# Program Partitioning using Dynamic Trust Models*

Dan Søndergaard[1], Christian W. Probst[1], Christian Damsgaard Jensen[1], and René Rydhof Hansen[2]

[1] Informatics and Mathematical Modelling
Technical University of Denmark
`s011283@student.dtu.dk,{probst,cdj}@imm.dtu.dk`
[2] Department of Computer Science
University of Copenhagen
`rrhansen@diku.dk`

**Abstract.** Developing distributed applications is a difficult task. It is further complicated if system-wide security policies shall be specified and enforced, or if both data and hosts are owned by principals that do not fully trust each other, as is typically the case in service-oriented or grid-based scenarios. Language-based technologies have been suggested to support developers of those applications—the *Decentralized Label Model* and *Secure Program Partitioning* allow to annotate programs with security specifications, and to partition the annotated program across a set of hosts, obeying both the annotations and the trust relation between the principals. The resulting applications guarantee *by construction* that safety and confidentiality of both data and computations are ensured. In this work, we develop a generalised version of the splitting framework, that is parametrised in the trust component, and show the result of specialising it with different trust models. We also develop a metric to measure the quality of the result of the partitioning process.

## 1 Introduction

There is an increasing reliance on open computing environments; virtual organisations, supply chain management, grid computing, and pervasive computing increase the need for spontaneous and dynamic formation of short term coalitions to solve a specific problem. The security implications of this evolution are profound and new techniques must be developed to prevent violation of confidentiality and integrity in applications that execute on a widely distributed computing platform, e.g., a computational grid infrastructure.

Developing applications for these distributed environments, however, is a difficult task. In scenarios where both data and hosts are owned by principals that do not necessarily trust each other, as is typically the case in service-oriented or

* This work has in part been supported by the EU research project #016004, *Software Engineering for Service-Oriented Overlay Computers.*

grid-based scenarios, development is further complicated. Language-based technologies have been suggested to support developers of those applications—the *Decentralized Label Model* and *Secure Program Partitioning* allow to annotate programs with ownership and access control information, and to partition the annotated program across a set of hosts, obeying both the annotations and the trust relation between the principals. Starting from annotated *sequential* code, the resulting *distributed* applications guarantee *by construction* that safety and confidentiality of both data and computations are ensured. While this is a very powerful mechanism, the trust model used in the original framework [18, 19] is very limited. We start in this work with developing a generalised version of the partitioning framework, that is parametrised in the trust component, and show the result of specialising it with different, increasingly dynamic and realistic trust models.

Another important aspect is the trust that principals in the system have in the partitioning generated by the splitter. Even though the sub-programs are guaranteed to obey the annotations as well as the confidentiality and integrity specifications by construction, principals will want a way to measure how well a certain partitioning fulfils all their constraints. Therefore we develop a metric to measure the quality of the result of the partitioning process.

The approach we base our work on is fundamentally different from simply securing regular algorithms by encrypting data. While there are algorithms that do operate on encrypted data [2, 1, 15], such algorithms are usually very specialised and it in the general case it is not possible to transform an algorithm into an equivalent algorithm that operates on encrypted data. This means that users of grid applications must generally accept that both algorithm and data will be known to the computer that performs the calculations. It is therefore important to consider the segmentation of programs and data, in order to prevent sending sensitive data to the wrong, that is untrusted nodes (e.g., processing personal information, such as an electronic patient record, from a European citizen on a computer in the U.S.A. constitutes a possible violation of the European Data Protection Directive[8]).

This paper is structured as follows. The rest of this section introduces a running example, that we will use to illustrate different approaches. It is followed by a description of the splitting framework in Sec. 2. In Sec. 3 we develop a metric to judge the quality of a partitioning computed by the splitting framework, and in Sec. 4 we describe dynamic trust scenarios and their effect on the splitting process. Finally, Sec. 5 discusses related work and Sec. 6 concludes the paper and discusses future work.

## 1.1 Example

This section introduces our running example, that we will use to illustrate both the general framework and different trust models.

Consider the example of a pharmaceutical company that is in the process of developing a new drug. An important element of the drug discovery phase consists of high performance computing on large volumes of data, e.g., simulating chemical reactions, comparing gene sequences, etc. It is common for smaller pharmaceutical companies to specialise in particular types of drugs, resulting in many companies that belong to the same (pharmaceutical) trade association, but do not directly compete with each other. This is the typical scenario for a setting where one wants to develop applications that deal with data from companies that potentially distrust each other in handling some of their data, but are willing to share other parts.

The fact that some companies are registered in the same country and belong to the same trade association may increase the initial confidence in their sites and serve as a recommendation in case no previous experience with that site has been recorded.

