Real-Time Verification of Integrity Policies for Distributed Systems

Ernesto Buelna*[1], Raúl Monroy[2]

[1] Microsoft Consulting Services
Microsoft México
México, D. F., México
*Ernesto.Buelna@microsoft.com
[2] Instituto Tecnológico y de Estudios Superiores de Monterrey,
Campus Estado de México, Estado de México, México

ABSTRACT
We introduce a mechanism for the verification of real-time integrity policies about the operation of a distributed system. Our mechanism is based on Microsoft .NET technologies. Unlike rival competitors, it is not intrusive, as it hardly modifies the source code of any component of the system to be monitored. Our mechanism consists of four modules: the specification module, which comes with a security policy specification language, geared towards the capture of integrity policies; the monitoring module, which includes a code injector, whereby the mechanism observes how specific methods of the system, referred to by some policy, are invoked; the verifier module, which examines the operation of the distributed system in order to determine whether is policy compliant or not; and, the reporter module, which notifies the system is policy compliant, or sends an alert upon the occurrence of a contingency, indicating policy violation. We argue that our mechanism can be framed within the Clark and Wilson security model, and, thus, used to realise information integrity. We illustrate the workings and the power of our mechanism on a simple, but industrial-strength, case study.

Keywords: real-time policy verification, integrity policies, distributed systems, .NET code injection, information security.

RESUMEN
En este artículo, presentamos un mecanismo para verificar, en tiempo-real, políticas de integridad asociadas con un sistema distribuido. Nuestro mecanismo está desarrollado en, y es aplicable a, tecnologías MS .NET. A diferencia de sus competidores, no es intrusivo, pues no requiere modificaciones de consideración en el código fuente de ningún componente del sistema a inspeccionarse. Nuestro mecanismo consiste de cuatro módulos: el módulo de especificación, diseñado especialmente para expresar políticas de integridad; el módulo de inspección, el cual incluye un inyector de código, a través del cual nuestro mecanismo observa cómo se invocan los métodos del sistema referidos en alguna de las políticas; el módulo verificador, el cual examina la operación del sistema para determinar si éste cumple o no con las políticas; y, el módulo reportador, que notifica si el sistema está conforme a la norma, o envía alertas en el caso que ocurra una contingencia, indicando la violación a una política. Demostramos que nuestro mecanismo formalmente se enmarca dentro del modelo de seguridad de Clark y Wilson, y, por lo tanto, puede aplicarse en el establecimiento de integridad. Para mostrar el funcionamiento y las bondades de nuestro mecanismo, incluimos un caso de estudio simple, pero típico de una aplicación industrial o comercial.

1. Introduction

The malfunctioning of a company's information system, due to an attack, a code or a design flaw, or a hardware failure, heavily impacts the company prestige and finance. Not surprisingly, (large) companies usually assign a (considerably) large budget to develop, debug, test, and maintain their information systems, in an attempt at timely detecting, and subsequently correcting a system flaw. Thus, to minimise risks, a lot of testing is conducted, both before and after the system has been deployed into production.

Testing, which used to be geared towards validating that a system complies with specific business functions, nowadays contemplates guarantees that the system will meet a security policy, and, hence, that it will not be misused to gain access to an organisation's asset. Guaranteeing that, at any point in time, a system will behave as expected, and obey to all business rules, is far from trivial. The problem is further magnified when the system is to interact with others, especially when they are distributed amongst several different companies, and possibly over one or more countries. What is more, although business rules often include security checks (for example, an Internet bank system will require that no user be allowed to take money from another user's account), they hardly consider the system, when run in parallel with others. This may give rise to a hazard, especially when several systems must cooperate one another in order to achieve a business function.

Unfortunately, the study and development of mechanisms for monitoring that a system complies with a given security policy has been largely ignored: emphasis has centred on detecting system misuse. For example, there are not commercial tools that take a security policy, specified at a high-level of a business function (or process), and then monitor a target system, so that, at real-time, they are able to report whether, so far, the system adheres to the security policy. Although existing tools can be adapted to do this task, they require quite a lot of development effort, and are not suitable for rapidly prototyping an inspection mechanism. What is more, as shall be discussed later on in the text, the mechanisms that have been proposed in the literature for this or a similar purpose are rather intrusive, in that they require heavy modifications in the underlying pieces of software, some such changes might not be realisable, e.g., for licenses' constraints.

In this paper, we present a mechanism that automatically alerts whenever a system is not complying with a (integrity) security policy. Unlike others, our mechanism does not interfere with or rely on the design or the implementation of the system under inspection. We also provide a security policy specification language, especially designed for capturing safety and liveness properties related with business functions. Policies are built out of predicates, where the domain of discourse contains parameters (values) of procedures or public methods of the system to be monitored, and that can be specified without knowledge of the system implementation; furthermore, policies might refer to events occurring throughout several systems, which the system under inspection communicates with. Although applicable in any context, our mechanism has been developed to work with Microsoft .NET distributed systems, since this framework allows us to automate most of its deployment.

Our mechanism works by dynamically injecting Microsoft intermediate language code on specific modules of the system to be monitored. Each of these small programs, we call an observation point, is responsible of intercepting any invocation to a public method referred to by the designated collection of policies, as well as reporting the method execution parameters to a distributed database module. A security policy verification engine is then used to confront the information collected in the database against the security polices, specified in a high-level policy specification language. Finally, the mechanism viewer is used to display the validation results, send audit reports, and alert a security officer upon any contingency. The mechanism viewer also displays rules that have been triggered as expected, thus, indicating the system under monitoring is policy compliant.

