links from the original graph that can be reached from the root of the original graph is visited during the traversal. The case for read access is similar, but again, with the traversal reversed.

The complete algorithm has been implemented using the MATTS 'Compose' framework, and the case for data writing can be applied using the script outlined in Table I.

This script will be described in the remainder of this section., although space constraints prevent us from providing detailed script syntax and this can be found elsewhere [12].

The main process of the script involves checking each component individually (lines 5-28). For each component a record is maintained of its u_W access rights (lines 10-11 and 18-19), representing the possible access rights of all data the component writes out. For each component a process is spawned to establish the access criteria for all data that enters the node (lines 9 and 17).

The spawned process (lines 30-75) does the bulk of the work. A network graph with points representing projected links and links representing data flow between pairs of points is established and traversed recursively (lines 39-74). As this graph is traversed the access rights d_W of the components that are passed through enroute are compared against the access rights u_W of the original component that we are testing against, and which the data will be flowing into (lines 48-51).

At each stage the formal slicing analysis is undertaken (lines 58-61) to establish if a pair of points should be linked, and if so the graph is updated to reflect it and traversed accordingly (lines 63-73).

The script represents a pattern that must be matched by the component structure, based on the properties assigned to each component, in order for the structure to be considered free from write access violations. The result is that we can be sure that each component only receives data with access rights d_W, which are compatible with the access rights u_W of that component. Consequently, the entire system makes no access violations, even in the event that different components have differing access rights and access data with differing access requirements, passing this data between them. The analysis can be conducted dynamically whenever the component topology changes. The invariant enforced by the analysis is that no data leaving a component will have access requirements d_W that conflict with the access rights u_W of any component that this data is transmitted to.

The case for read access is similar, except that we must consider data flowing out of a component, and hence the depth-first traversal must be performed in reverse; that is, travelling

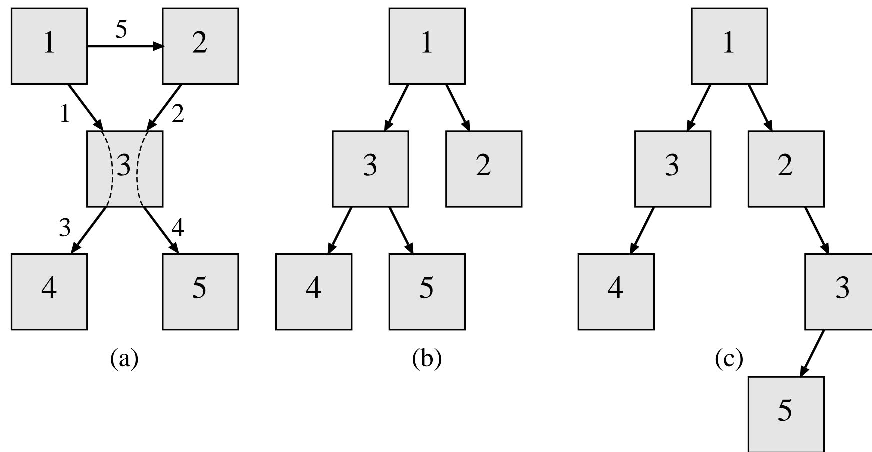


Fig. 4. Traversing a graph (a) considering components (b) and pairs of links (c)

along links in the reverse direction.

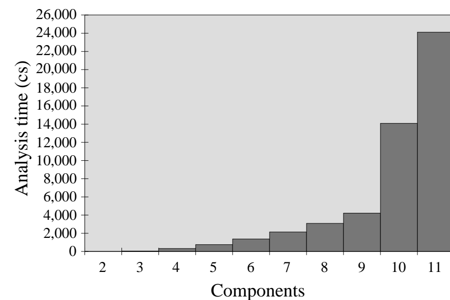
One of the important advantages of using such an analysis over enforcing access policies directly during execution, is that it opens up the possibility of preventing access violations through analysis and potentially providing solutions using trusted intermediary components strategically placed within the component topology. At present we leave methods for generating such solution topologies for future work.

IV. Results

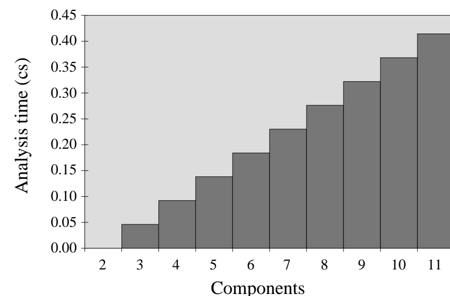
Using the algorithms described above, we ran a number of tests to establish the potential for access violations in various structures. We also aimed to establish any timing differences between a componentised approach as compared to the traditional formal methods of analysing a system as a contiguous whole.

The analysis algorithms were created using MATTS with results obtained running the analysis on an ARM compatible Intel XScale 80321 processor running at 600 MHz and with 512 Mb of RAM. Such resources are similar to those that might currently be found on a top end PDA. All tests were repeated 50 times, with the results given being averaged timings.