The case that we will use as example is that of two companies $A$ and $B$ who want to execute a gene sequencing. In order to save costs, they use the same provider $C$ of such a service. In our setting, company $A$ has a license agreement with $C$, such that they receive a gene once $C$ has found a match. In contrast, $B$ only receives the boolean result and would have to pay for actually obtaining the gene.

## 2 The Splitting Framework

This section describes the components of the original framework for Secure Program Partitioning as described by Myers *et al.* [18, 19]. The main components

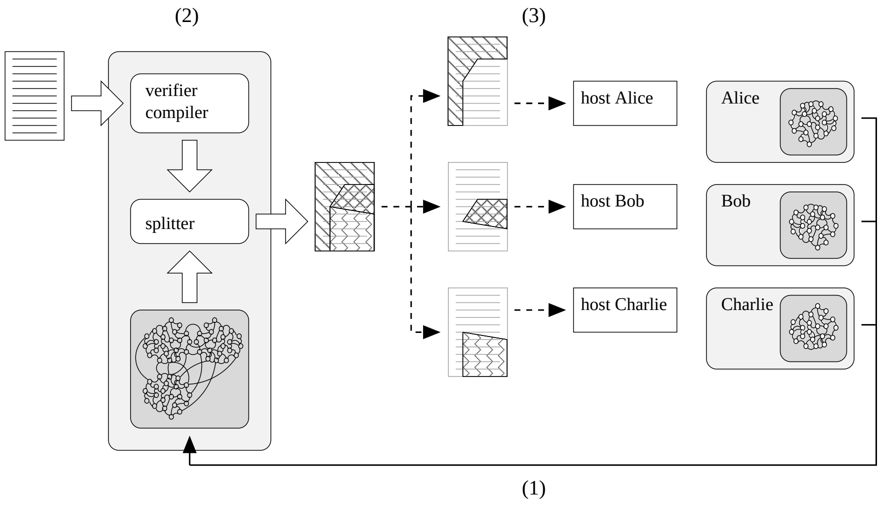

**Fig. 1.** The general splitting framework. Each principal in the system specifies a trust and an integrity graph (1), that is combined into the global trust graph and is used by the central splitter to partition the application across hosts in the net (2). Then the generated partitioning is distributed across the network and executed (3).

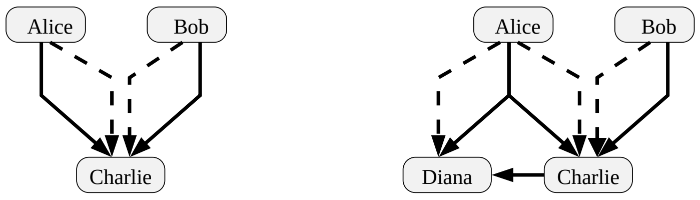

**Fig. 2.** Static trust graph for the example application. Solid edges denote confidentiality, while dashed edges denote integrity. The left graph depicts the initial static situation, the right hand graph the situation after Diana has joined the network.

are a security-typed programming language and a trust-based splitter. The partitioning process then essentially consists of two steps—*compiling* and *splitting*. The compiler processes the source code with security annotations as provided by a user, and the splitter tries to find a partitioning based on the trust relation of the principals. Fig. 1 shows the schematic overview of the framework. The input is an application written in a security-typed language. An example for a security-typed language is JIF [12] that extends Java with the Decentralised Label Model [13, 14]. Before discussing the framework in more detail we introduce the notion of trust used in it.

## 2.1 Confidentiality and Integrity

Trust in the original framework is divided in the two components *confidentiality* and *integrity*. It is noteworthy that these two components are independent from each other.

*Confidentiality* is used to express that an actor $A$ in the system has trust into another actor $B$ as to keeping $A$'s data confidential. This means that $A$ is willing to store its data on a host operated by $B$. For the rest of this paper we tacitly identify a host with the user operating it. *Integrity* is used to express that an actor $A$ assumes that results obtained from actor $B$ have been computed correct. This means that $A$ is willing to split computations on a host operated by $B$. Both confidentiality and integrity are complete and *non-transitive*, i.e., if $A$ trusts $B$ and $B$ trusts $C$, then $A$ does not automatically trust $C$.

A third measure that could easily be added is that of *availability* of a host in a distributed system. Principals could use this to express their trust in a certain host to be available to execute their computation. In contrast to confidentiality and integrity, which are subjective and hard to validate, availability is measurable in that one can check whether or not a host performed a certain operation. Therefore the availability assigned to a host could be increased or decreased over time. We are planning to further investigate this with our prototype implementation.