In order to illustrate the workings, the strength, and the functionality of our mechanism, we include and go through a simple, but industrial-strength case study, in the field of value-added chain, and associated with the processing of a purchase order. In particular, for this experimental test, we have: (i) developed a simulator for a typical purchase order workflow; (ii) identified and specified eleven integrity security policies; and, finally, (iii) simulated attacks to the purchase ordering process, in order to validate if any policy violation was reported through our monitor. In our simulations, attacks were introduced on a random basis. Our mechanism was successfully tested: our experimental results show that each policy violation was spotted timely, and accordingly.

Overview of Paper The remaining of the paper is organised as follows: In Section 2, we provide an overview of both information integrity and security policies, and the security model used to theoretically justify the design and the workings of our mechanism. In Section 3, we discuss related work, identifying key competitor mechanisms. Then, in Section 4, we introduce our mechanism, outlining the internal workings of all its components. Section 5 provides an overview of our case study, specifying: the problem under consideration, the security policy, the mechanism deployment, and the simulation environment. Section 6 is a follow up to Section 5, introducing the testing attack scenarios used in our validation experiments, as well as outlining the associated results. Finally Section 7 concludes the paper, and gives directions for further work.

2. Preliminary Concepts

The aim of this section is twofold. First, it aims to provide an overview of security policies, given special attention to integrity policies, the kinds of policies we would like to automatically inspect and verify for system compliance. Second, this section also aims to outline the integrity model of Clark and Wilson, which provides a formal framework to our mechanism for verifying integrity security policies.

2.1 Security Policies

Security policies are rules or conventions that aim to protect an organization's assets [1]. Following [2], a security policy is defined in terms of a set of predicates, Π, and a set of execution traces, T, each of which is a, possibly infinite, sequence of actions over a computer system (e.g. instructions, states or invocations to methods or procedures, etc.) T satisfies Π if and only if $\forall P \in \Pi.\ \tau \in T.\ P(\tau)$ holds. Notice that each individual execution trace, $\tau \in T$, is verified against every security policy in an isolated manner; i.e., without correlating $\tau \in T$ with some other $\tau' \in T$ [3].

2.2 Integrity Policies

In the early years, computer security centred on confidentiality, so as to prevent the disclosure of critical information to an unintended party. Then, a number of mechanisms for realising confidentiality policies were suggested and thoroughly studied, including mechanisms for authenticating users, controlling the access of users to resources, and for generating audit information. In this vein, for example, the United States Department of Defense developed criteria for evaluating whether a computer system is secure. These criteria were collected in a manual, better known as the Orange book. Although created in a military environment, the Orange book is an important asset for general security.

On a business context, integrity is as paramount as confidentiality. Yet, what integrity means has not a unique answer. For example, integrity of an item could mean that the item is accurate, correct, unchangeable, or changeable, but only in a predefined way or by a collection of authorised users, etc. [4]. There are three general, prevailing aspects to integrity, though: protection of resources, detection and correction of errors, and separation of duties.

In this, paper, integrity policies are used to specify the correct behaviour of a distributed system, in accordance with the three general aspects above mentioned. Our policies are therefore normative: they specify what ought to happen, so that we can validate whether system operation is policy compliant or not. One of the main challenges of specifying these kinds of policies, though, lies in the feasibility of creating a global model of a distributed system, where we have several processes, distributed over some region, and that are independent one another. Then, verifying that the system is policy compliant has to consider all these processes. The usual way to approach system communication is through system services, which, when correctly designed and implemented, do not expose the internal workings of the calling service. In the context of MS .NET technology, these services usually are implemented through a collection of public methods, with designated parameters.

Below, we outline the Clark and Wilson model for integrity, which we shall use to formally frame our mechanism for ensuring that a system adheres to an integrity security policy.

2.3 The Clark-Wilson Integrity Model

Clark and Wilson, see [5], created a security model, aimed at preserving the integrity of information. The model is composed of nine rules, given in terms of the following elements:

Transformation Procedure, TP, which is responsible for manipulating information, using well-formed transactions.

Constrained Data Item, CDI, which is data that must abide with the integrity model, and be handled exclusively by TP's.

Unconstrained Data Item, UDI, which is data not restricted to comply with the integrity model.

Integrity Verification Procedure, IVP, which is responsible for verifying that every CDI conforms to the specification.

Under Clark-Wilson, CDI’s should be handled exclusively by TP’s, and TP’s should perform only well-formed transactions; a well-formed transaction consists of a sequence of operations, each of which makes a system go from a consistent state to another, which is also consistent. Verifying that a system is in a correct state is performed by a rule of integrity, implemented in an IVP. It follows, by induction on time, that, if an IVP certifies the integrity of the state of a system at some initial time, t0, then, integrity is ensured for any subsequent time, tn, where tn> t0, as long as the same IVP still runs and verifies the system at that time, tn [5].

Ensuring the integrity of a system is achieved by means of two sets of rules: certification (C), and insurance (E). C rules may require human intervention, while E rules are implemented as part of the system. The security task is divided into two steps. In the first step, rules, whether C or E, are used to ensure the internal consistency of CDI’s:

- C1. IVP’s must ensure that CDI’s are in a correct state.
- C2. TP’s must ensure that, after execution, CDI’s end up in a correct state.
- E1. At any time, the system must keep a list specifying which TP’s are allowed to manipulate every CDI, emitting the associated certificate. The system must involve access control to guarantee that a TP can change only CDI’s for which it has been certified.