We considered a series of components composed linearly, with data being sent from the first component through to the last along a series of data channels. Using this configuration we are able to compose a variable number of components and compare the timing as more components are introduced into the system. The MATTS framework is based on networked mobile agent interactions, so we compiled a number of agents that send arbitrary strings of data between each other with various access rights assigned to them. We then applied the analysis script described above to determine whether an access violation would occur in the system. Although based on mobile agents, this implementation can be seen to reflect a typical componentised system with data being passed between components.



(a) Formal analysis



(b) Component analysis

Fig. 5. Experimental timing results

In order to establish whether component analysis resulted in timing benefits over a traditional formal approach, we also undertook tests which treated the multiple components as a single contiguous block of code, with data moving around within this single code segment. Rather than use the script to combine the results of the multiple components, we instead use formal analysis on the code of the complete composed application. What would have been links between components have to be modelled using stacks, with one stack set up for each channel. Data sent out on a particular channel is stored in the corresponding stack, subsequently being pulled off the stack when data is read from the channel. In effect, we applied formal analysis to a single mobile agent compiled to undertake

TABLE I
WRITE ACCESS CONTROL ANALYSIS SCRIPT.

```
 1 <property id="dW">Component data access</property>
 2 <property id="uW">Component authorisation</property>
 3 <property id="slice">Slicing analysis</property>
 4
 5 <configuration id="c1" init="1">
 6   <component id="c2">
 7     <input format=""/>
 8     <input id="in1" init="0" format="*"/>
 9     <process config="check" action="check=@n"/>
10     <process id="auth1" init="1" action="@a[@n][0]
11       [uW]"/>
12     <process cond="result &lt; auth1" action=
13       "c1=0"/>
14     <output id="out1" init="0" format="*" cond=
15       "c1==1"/>
16     <component>
17       <process config="check" action="check=@n"/>
18       <process id="auth2" init="1" action="@a[@n][0]
19         [uW]"/>
20       <process cond="result &lt; auth2" action=
21         "c1=0"/>
22       <input id="in2" init="0" format="*"/>
23       <output id="out2" init="0" format="*" config=
24         "c2" follow="fresh" cond="c1==1"/>
25       <output format="" cond="c1==1"/>
26     </component>
27   </component>
28 </configuration>
29
30 <process id="check" init="0">
31   <process id="result" init="0" action="@v=(0-1)"/>
32   <process id="stack_pos" init="0" action="@v=0"/>
33   <process id="node_stack" init="0" action=
34     "@v=check"/>
35   <process id="link_in" init="0" action="@v=(0-1)"/>
36   <process id="link_out" init="0" action="@v=0"/>
37   <process id="dir" init="0" action="@v=0"/>
38
39   <process id="recurse">
40     <process id="cur_node" init="0" action="@v=
41       node_stack[stack_pos]"/>
42     <process id="cur_link_in" init="0" action=
43       "@v=link_in[stack_pos]"/>
44     <process id="cur_link_out" init="0" action="@v=
45       link_out[(2 + cur_link_in + (8 *
46       cur_node))]"/>
47
48     <process id="write" init="0" action=
49       "@a[cur_node][0][dW]"/>
50     <process cond="(result &lt; 0) || (result &gt;
51       write)" action="result=write"/>
52     <process cond="(cur_link_out &lt; @olnum
53       [cur_node]) &amp;&amp; (cur_link_in &lt;
54       @ilnum[cur_node]) &amp;&amp; (cur_link_out
55       &gt;= 0) &amp;&amp; (cur_link_in &gt;= 0)"
56       action="cur_channel_out = @oloc[cur_node]
57       [cur_link_out]">
58       <process id="cur_channel_out" init="0" config=
59         "recurse" cond="!@ai[cur_node][cur_link_in]
60         [slice]" action="(link_out[(2 + cur_link_in
61         + (8 * cur_node))] = (cur_link_out + 1))"/>
62     </process>
63     <process config="recurse" cond="(cur_link_out
64       &lt; @olnum[cur_node])" action="(stack_pos =
65       (stack_pos + 1)) + (node_stack[stack_pos] =
66       @oln[cur_node][cur_link_out]) + (link_in
67       [stack_pos] = @olil[cur_node][cur_link_out]) +
68       (link_out[(2 + cur_link_in + (8 * cur_node))]
69       = (cur_link_out + 1)) + (dir = 1)"/>
70     <process config="recurse" cond="(cur_link_out
71       &gt;= @olnum[cur_node]) &amp;&amp; (stack_pos
72       &gt; 0)" action="(stack_pos = (stack_pos - 1))
73       + (dir = (0-1))"/>
74   </process>
75 </process>
```

TABLE II
ANALYSIS TIMING RESULTS

Components	Entire flow analysis		Restricted flow analysis	
	Formal	Component	Formal	Component
2	4.30	0.000	2.80	0.00000
3	19.10	0.046	44.40	0.04624
4	33.65	0.092	320.50	0.09224
5	48.15	0.138	760.60	0.13824
6	62.60	0.184	1372.60	0.18424
7	76.95	0.230	2147.40	0.23024
8	91.55	0.276	3093.20	0.27624
9	106.05	0.322	4211.80	0.32224
10	120.50	0.368	14089.50	0.36824
11	134.85	0.414	24114.15	0.41424

the same process as a multiple mobile agent network.