The two trust components confidentiality and integrity can be expressed as sets:

$$C_p = \{a_1 :, \ldots, a_n :\} \quad \text{and} \quad I_p = \{? : b_1, \ldots, b_n\} \tag{1}$$

where $p$, $a_i$, and $b_i$ are principals in the system. Using this notation, $C_p$ specifies, who has confidence in $p$, and $I_p$ specifies who believes in $p$'s integrity. For instance the trust graph on the left hand side of Fig. 2 can be expressed as

$$\begin{aligned} C_{Alice} &= \{Alice :\} & I_{Alice} &= \{? : Alice\} \\ C_{Bob} &= \{Bob :\} & I_{Bob} &= \{? : Bob\} \\ C_{Charlie} &= \{Alice : ;\ Bob : ;\ Charlie :\} & I_{Charlie} &= \{? : Alice, Bob, Charlie\} \end{aligned}$$

For the graph on the right hand side of Fig. 2, the results for Alice, Bob, and Charlie remain the same, and for Diana one gets

$$C_{Diana} = \{Alice : ;\ Charlie :\} \qquad I_{Diana} = \{? : Diana, Charlie\}$$

Because the trust model is *non-transitive*, Bob does not trust Diana with regards to confidentiality.

The next section introduces the formalisms used to annotate data and code with confidentiality and integrity.

## 2.2 Security-typed Languages

This section introduces the necessary background on the Decentralized Label Model (DLM) and shows the annotated high-level source code for our example application (Fig. 3).

The central notion in the DLM is the *principal* ($p \in \mathsf{Principals}$), which represents any entity that can own data or operate on it (that is users, processes, hosts, etc.). Principals are used in two ways—on the one hand in expressing ownership and access rights of data, on the other hand to define the authority that the program at any given point in time has. In the example, principals are $A$ and $B$, representing the pharmaceutical companies, and $C$, representing the service provider mentioned in Sec. 1.1.

Both ownership and access rights to data are expressed by *security labels* $L \in \mathsf{Labels} = \mathcal{P}(\mathsf{Principals} \times \mathcal{P}(\mathsf{Principals}))$, which can be written as sets. Each label specifies a set of owners for a piece of data and for each owner the principals that are allowed to access the data. In Fig. 3, the first part of the label of variable `seq1` ($\{A : C\}$) indicates that the variable is owned by principal $A$ and that $A$ allows $C$ to read it. Similarly, the label part $\{A : C; C : A\}$ at variable `t1` indicates that this variable is owned by $A$ and $C$, and each allows the other to read it. All components of a label $L$ must be obeyed in accessing data labelled with $L$. Compound labels with several owners also arise during computations using data tagged with only one owner, e.g., when invoking the method `scan` in Fig. 3 with arguments `g` labelled $\{C : A\}$ and `seq2` labelled $\{B : C\}$. The resulting label $\{C : A, B : C\}$ allows only $C$ to read data since the owner of data is automatically allowed to read it.

In addition to confidentiality information, labels can also contain *integrity* information. A label $\{? : p_1, \cdots, p_n\}$ specifies that principals $p_1$ through $p_n$ believe

```
sequence seq1 {A:C; ?:A};
sequence seq2 {B:C; ?:B};

gene g {C:A; ?:A,B};

// temporary variables
bool t1 {A:C; C:A; ?:A};
bool t2 {B:C; C:A; ?:B};
// return values
bool r1 {A:; ?:A};
bool r2 {B:; ?:B};

t1 := scan(g, seq1);
t2 := scan(g, seq2);
r1 := declassify(t1,{A:; ?:A});
r2 := declassify(t2,{B:; ?:B});
```

**Fig. 3.** High level code for the example from Sec. 1.1. Companies $A$ and $B$ use a gene-sequencing service provided by $C$. While $A$ has a license agreement with $S$ that allows free access to genes (hence the label allowing $A$ to read `g`), $B$ only is allowed to read the result of the sequencing (and would have to pay for receiving the gene). For an explanation of the annotations c.f. Sec. 2.2.

that the data is correctly computed. As shown in Fig. 3, we combine confidentiality and integrity components in labels, and use the extraction functions $C(L)$ and $I(L)$, respectively.

Following [18, 19] we write $L_1 \sqsubseteq L_2$ whenever $L_1$ is less restrictive than $L_2$. In the case of confidentiality this means that data labelled with $L_1$ allows *more* principals to read it than data labelled with $L_2$, and in the case of integrity this means that code labelled with $L_1$ is trusted by *less* principals than code labelled with $L_2$.

Before any attempt to split the program is made, the compiler verifies that the security labels are statically consistent. This initial verification ensures that all the security annotations are obeyed, for example that variables are not read by principals who lack the right to do so. For a detailed description see [18, 19].

## 2.3 Splitting

Once the program has been verified, the next step is to compute a partitioning of the program onto available hosts in the network. Beside the security annotations in the code, the partitioning is based on the confidentiality and the integrity relation, which each principal specifies. As stated above, together these relations specify the amount of trust a principal has in the other principals in the system. The splitter takes the annotated program and these relations, and produces a split if possible. [18] discuss situations and counter-measures in situations that do not allow a split. The resulting sub-programs are then distributed on the individual hosts, and together implement the original program. In the generated sub-programs each field and statement has been assigned to a principal. In the rest of this section we briefly describe how this split is determined.