By contrast, in the second step, rules are used to ensure external consistency of CDI’s:

- E2. Extend the list relating TP’s and CDI’s so as to include the ID’s of the users authorised to execute such TP’s.
- C3. Ensure that the list relating ID’s, TP’s and CDI’s, see rule E2, abides by the principle of separation of duties, which prevents frauds and errors by distributing, amongst several users, the tasks, together with their associated privileges, for carrying out a critical business process [6].
- E3. The system must authenticate every user attempting to execute a TP.

The feasibility of verifying the above rules requires an extra rule:

- C4. Every TP must log enough information so that it is possible to reconstruct every transaction on an append-only CDI.

The rule below indicates how IDU’s are loaded into a system:

- C5. Any TP, which takes an UDI as input, must perform only valid transformations on it, thereby converting the UDI into a CDI, or else reject it.

Finally, to complete the set of rules, Clark-Wilson also demands that users can execute only certified TP’s:

- E4. Ensure that the certification list, see E2 above, is modifiable only by certification servers, and that these servers cannot run any of the TP’s in that list. Notice how this clause specifies that certification servers are not system users themselves.

Although Clark-Wilson defines a complete set of rules for guaranteeing integrity, in an actual system, it is impractical to manually perform all the model certification processes. This is both because certification easily gets complex, and because processes themselves change continuously. The solution of this paper is aligned to the Clark-Wilson model, automating partially some of the certification rules.

3. Related Work

There are several tools in the market and mechanisms in the literature that aim to help organisations detect and resolve system failures at production time. As for tools, closely related to our mechanism is Microsoft System Center Operations Manager (SCOM) © [7], which monitors the correct operation of system software and hardware infrastructure. SCOM works by deploying software agents on the computers where system activity is to be inspected. After gathering information from log sources, the agents try to identify failure applying a set of predefined rules. Installing SCOM requires that the system to be monitored be modified at the source code, thus imposing a strong delay in the deployment of the system into production; this delay is further magnified when the system is part of a larger one. By contrast, our mechanism does not require the system source code to be manually or heavily modified, or developed in a special way (other than within .NET Technology). Further, a team, different from the development one, can apply our mechanism, thus separating duties and enabling a certification process for the inspection part (c.f. C2).

An Information Security Manager (ISM), such as OSSIM, or Cisco's MARS ©, can be adapted to perform integrity verification. However, this usually involves developing extra code, particularly plug-ins, or modifying the system source code, so that the ISM is able to gather security events. What is more, as are designed towards detecting intrusions or violations to security policies, the use of the ISM has also to be adapted so that it is able to tell whether, so far, the execution of the business function is up to the security standard.

On the academy side, our work is closely related to that of Liu et al. [8]. Given that testing large-scale distributed systems is challenging, because errors may appear upon a combination of system, or network failures, Liu et al. developed D3S, a technique for debugging a distributed system, which already is in production.

Using D3S, it is possible to specify and test a desirable property. Upon property violation, D3S outputs the sequence of states previous to the violation state, allowing developers to quickly spot the source of failure. In contrast with our mechanism, D3S system developers are required to write the predicates, and to know quite a lot of low-level system's implementation details; what is more, the approach is oriented towards finding flaws, rather than looking for integrity security policy violations. D3S is expensive, because it is difficult to maintain. This difficulty stems both from the extra effort required in the development phase, and from that any change to the security policy (or to the system code) requires a good deal of programming effort, due to the lack of metadata in the deployed code. By way of comparison, in our mechanism, an integrity policy can be implemented following a certification process, and so be conducted by staff other than the system developers. The certifying staff does not need to know system's implementation details.

4. A Mechanism for Verifying Integrity Security Policies

Figure 1 depicts a static diagram of our mechanism, identifying its main working elements, namely: the policy specification module, the generic code injector, the monitor, the policy verifier, and the reporter.

Below, we shall detail all of our mechanism's components in turn, considering that they all communicate one another across a trusted network. That is, we consider that our mechanism has been secured following Clark-Wilson and general recommendations for physical security.

Notice that, in particular, in accordance with rule C4, observation points can only append records on to the system logs, but cannot read them, or modify them in any other way, and that, in accordance with rule E4, the policy verifier can read the system logs, but not modify them.

4.1 Policy Specification Module

In our approach to the verification of integrity security policies, the system under study executes, possibly in cooperation with others, a collection of critical business functions. To verify it is policy compliant, each business function should be abstracted out by means of a suitable modelling formalism, such as information workflow, process calculi, etc. Then, every associated policy must be specified, also using a suitable formalism, as a property of the model, in terms of key elements.

In our case, we model a system in terms of the discrete events it may ever perform. So, we might use an automaton, a workflow, a process, or another similar formalism, to serve this purpose. Then, we set policies to be properties over a sequence of events, which may hold along a pre-set period of time. Each property can be of either two types: *safety*, which are used to specify that, during the execution of a process, something bad will never happen; or *liveness*, which are used to express that something good will eventually do. Now, we set events to be observable actions of the systems under inspection; each event corresponds to a service that the system executes. Thus, our policies refer to the invocation and completion of the services (public methods in our case) of the system, but not to any services' local variable.