Two cases were considered. First we tested a case where data sent from the first component is sent onwards throughout the entire system. For both the the formal and component techniques, analysis time increased linearly with each new component added to the system. Although the component technique ran significantly faster (by up to approximately 315 times), the analysis time appeared to grow linearly in both cases. The second test, however, provided a more convincing contrast. For this test the penultimate component failed to relay the data to the final component, in order to simulate a more complex data flow situation. The timing results for this can be seen in figures 5(a) and 5(b). The exponential time increase incurred by adding components in the case of the formal technique is improved to a linear increase in the case of the composition analysis: a qualitative improvement. The full set of timing results can be seen in Table II.

Such improvements follow as a direct consequence of the componentisation of the code. We posit that other componentised analyses are likely to be amenable to such improvements.

V. CONCLUSION

In this paper we presented an analysis technique that can be used to check whether data flow through a componentised system will result in access control violations, based on the access control requirements and rights associated with data and components respectively.

An algorithm was presented to undertake a depth-first traversal of projected pairs of links, whilst also incorporating the movement of data through individual components using slicing. The utility of this algorithm lies in its ability to assess data flow in relation to access control policies and determine if such flow is likely to cause violations of such policies. The algorithm was tested through the analysis of a system of interacting mobile agents.

Whilst the algorithm is of interest for the analysis of composed systems from the point of view of security, we feel the potential for secure composition to reduce an exponential time analysis into one taking only linear time - as verified by the tests outlined in this paper - is of particular importance.

Although theoretical composition results already exist in the literature, the work here provides a practical method for harnessing composition analysis to tackle a genuine security problem.

This builds on earlier work considering other security problems that can be tackled in this way, but also opens out the possibility for future work in establishing new security results for composed systems for implementation as MATTS scripts. The optimisation of existing resource intense security analysis techniques, such as the detection of security flaws within programs, also presents a potential direction for future work.

In the nearer term, we hope to develop the results beyond a proof-of-concept tool and integrate them into a security solution for Networked Appliances. To do this, a number of questions remain, largely relating to the practical problem of assessing dynamic changes between components. We are currently considering instrumentation as a way to achieve this.

REFERENCES

[1] E. Wobber, M. Abadi, M. Burrows, and B. Lampson, "Authentication in the taos operating system," *ACM Transactions on Computer Systems*, vol. 12, no. 1, pp. 3–32, 1994.

[2] T. Chothia, D. Duggan, and J. Vitek, "Type-based distributed access control," in *IEEE Computer Security Foundations Workshop (CSFW)*. Asilomar, California, USA: IEEE, 2003.

[3] DSS, "Dacs: The distributed access control system, http://dacs.dss.ca/," 2005.

[4] M. Pourzandi, D. Gordon, W. Yurcik, and G. A. Koenig, "Clusters and security: Distributed security for distributed systems," in *Cluster-Sec 2005, The First International Workshop on Cluster Security*. Cardiff, UK: IEEE/ACM, 2005.

[5] U. Lang and R. Schreiner, *Developing secure distributed systems with CORBA*, ser. Artech House computer security series. Norwood, MA, USA: Artech House, 2002.

[6] A. Bossi, R. Focardi, C. Piazza, and S. Rossi, "Verifying persistent security properties," *Computer Languages, Systems and Structures*, vol. 30, no. 3-4, pp. 231–58, 2004.

[7] S. Tini, "Rule formats for compositional non-interference properties," *Journal of Logic and Algebraic Programming*, vol. 60-61, pp. 353 – 400, 2004. [Online]. Available: http://dx.doi.org/10.1016/j.jlap.2004.03.003

[8] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli, *Role-Based Access Control*, ser. Artech House computer security series. Norwood, MA, USA: Artech House, 2003.

[9] B. Carré, *Graphs and Networks*, ser. Oxford Applied Mathematics and Computing Science Series. Oxford, UK: Clarendon Press, Oxford University Press, 1979.

[10] M. Weiser, "Program slicing," *Proceedings of the 5th International Conference on Software Engineering*, pp. 439–449, 1981.

[11] R. W. Floyd, "Assigning meanings to programs," *Mathematical Aspects of Computer Science*, vol. 19, pp. 19–32, 1967.

[12] D. Llewellyn-Jones, M. Merabti, Q. Shi, and B. Askwith, "Buffer overrun prevention through component composition analysis," in *COMPSAC 2005*, 2005.