*Assigning fields.* In determining, whether a field of the program can be placed at a host $h$ operated by principal $p$, the constraints $C(L_f) \sqsubseteq C_p$ and $I_p \sqsubseteq I(L_f)$ must be satisfied. The equations express that the principal $p$ has at least as much confidentiality as the field $f$ and at most as much integrity as the field. In the example (Fig. 3), the field `seq1` has $C(\texttt{seq1}) = \{A, C\}$ and $I(\texttt{seq1}) = \{A\}$, resulting in host $A$ being the only possible host this field can be scheduled to.

*Assigning statements.* A statement $S$ can be assigned to a principal $p$ if the principal has at least the confidentiality of all values used in the statement. Additionally the principal must have the integrity of all values defined. To ensure this, the two constraint $L_{in} = \bigsqcup_{v \in U(S)} L_v$ and $L_{out} = \bigsqcap_{l \in D(S)} L_l$ are enforced. Here, $U(S)$ denotes all values used in $S$, and $D(S)$ denotes all definitions in $S$. For a principal $p$ to be able to execute $S$, the constraints $C(L_{in}) \sqsubseteq C_p$ and $I_p \sqsubseteq I(L_{out})$ must be satisfied. In the example (Fig. 3), the arguments to the second call to function `scan` have readers $\{A, C\}$ and $\{B, C\}$, resulting in $C$ being the only possible host. A similar result is obtained for integrity, therefore this statement is scheduled on host $C$.

*Declassification.* One problem with security-typed languages is that labels tend to become more and more restrictive in terms of allowed readers and required integrity. To circumvent this, explicit declassification has been introduced, which allows the programmer to specify a new label for a data item. Since a declassification statement is executed with a certain authority, all principals $P$ whose authority is needed must trust the integrity of the execution reaching the declassification statement. This information is recorded in the label of the program counter, so we require $I(pc) \sqsubseteq \{? : P\}$. In the example program declassification is needed since the result of the gene scanning has to be sent back to the individual hosts, allowing them to read the data.

The result of the partitioning with respect to the two trust graphs from Fig. 2 is shown in Fig. 4.

## 2.4 Infrastructure

Before investigating splitting using more dynamic types of trust graphs, we briefly summarise the infrastructure used in our approach. As shown in Fig. 1, the system consists of a set of hosts and a central splitter. The splitter takes requests from clients, that send a program written in a security-typed language, and computes a partitioning of the program on the hosts in the system. The partitioning is guaranteed to obey all ownership and access control annotations present in the program. To compute the partitioning, the splitter uses a trust graph as specified by each client upon entering the network. This trust graph combines the two

```
A: sequence seq1 {A:C; ?:A};         A: sequence seq1 {A:C; ?:A};
B: sequence seq2 {B:C; ?:B};         B: sequence seq2 {B:C; ?:B};

C: gene g {C:A; ?:A,B};              C: gene g {C:A; ?:A,B};

// temporary variables               // temporary variables
C: bool t1 {A:C; C:A; ?:A};          D: bool t1 {A:C; C:A; ?:A};
C: bool t2 {B:C; C:A; ?:B};          C: bool t2 {B:C; C:A; ?:B};
// return values                     // return values
A: bool r1 {A:; ?:A};                A: bool r1 {A:; ?:A};
B: bool r2 {B:; ?:B};                B: bool r2 {B:; ?:B};

C: t1 := scan(g, seq1);              D: t1 := scan(g, seq1);
C: t2 := scan(g, seq2);              C: t2 := scan(g, seq2);
C: r1 := declassify(t1,{A:; ?:A});   D: r1 := declassify(t1,{A:; ?:A});
C: r2 := declassify(t2,{B:; ?:B});   C: r2 := declassify(t2,{B:; ?:B});
```

**Fig. 4.** The result of splitting the example code from Fig. 3. The code on the left has been split using the left trust graph from Fig. 2, the code on the right has been split using the right trust graph in that figure. As a result of adding the host operated by Diana, statements that $A$ has rights on have been partitioned to host $D$.

components *confidentiality* and *integrity* as described in Sec. 2.1. As a result, clients can rely on the fact that their data and code is only stored or executed on hosts that are operated by users who they trust to ensure confidentiality and/or integrity.

## 3 Quality of a Partitioning

The partitioning of the original program as computed by the splitter is guaranteed to obey all ownership- and access control-annotations, as well as the confidentiality and integrity specified by the principals. However, the original framework does not specify how to choose between several possible solutions for the constraint system generated from the program (Sec. 2). This section introduces a metric to judge the quality of a partitioning from the view point of a principal $A$. The idea is to judge the risk of a set of principals violating the confidentiality of data that is owned by $A$ and has been partitioned to a host operated by $B$.