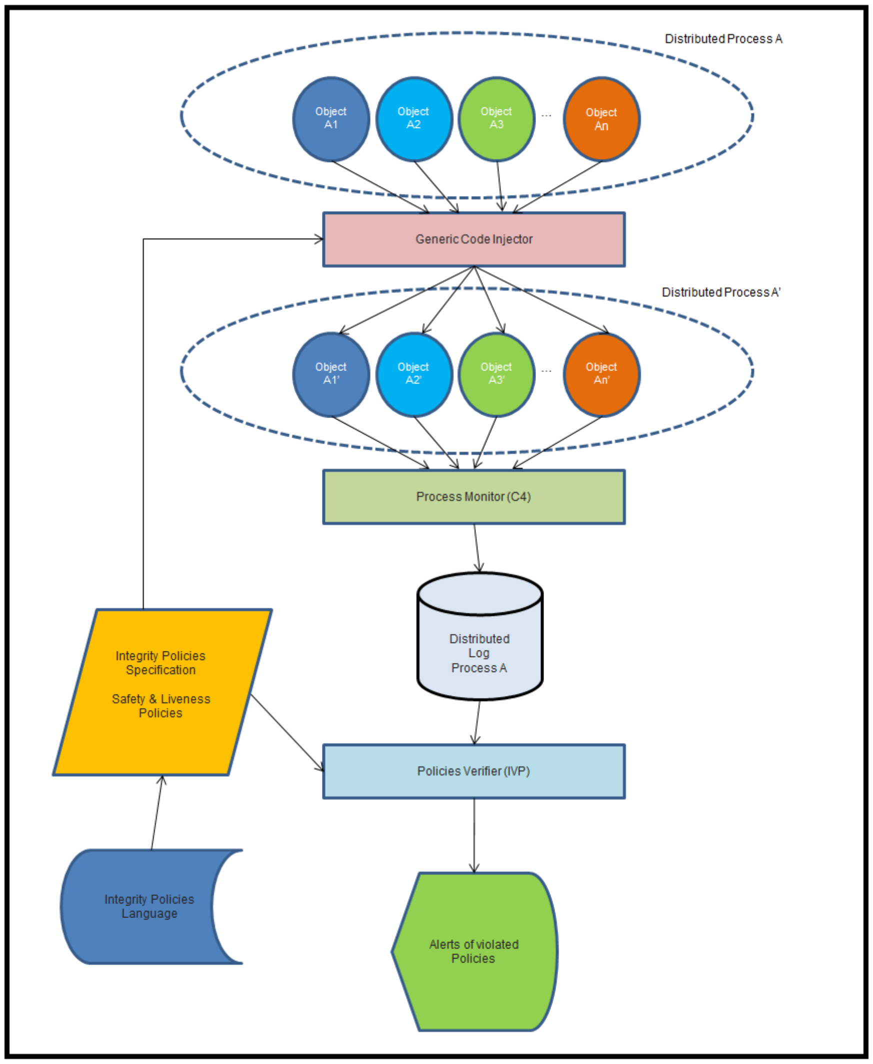

Figure 1. Mechanism for the verification of integrity security policies

More specifically, each event is formed using the caller identity, the callee identity, the public method name, the public method parameters, and the associated parameter values. As is standard in the literature, we work under the assumption that, for synchronising the distributed system, there has been implemented a global clock, such as the network time protocol, and that each intervening process has a unique global identifier.

Each policy is of the form *Precondition* → *Policy*. *Precondition* is a formula, written in our policy language, that the system(s) must satisfy for *Policy* to be applicable. *Policy* is the policy itself, also given by means of a formula, describing the conditions that are necessary in order to assess whether the policy is enforced by the system. Formulae are built out of constant services, using logical “Or” or “And”. The grammar of our integrity policy language appears in Figure 2.

4.2 Generic Code Injector

As the name suggests, the generic code injector modifies a system service, inserting small pieces of code, collectively called an *observation point*, through which the modified service is able to report any invocation or completion of the method. The generic code injector modifies only service methods that are referred to by at least one of the designated integrity security policies. Since the system under monitoring has been developed under .NET technology, the generic code injector changes the methods of interest directly in the assemblies, adding the observation point in an intermediate language code. This is achieved fully automatically, and so that, other than the added reporting capability, each modified method works as before, and thus satisfies every property it used to.

4.3 The Monitor and the Policy Verifier

The Monitor process gets and stores all the information sent by every observation point. The information collected is kept in a generic database, which is composed of as many tables as the number of methods under monitoring. Method tables are automatically created, directly from the set of designated policies. If monitoring covers all the methods involved in the set of policies, our mechanism will fully comply with the C4 rule of Clark-Wilson, since the observation points, our TP's, log all service invocations along with the values of the input and output parameters, so that it is possible to reconstruct the operation of the system at any given time.

The policy verifier plays the role of the Clark-Wilson IVP (Integrity Verification Procedure), which is responsible for validating that system operation conforms to the integrity policy. It takes two inputs: one is the information stored in the monitor database, and the other is the set of designated policies. Using the semantics of the policy specification language, the verifier automatically translates each policy into a collection of Microsoft SQL server queries, which it then checks against the database to determine whether or not we are facing a policy violation. Figure 3 depicts all the steps required for implementing the integrity policy verifier.