The mathematical foundation for this approach is Maurer's work on probabilistic trust [11], where trust values can be specified by numbers between 0 and 1, representing no and total trust, respectively. If we reconsider the static trust graph introduced in the previous section, we can annotate the confidence and integrity edges with probability 1, since the trust model is complete.

Now assume that a piece of data or code owned by principal $A$ should be scheduled to a host $h$. How would $A$ evaluate whether $h$ is well-suited to host its data, or not. Again there are two components to this question. First of all, $A$

will want to inspect which other principals might allow the splitter to partition their code to $h$. Any other code running on the host might result in an increased risk of $A$'s data being leaked.

To compute the trust that a principal has in this *not* happening, we follow Maurer's formula. For a principal $A$'s view we use his confidentiality graph $CG_A$. Like Maurer we are interested in computing the probability that particular subsets, from which the trust of $A$ in another principal $B$'s confidentiality can be derived, are contained in this view. The probability is

$$P(\nu \subseteq CG_p) = \prod_{S \in \nu} P(S)$$

To compute the overall trust that principal $A$ has in $B$ we use Maurer's formula and get

$$\begin{aligned} \mathit{confidence}(A,B) = & \sum_{i=1}^{k} P(\nu_i \subseteq CG_A) \\ & - \sum_{1 \le i_1 < i_2 \le k} P((\nu_{i_1} \cup \nu_{i_2}) \subseteq CG_A) \\ & + \sum_{1 \le i_1 < i_2 < i_3 \le k} P((\nu_{i1} \cup \nu_{i_2} \cup \nu_{i_3}) \subseteq CG_A) \\ & - \cdots \end{aligned} \tag{2}$$

Obviously, the risk that $B$ leaks the data is the inverse of the confidence value, that is $1 - \mathit{confidence}(A, B)$.

In computing the probability that any of the principals that trust the host $h$ will leak $A$'s data we face the problem that these events are certainly not independent, so that it is hard to compute the probability for the case that this happens. However, the metric just needs to allow to compare different scenarios, but does not need to be a probability. So the first part of the metric is to sum up the inverse of all *conf* values as defined above

$$\mathit{leak}(A,h) = \sum_{p \in C_h} (1 - \mathit{confidence}(A,p))$$

The higher the value of $\mathit{leak}(A, h)$ is, the less confidence has $A$ in other principals that have confidence in $h$.

The second component is the confidence that principal $A$ has in host $h$, that is a measure for how likely it is that $h$ leaks data owned by $A$.

The metrics we suggest is defined as the quotient of these two numbers

$$M(A,h) = \frac{\mathit{leak}(A,h)}{\mathit{confidence}(A,h)}$$

The higher it is, the higher $A$ judges the likelihood, that $h$ will leak its data if stored on $h$. We voluntarily base the metrics only on the confidence values specified by $A$, since we consider the confidentiality of data to be of uttermost importance. One could easily extend the metrics to include integrity, or replace it altogether—our framework allows for easy experimentation with different metrics.

The metrics is used by the splitter whenever there are several hosts that data or code owned by a principal could be partitioned to. For example in the right hand side of Fig. 4 the metrics has been used to pick host $D$ over $C$ since the metrics values are smaller for the former.

## 4 Improving Dynamics

In dynamic distributed systems, the static trust model described in Sec. 2 is insufficient. Complete trust and *non-transitivity* are too simple concepts for most realistic trust scenarios. Additionally, the inability to handle dynamic networks makes the static trust model insufficient for dynamic distributed systems.

In order to make the Secure Program Partitioning applicable to more realistic applications, we extend the trust model in the rest of this paper to first handle dynamic extensions when hosts join or leave the network (Sec. 4.1), and to allow specification of partial or probability-based trust (Sec. 4.2).

Like in the static setting, the centralised splitter in the dynamic setting maintains a global trust graph, which contains trust relations (that is confidence and integrity values) of all principals.

### 4.1 A Dynamic Network

In the dynamic scenario the set of available principals and the trust graph are no longer static. When a principal $p$ joins the network as depicted in Fig. 5, the principal is added to the set $P_{active}$ of active principals. The joining principal informs the splitter of its trust relations, i.e., who it trusts. The splitter asks all the other principals about their confidence and integrity values with respect to the new principal and updates the trust graph with this information. It then decides if the program should be re-split, which happens if the trust of principals into the partitioning can be increased, as measured by the metrics.

When a principal $p$ leaves the network, the program needs to be re-split if the leaving principal was storing data or code of the program. $p \in P_{split}$. The splitter will try to re-split the program. If this is not possible, the program cannot be executed. Execution is halted until the set of active principals, $P_{active}$, again can produce a legal split.

### 4.2 Dynamic Probabilistic Model

In the real world complete trust rarely exists. Therefore a binary trust model as the one proposed in [18] is too simple for many applications.

Instead we use probabilistic trust [11]. In this model, trust edges are annotated with probability values in the range 0 to 1, where 0 is complete distrust, and

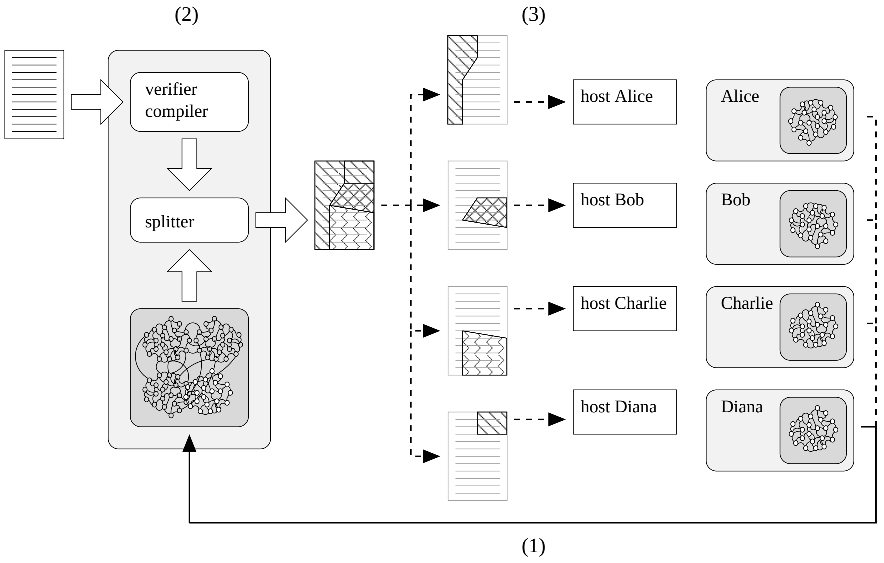

**Fig. 5.** Principal $D$ joins the network. As a result, the trust graph is updated (1), the program is re-split (2), and if the splitting results in new sub-programs, the changed parts are re-distributed (3). In this scenario, part of the sub-program that had been scheduled to host $A$ has been re-partitioned to run at host $D$ (c.f. Fig. 1).

1 complete trust. Trust is divided into confidentiality and integrity. Trust from a principal $A$ to $B$ is expressed as $P(TrustC_{A,B}) = \phi_{T_C}$ and $P(TrustI_{A,B}) = \phi_{T_I}$, where $C$ and $I$ denote confidentiality and integrity, respectively. The probability value essentially states, how probable it is that a principal is trustworthy.

Each principal can then specify what its lowest acceptable confidence level is, allowing principals to influence the safety of their data. A principal's confidence level might vary with how sensitive the information is.

## 4.3 Recommended trust

Another modification to the original trust model is adding *recommended trust.* A principal can declare trust in another principals ability to recommend other principals by $p(RecC_{A,B,i}) = \phi_{R_C}$ and $p(RecI_{A,B,i}) = \phi_{R_I}$. Here $i$ is the maximum allowable recommendation distance. For instance $i = 1$ will allow the principal to recommend only neighbouring principals in the trust graph. This is motivated by recommendations in, e.g., public-key infrastructures. We have left certificates out of our model to keep the presentation simpler, but of course frameworks like [11] can easily be put on top of our approach.

The trust graph now is constructed from a set of trust and recommendation statements, e.g.

$$TG = \{TrustC_{A,B}, TrustI_{,A,B}, RecC_{A,B,1}, TrustC_{B,C}\}$$

The first step of calculating the confidence in a trust statement, e.g., that $A$ trusts in $B$ ($TrustC_{A,B}$), is to find all minimal paths from $A$ to $B$. A minimal path is a path from $A$ to $B$ which is not a superset of any other path from $A$ to $B$. Using minimal paths we define confidence similar to the definition of confidence in [11] using the same idea as in Sec. 3:

$$\begin{aligned} conf(Trust_{A,B}) = & \sum_{i=1}^{k} P(\nu_i \subseteq TG) \\ & - \sum_{1 \leq i_1 < i_2 \leq k} P((\nu_{i_1} \cup \nu_{i_2}) \subseteq TG) \\ & + \sum_{1 \leq i_1 < i_2 < i3 \leq k} P((\nu_{i_1} \cup \nu_{i_2} \cup \nu_{i_3}) \subseteq TG) \\ & - \cdots \end{aligned} \tag{3}$$

Then the probability that a path of trust statements is valid can be calculated by multiplying the probabilities of the individual statements:

$$P(\nu \subseteq TG) = \prod_{S \in \nu} P(S) \tag{4}$$

Our model differs from Maurer's [11] in that it deals with trust in confidentiality and integrity, not certificates. Maurer calculates the confidence in a certificate, while we calculate the confidence in a principal's trustworthiness. In practise this means that recommendations are fundamentally different. In a public key infrastructure a certificate relies on the principal who issued it. If the principal issuing it has an invalid certificate, the issued certificate is also invalid. This dependency is used when calculating a recommendation path.