4.4 The Viewer

As the name suggests, the viewer display which rules have been triggered, and which have failed to do so, considering a given a time limit. In the case of a rule violation, it sends specific, and hopefully noticeable alerts, indicating the rule that has been violated, and other complementary information. These notices will help the system administrator to start the appropriate emergency response procedure.

We are now ready to look into a case study, through which we aim to convey how to use, our integrity security policy verification mechanism, as well as the mechanism strengths and limitations.

```
Goal   ::= “Policy” Identifier  PolicyType Policies
PolicyType ::=”Safety” | “Liveness”
Policies ::= Policy | Policy ”;” Policies
Policy ::= Precondition “→” Condition
Precondition::= Condition
Condition ::=Predicate | Predicate LogicalOperator
Condition
LogicalOperator ::= “And”|”Or”
Predicate ::= Operand BinaryRelation Operand | “(“
Predicate “)”
BinaryRelation ::= “==”|”<=”|”>=”|”!=”|”<”|”>”
Operand   ::= StaticProperty |
ParameterObject “Where” Condition|
Parameter|
Function|
InternalFunction  |
AccomplishedPolicy |
InternalConstant  |
“(“ Operand “)” |
String |
Number
ParameterObject ::=  “[“ Object ”.” Parameter ”]”  |
                       FunctionObjects ”(“
ParameterObject”)”
Object ::=  Identifier
DatabaseField ::=  “[“Server ”.” Table ”.” Field ”]”  |
FunctionObjects::=  “Count”
Parameter ::= Identifier
InternalFunction ::= InternalOperator ”(“ Operands”)”
AccomplishedPolicy ::= “PolicyCumplida” ”(“ PolicyName
”@” Operands “)”
PolicyName ::= Identifier
InternalOperator ::= “::T0”|”::T1”|”::Id”|
Function ::= Identifier “(“ Operands ”)” |
                 Operand ArithmeticOperation Operand |
                 “(“ Function “)”
InternalConstant   ::= “null” |“true”  |“false” |“this”
ArithmeticOperation ::=  “+” | “-“ | “*” |”/”
Operands ::= Identifier | Identifier”,”Operands
StaticProperty ::= “[“ Identifier ”]”
String ::= “’” Identifier “’”
Number ::= NumericChar | NumericChar Number
Identifier ::= AlphaChar| AlphaChar AlphaId
AlphaId ::= “.” | Character | Character AlphaId
```

Figure 2. Grammar of integrity policy language

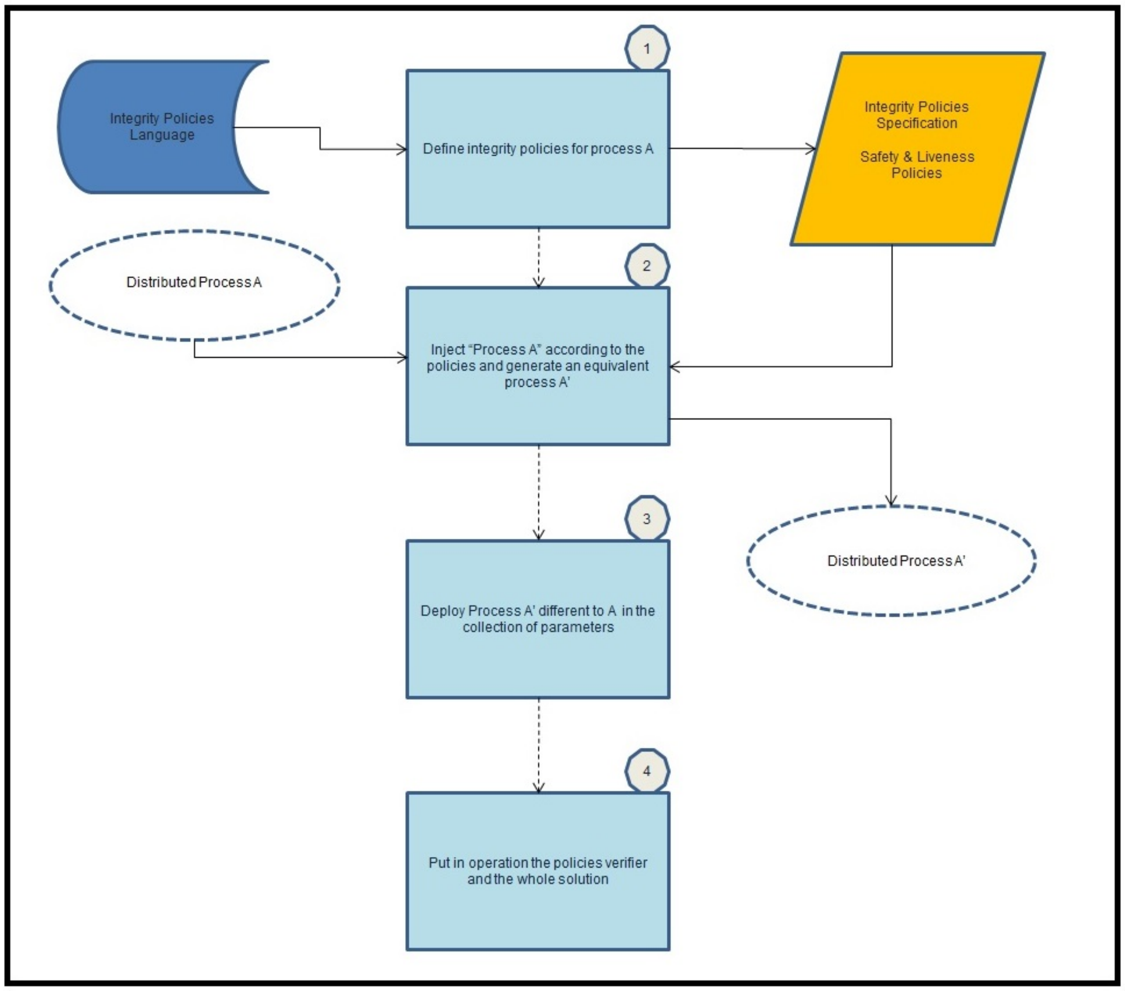
Integrity Policies Language
Define integrity policies for process A
1
Integrity Policies Specification
Safety & Liveness Policies
Distributed Process A
2
Inject "Process A" according to the policies and generate an equivalent process A'
3
Distributed Process A'
Deploy Process A' different to A in the collection of parameters
4
Put in operation the policies verifier and the whole solution