In our model, a principal might recommend a principal who is trustworthy, even though the recommender is not trustworthy. This means that the recommendation is not dependent on the recommender's trustworthiness.

## 4.4 Example with probabilistic trust model

We now return to our example, now using the probabilistic model. A new principal $F$ is added, and probabilities are added to the trust graph as show in Fig. 6.

All principals require that the confidence in a trust statement must be greater than or equal to 0.90. Now the confidence in the trust statements has to be calculated. To simplify calculations the probabilities of confidentiality and integrity have been chosen identical. Paths in the trust graph starting in $A$, $B$, or $C$, are unique, resulting, e.g., in

$$conf(Trust_{A,C}) = P(Trust_{A,C} \subseteq TG) = p(Trust_{A,C}) = 0.9$$

More interesting is the calculation of $E$'s confidence in $C$ and $D$. There exist two minimal paths $V_1 = \{Rec_{E,A,1}, Trust_{A,C}\}$ and $V_2 = \{Rec_{E,B,1}, Trust_{B,C}\}$.

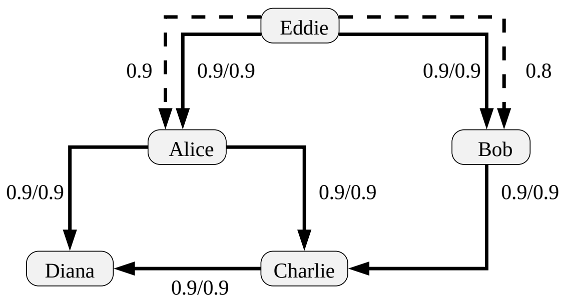

**Fig. 6.** A dynamic trust graph for the example application. Solid edges denote both confidentiality and integrity (we use the same values for both to simplify computations), while dashed edges denote trust in recommendation. The annotations on edges denote the probability measures assigned by principals.

The probability for each path is calculated using (4):

$$P(V_1 \subseteq TG) = p(Rec_{E,A,1}) \cdot p(Trust_{A,C}) = 0.9 \cdot 0.9 = 0.81$$
$$P(V_2 \subseteq TG) = p(Rec_{E,B,1}) \cdot p(Trust_{B,C}) = 0.8 \cdot 0.9 = 0.72$$

Because the paths are disjoint, the combined probability becomes:

$$P((V_1 \cup V_2) \subseteq TG) = P(V_1 \subseteq TG) \cdot P(V_2 \subseteq TG)$$

The confidence can now be calculated using (4):

$$\begin{aligned} conf(Trust_{E,C}) &= P((V_1 \subseteq TG) \vee (V_1 \subseteq TG)) \\ &= P(V_1 \subseteq TG) + P(V_1 \subseteq TG) - P((V_1 \cup V_2) \subseteq TG) \\ &= 0.81 + 0.72 - 0.81 \cdot 0.72 = 0.9468 \end{aligned}$$

Since there is only one path from $F$ to $E$ we get

$$conf(Trust_{E,D}) = 0.9 \cdot 0.9 = 0.81$$

Using these confidence values, the program can be split as shown in Fig. 7. Note that the calculations using $E$'s data has been assigned to $C$ as $0.9468 \geq 0.90$.

## 5 Related Work

Trust and explicit trust management has recently emerged as an important innovation in computer security[5]. Common trust management systems[4, 7, 3] implement a decentralised access control mechanism based on assertions, which may be contained in signed credentials. Assertions allow a principal to prove that

```
A: sequence seq1 {A:C; ?:A};
B: sequence seq2 {B:C; ?:B};
E: sequence seq3 {E:C; ?:E};

C: gene g {C:A; ?:A,B};

// temporary variables
D: bool t1 {A:C; C:A; ?:A};
C: bool t2 {B:C; C:A; ?:B};
C: bool t3 {E:C; C:E; ?:E};
// return values
A: bool r1 {A:; ?:A};
B: bool r2 {B:; ?:B};
E: bool r3 {C:; ?:C};

D: t1 := scan(g, seq1);
C: t2 := scan(g, seq2);
C: t3 := scan(g, seq3);
D: r1 := declassify(t1,{A:; ?:A});
C: r2 := declassify(t2,{B:; ?:B});
C: r3 := declassify(t3,{E:; ?:E});
```

**Fig. 7.** Extended version of the example from Fig. 3, using three client to the splitting service. The partitioning of the program is performed using the trust graph from Fig. 6.

she is sufficiently trusted to be granted certain privileges. Recent trust management frameworks, e.g., Trustbuilder[17], extend this access control framework with a mechanism that gradually discloses credentials in order to build the necessary level of trust without compromising the privacy of either party [9, 16]. Security mechanisms based on the human notion of trust has also been proposed for pervasive computing[6, 10]. We note that all these systems implement functionality that is orthogonal to our notions of dynamic trust graphs. We can easily extend our system (and enhance the usability) by adding almost any of these tools.

Regarding the splitting framework we have already covered the related work as we rely on Myers work [13, 14, 18, 19] as the underlying foundation for our prototype implementation.

## 6 Conclusion and Future Work

We have presented an extension of a framework for secure program partitioning with a notion of dynamic trust and shown the results that this dynamics has on the computed partitioning. Given the increasing reliance on open computing environments, there also is an increased need for spontaneous and dynamic formation of short term coalitions to solve a specific problem. Our extensions make automatic partitioning of annotated code a viable technique to prevent violation of confidentiality and integrity in applications that execute on a widely distributed computing platform, e.g., a computational grid infrastructure. While the partitioning is already provided by the original infrastructure, only the added dynamics of network and trust make the technique applicable in this setting.

Another important addition of our work is the development of a metric to judge how well a partitioning of data or code to a specific host fulfils the requirements of a principal—on top of the guarantee that the sub-programs obey the annotations as well as the confidentiality and integrity specifications by construction. This allows a *qualitative* assessment of a partitioning on top of automated guarantees.

We plan to use our prototype implementation to experiment with more dynamic trust models and investigate more closely the interdependence between our metric, the computed partitioning, and the trust model. Another research avenue is to look closer at how to model *availability* as a measure of trust and specification, and to add some of the tools that allow trust management on top of our framework.

## References

1. M. Abadi, J. Feigenbaum, and J. Kilian. On hiding information from an oracle. *Journal of Computer and System Sciences*, 39(1):21–50, Aug. 1989.
2. G. R. Blakley and C. Meadows. A database encryption scheme which allows the computation of statistics using encrypted data. In *Proceedings of the 1985 Symposium on Security and privacy (SSP '85)*, pages 116–122, Los Angeles, Ca., USA, Apr. 1990. IEEE Computer Society Press.
3. M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote trust management system. Internet Request for Comment RFC 2704, Internet Engineering Task Force, Sept. 1999. Version 2.
4. M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The role of trust management in distributed systems security. *Lecture Notes in Computer Science*, 1603:185–210, 1999.
5. M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, May 1996. IEEE Computer Society, Technical Committee on Security and Privacy, IEEE Computer Society Press.
6. V. Cahill, E. Gray, C. J. J.-M. Seigneur, Y. Chen, B. Shand, N. Dimmock, A. Twigg, J. Bacon, C. English, W. Wagealla, S. Terzis, P. Nicon, G. di Marzo Serugendo, C. Bryce, M. Carbone, K. Krukow, and M. Nielsen. Using trust for secure collaboration in uncertain environments. *IEEE Pervasive Computing*, 2(3):52–61, July–Sept. 2003.
7. Y.-H. Chu, J. Feigenbaum, B. A. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust management for web applications. *Computer Networks*, 29(8-13):953–964, 1997.
8. European Parliament. Directive 95/46/ec of the european parliament and of the council of 24 october 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data, October 1995.

9. V. E. Jones and W. H. Winsborough. Negotiating disclosure of sensitive credentials, Dec. 09 1999.
10. L. Kagal, T. W. Finin, and A. Joshi. Communications - trust-based security in pervasive computing environments. *IEEE Computer*, 34(12):154–157, 2001.
11. U. Maurer. Modelling a public-key infrastructure. In *ESORICS: European Symposium on Research in Computer Security*. LNCS, Springer-Verlag, 1996.
12. A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.
13. A. C. Myers and B. Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.
14. A. C. Myers and B. Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.
15. D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.
16. W. Winsborough, K. Seamons, and V. Jones. Automated trust negotiation. Technical Report TR-2000-05, Department of Computer Science, North Carolina State University, Apr. 24 2000. Mon, 24 Apr 2000 17:07:47 GMT.
17. M. Winslett, T. Yu, K. E. Seamons, A. Hess, J. Jacobson, R. Jarvis, B. Smith, and L. Yu. The trustbuilder architecture for trust negotiation. *IEEE Internet Computing, volume 6, number 6*, pages pages 30–37, 2002.
18. S. Zdancewic, L. Zheng, N. Nystrom, and A. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Symposium on Operating Systems Principles*, pages 1–14, 2001.
19. S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Secure program partitioning. *ACM Transactions on Computer Systems*, 20(3):283–328, Aug. 2002.