Figure 3. Flow diagram for the integrity policy verifier

5. Case Study

In this section, we introduce a simple, yet illustrative and significant case study, involving purchase order management. Through it, we intend to illustrate how to specify high-level integrity security policies, how to deploy the mechanism for inspection, and how the verification proceeds in order to spot policy compliance, or violation.

5.1 Purchase Order Management: the Simulation of a Distributed Process

When setting a manufacturing order, a client initiates a complex process that may involve steps ranging from the transformation of raw materials, to the shipping of a product. At each step of the process, value is added, including the refinement of raw materials, the transportation of a product, etc. The process ends when the product has been fully manufactured, sold, and shipped to the end user. These types of processes are called value-added chain, because at every process step, the product is transformed, or distributed in some way [10].

Purchase order processing is a distributed, value-added chain process. To characterize it, we have used a workflow, which consists of a collection of interconnected processes, each of which pertains to work done by either a person, or a process [10], and, so, can be represented by one or more interconnected automata. Roughly speaking, purchase order management is about orchestrating the interaction amongst two separate processes, one called *the buyer*, and the other *the supplier*. In a first step, the buyer issues a purchase order to the supplier, which then performs a few checks. Upon purchase order authorisation, the supplier confirms the sale order to the buyer. Goods agreed upon the purchase order are to be delivered to a specified address timely, but purchase order cancellations may occur. The general workflow of purchase order is given in figure 4. There, and in what follows, PO stands for Purchase Order, D for Delivery, and DO stands for Delivery Order.

Using .NET, we have implemented the general functionality of both the buyer and the supplier, considering two separate processes; in particular, both processes are given in terms of high-level business operations, such as putting a purchase order, cancelling a delivery order, etc. The buyer (respectively, the supplier) has an interface, through which it gets (respectively, puts) messages from (respectively, to) other system components; both processes have a component that coordinates the execution of the workflow.

Also, we have implemented a simulation environment, which parameterised with several rates, involving the number of purchase orders issued per hour, the number of failures (respectively, cancellations, communication or message format errors, etc.) that occur per purchase order, and so on, and so forth. Notice that policies are verified against the actions of each process, so we assume that both buyer and supplier are able, in practice, to produce a set if integrity security policies, and, ask a third party to certify that its process works correctly and that is policy compliant. We stress that the certifier would not need to have access to the source code, that it should know only how processes and components are (and methods) involved.

All the pieces of code developed for this case study, as well as the specification module, the generic code injector, the policy verifier, and the viewer, can be obtained by sending electronic mail to any of the paper authors.

5.2 Policies

Associated with purchase order management, there are two major integrity security policies, safety and liveness:

- (Safety) Bad events, such as deadlocks, or livelocks, may not ever happen; a means for deadlock (respectively, livelock) recovering must be included for making the purchase order system robust; and

- (Liveness) Good events must eventually happen. Basically, this means that purchase order runs as expected; that is, either it terminates with success (product delivery), or failure (purchase order cancellation).

Below appear the eleven integrity security policies defined for purchase order management, given in our policy specification language. We emphasise that these policies are all automatically monitored upon application, as an observation point is inserted into any method that refers to an object that is also referred to by the policy.

1-3. The first three policies are safety. They state that serving an order (of type either purchase confirmation, delivery confirmation, or delivery) cannot be postponed for more than a given amount of time, as otherwise there is an impact on the system performance. A violation to these kinds of policies may indicate the occurrence of a (internal) flaw, or an (external) error event. We provide the specification of the “wait for delivery” policy below; the others are just as similar.

```
Policy DeliveryNotReceivedOntime Safety
       DOIdentifier!=null → ( ::T1(this) -
::T0(this) ) > [DO.MaximunWaitTime]
       Where [DO.DOIdentifier]== DOIdentifier
```

4. (Liveness policy) Every PO which has been authorised by the supplier, but which has not been cancelled, must eventually either get cancelled, or be associated with one and only one DO.

```
Policy
AuthorizedPOEventuallyGeneratesDOorGotCan
celled Liveness
          PO.POIsAuthorized==true →
           1    ==  Count([DO.DOIdentifier])
              Where
                  [DO.POIdentifier]==
PO.POIdentifier
                  Or
                  (
                    [PO.Status]=='Cancelled'
Where
[PO.POIdentifier]==PurchaseOrder.POIdentifier
                  )
```

5. (Liveness policy) A PO, which has a DO, must eventually both reach the state "Delivered" in the context of the supplier, and cause the buyer to reach the state "Claim D" or "Close PO".

```
Policy DispatchedDOIsclosedOrclaimed
Liveness
        DeliveryOrder!=null  →  true  ==
[DO.OrderDelivered]
          Where
             [DO.POIdentifier]==
DeliveryOrder.POIdentifier
              And
              (
 IsSatisfiedPolicy(

ClaimDelivery@DeliveryOrder.POIdentifier )==
true          Or
 IsSatisfiedPolicy (

ClosePO@DeliveryOrder.POIdentifier)== true
              )
Policy   ClaimDelivery Liveness
          POIdentifier!=null→ true

Policy  ClosePO Liveness
          POIdentifier!=null→ true
```

6. (Safety policy) A PO that was either rejected or cancelled cannot be associated with any DO, as this could lead to duplicate delivery orders, indicating a bug in the system, or that the system has been tampered with.

```
Policy  PORejectedOrCancelledWithoutDO
Safety
         PurchaseOrder!=null→
             1 >  Count([DO.DOIdentifier])
              Where
              [DO.POIdentifier]==
              PurchaseOrder.POIdentifier
```

7. (Liveness policy) A PO, which has been deemed invalid in the context of the supplier, must eventually taken to has been rejected in the context of the buyer.

```
Policy  InvalidPORejected Liveness
          PurchaseOrder.IsInvalid==true  →
          'Rejected'  == [Buyer.PO.Status]
Where
           [PurchaseOrder.POIdentifier]==
           PurchaseOrder.POIdentifier
```

8. (Liveness policy) A PO that has been accepted by the supplier shall be processed in the context of the buyer, eventually reaching the state "Wait DO for PO".

```
Policy  ProcessAcceptedPO Liveness
          PurchaseOrder.Status=='Received'  →
IsSatisfiedPolicy(WaitDOforPO@PurchaseOrder
.POIdentifier )== true
```

9. (Safety policy) Global processing time, at both partners, must match.

```
Policy  POProcessedBeforeGenerated Safety
          PurchaseOrder!=null→
          PurchaseOrder.ProcessDateTime  >
[Buyer.PO.ProcessDateTime]
Where
         [PO.POIdentifier]==
          PurchadseOrder.POIdentifier
```

10. (Liveness policy) An unauthorised DO will cause the PO to be cancelled in the context of both the buyer and the supplier.

```
Policy  DONotAuthorizedOrCancelled Liveness
        PurchaseOrder.Status=='NotAuthorized'
→
        IsSatisfiedPolicy
(POCancelledAtBuyer@PurchaseOrder
.POIdentifier )== true
   And
        IsSatisfiedPolicy
(POCancelledAtSupplier@
PurchaseOrder.POIdentifier)== true
```

11. (Safety policy) Every Delivery must be associated with a PO, and a DO, in the context of both the buyer and the supplier.

```
Policy  DeliveryWithoutPOorDOAtBuyer
Safety
 PurchaseOrder!=null  →
           1    =!
           Count([Buyer.DO.DOIdentifier])
            Where
              [Buyer.PO. POIdentifier]==
               PurchaseOrder .POIdentifier
            Or
            1     =!
            Count([Buyer.DO.DOIdentifier])
             Where
              [Buyer.DO.POIdentifier]==
               PurchaseOrder.POIdentifier

Policy  DeliveryWithoutPOorDOAtSupplier
Safety
 PurchaseOrder!=null  →
            1     =!
            Count([Supplier.DO.DOIdentifier])
            Where
           [Supplier.PO.POIdentifier]==
            PurchaseOrder .POIdentifier
         Or
            1    =!
            Count([Supplier.DO.DOIdentifier])
            Where
           [Supplier.DO.POIdentifier]==
            PurchaseOrder.POIdentifier
```

The code injector takes all these policies and automatically injects the appropriate observation point, looking into the system components' specification. Notice that, since we are working within the context of .NET technology, automation is possible since the name of the objects referred to by one of the policy rules must appear readily in the interface of some or several public methods.

5.3 Results of Simulated Experiment

Using the simulation environment, we set several attack scenarios, which would violate one or more of the stated 11 integrity security policies. Policies violations were all timely detected, as expected. In what follows, we describe the strategy, in terms of actions or events, which we have followed in order to realise a policy violation (we omit figures, such as actual purchase order rate, timeout settings, etc., as they do not add significant information, when considering the overall action effect).

1. Buyer issues several purchase orders, but supplier is down, thus, causing that no purchase order confirmation arrives in time to the buyer.

2. Denial of service attack: buyer is set to issue purchase orders at an unexpectedly large rate, causing the supplier to be flooded by the number of purchase order requests.

3. Buyer issues several purchase orders, which are then authorised and served by the supplier; however, supplier fails to deliver a few of them, thus, causing the buyer fall into an indefinite postponement; orders for which the product is not to be delivered are picked at random.

4-7. In the following four scenarios, we have modified the behaviour of the supplier so that it does not behave as expected, simulating that the system is under the control of an attacker. The attacker fakes messages of the expected kind, but with different information, or prevents actual messages from being sent (NB, alternatively, changes in the behaviour of the supplier could be taken to be simply software or hardware flaws.) The first scenario comprises an impersonation attack, where the intruder sends extra delivery orders to the buyer, guessing both the consecutive number of purchase orders, and the number of each individual packet. In the second scenario, the attacker simply stops a few selected messages from being delivered to the supplier. In particular, the attacker aims at intercepting messages from the buyer, claiming that a confirmed purchase order has not been complete or delivered. In the third scenario, the attacker impersonates the supplier, faking a message whereby a purchase order is rejected; i.e., cancelled, for whatever reason. Finally, in the fourth scenario, the attacker now impersonates the buyer so as to issue extra purchase orders, which are then served by the supplier, and eventually attempted at being delivered at the buyer side.

8-11. Four further attacks can be elaborated, similar to the above ones, this time, though, attempting to fool the buyer. So, the first scenario comprises an impersonation attack, where the intruder sends confirmation messages to the buyer for purchase orders that have been cancelled or rejected by the supplier. In the second scenario, the intruder prevents purchase confirmation messages from the supplier from being delivered at the buyer. In the third scenario, the intruder again impersonates the supplier, randomly rejecting some purchase orders, which have been actually accepted by the supplier. Finally, in the fourth scenario, the attacker now impersonates the supplier so as to issue extra confirmation messages to purchase orders that were never set by the buyer.

6. Conclusions and Further Work

The verification technique for integrity policies developed in this work allows processes to be validated from an independent party, which was not involved during the development cycle of the system. Our technique does not require to modify or to have access to the source code. It can be applied during system testing, but more importantly during production. The solution allows monitoring the correct functioning of the processes according to a set of defined security integrity policies. Further, our technique is fully automated, though it works only for systems that have been developed under .NET.

Our technique comes with an integrity policy language, with which we can succinctly capture integrity policies, in terms of the system expected behaviour. In a way, they express the ideal system behaviour upon the absence of any intruder. Policies are expressed in terms of method's parameters or return variables, and may involve as well temporal conditions to capture time limits as to how one should wait for an event to occur. Our technique then reads the set of security integrity policies and then verifies them, using observation points, each of which is looking into input/output method parameters referred to by the policies.

We successfully tested our verification technique using a case study involving a critical process for purchase order management. For this case study, we developed separate processes exhibiting the expected functional behaviour of each party. Then, we developed a set of security integrity policies, which attempted to capture the expected behaviour of the entire system, upon correct conditions. Then, we simulated the execution of the system, considering several rates, including purchase order arrival, average service time for an order on each party, etc. Finally, we considered several scenarios, where an intruder attempts to get the system confused or disrupted. These kinds of attacks are very common, and usually are the intruder first action, while it is looking for diamonds in the dessert. We emphasise that these attacks were all timely detected.

6.1 Directions for Further Work

Further work includes extending the injection process in order to support .NET signed components (Signing .NET components is part of a Microsoft technique for avoiding unauthorized modifications or tampering) [11]; this may involve designing an on-top layer to provide critical security properties as in [12]. While here we have worked under the assumption that system communication is secure, and fault-tolerant, further work involves specifying rules and network architectures under which support our assumptions, and that are beyond the scope of this paper. Further work also includes translating a set of integrity security policies into a mechanism other than a SQL server, as this approach may not be able to scale up for larger, and more complex systems to be monitored. Likewise, further work involves extending the mechanism so as to make it able to deal with a larger number of events, coming from a number of system components.

References

[1] SearchSecurity, Security, Audit, Compliance and Standards, TechTarget. Available from http://searchsecurity.techtarget.com/resources/Security-Audit-Compliance-and-Standards

[2] F. B. Schneider, "Enforceable Security Policies", ACM Transactions on Information and System Security vol. 3, no. 1, pp. 30–50, 2000.

[3] C. P. Pfleeger and S. Lawrence, "Security in Computing", Boston, MA: 4th Ed., Prentice-Hall, 2006, pp. 1-771.

[4] P. Li et al., “Information Integrity Policies”, in Workshop on Formal Aspects of Security & Trust (FAST), Philadelphia, PA: University of Pennsylvania, 2003, pp. 53–70.

[5] D. D. Clark and D. R. Wilson, “A Comparison of Commercial and Military Computer Security Policies”, in IEEE Symposium on Research in Security and Privacy (S&P'87), Oakland, CA: IEEE Computer Society Press, 1987, pp. 184–193.

[6] CSO, Security and Risk, Data Protection, Separation of Duties and IT Security. Available from http://www.csoonline.com/article/446017/separation-of-duties-and-it-security

[7] Microsoft, SCOM: System Center Operations Manager, Available from http://technet.microsoft.com/en-us/library/hh205987.aspx

[8] X. Liu et al, “D3S: Debugging Deployed Distributed Systems”, in Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI’2008), San Francisco, CA: 2008, pp. 423-437.

[9] G. R. Andrews, “Foundations of Multithreaded, Parallel, and Distributed Programming”, Addison–Wesley, 2000, pp. 1-664.

[10] G. W. Treese and L. C. Stewart, “Designing Systems for Internet Commerce”, 2nd Ed., Addison-Wesley Professional, 2002, pp. 1-496.

[11] M. Downen, “Using Strong Names Signatures”, MSDN Magazine, July, 2006, Available from http://msdn.microsoft.com/en-us/magazine/cc163583.aspx

[12] V. Alarcón-Aquino et al., “Design and Implementation of a Security Layer for RFID Systems”, Journal of Applied Research and Technology vol. 6, no. 1, pp. 69-83, 2008.