A Non-Functional Approach to System Integrity

Simon N. Foley, Member, IEEE

Abstract—**Systems provide integrity protection by ensuring that there is no unauthorized modification of information. Traditional models of protection tend to define integrity in terms of ad-hoc authorization techniques whose effectiveness is justified more on the basis of experience and "best practice", rather than on any common theoretical foundation. A formal definition of integrity is proposed that is independent of any particular implementation mechanism. A series of simple examples are used to demonstrate that existing integrity mechanisms such as separation of duties, well-formed transactions, and so forth, can be regarded as implementation techniques for achieving integrity. The proposed characterization of integrity is non-functional, that is, it falls into the same category of properties as non-interference and its relatives. As a consequence, validating that a system has integrity can be expected to be as challenging as validating that a system upholds non-interference.**

Index Terms— **Computer security, Cryptography, Data Security, Fault tolerance, Modeling, Protection, Protocols, Reliability, Software verification and validation, System analysis and design.**

I. Introduction

Early security research [2] characterized integrity in terms of read-write access controls between subjects and objects. This provides for a very coarse interpretation of integrity [25]; for example, once granted access to an account database, a bank clerk can make any change to the customer's account details. Access triples, well-formed transactions, and the principles of encapsulation [4], [18], provide finer grained control by constraining the operations that a subject may carry out on an object: the bank clerk may execute only deposit or withdraw operations to access an account database.

Many integrity compromises are a result of 'insiders' executing authorized operations that are fraudulent. For example, the bank clerk executes an account deposit without lodging actual funds. Separation of duty [4], [8], [29], [32] controls decrease the potential for fraud by involving at least two individuals at different points in a transaction: for example, by reconciling bank accounts and funds received each day, a supervisor detects and corrects the fraudulent deposit by the clerk. Role Based Access Control models [26] and authorization models [1], [15] provide integrity controls based on structures that organize related operations into roles and constrain the way in which roles may be assigned and/or inherited by users; separation of duty is expressed within these models using role constraints.

These conventional security models describe *controls* for achieving integrity; they take an operational and/or implementation oriented approach by defining *how* to achieve integrity. No attempt is made to formalize a *property* that defines *what* is meant by integrity. For example, [4] recommends a combination of separation of duties, access-triples and auditing as a strategy for achieving integrity: it does not attempt to address what is meant by integrity. Confidence is achieved to the extent that good design principles have been applied; there is no assurance that an integrity property is upheld. Thus, when we define a complex separation of duty policy we do not know, for certain, whether a dishonest user can bypass the intent of the separation via some unexpected circuitous, but authorized, route.

Jacob [12] formalizes integrity as a functional property. This means that an integrity mechanism determines whether the current request for an operation is authorized, based on the history of past authorization requests leading to the current state. In [9] and in this paper, we illustrate that integrity is non-functional. Non-functionality means that in order to determine authorization, it is necessary to examine *every possible* history of requests that could have reached the current state. Non-interference is an example of a non-functional property, and while non-interference and its derivatives have been extensively studied [6], [10], [22], [24], designing and verifying security mechanisms that uphold non-functional properties is difficult [17], [28]. Thus, we conjecture that building mechanisms to uphold integrity can be expected to be as difficult as building mechanisms to uphold non-interference.

The paper is organized as follows. Section II provides two formal characterizations of integrity. The first characterization defines integrity as an attribute of dependability based on the interpretation in [16]. The second considers integrity as external consistency as described in [4]. A series of simple examples in Section III illustrate how various implementation mechanisms achieve integrity. Section IV describes why integrity should be regarded as a non-functional property and investigates some of its properties. Section V considers how integrity might be cast as a liveness-style property.

The Communicating Sequential Processes (CSP) process calculus [21] is used to provide a consistent framework for presenting the results in this paper. The Appendix provides a short overview of CSP and the notation that is used in this paper.

II. Characterizing Integrity

Traditional requirements analysis identifies the *essential* functional requirements that define *what* a system must do. An implementation defines *how* the system operates and must take into consideration the fact that the *infrastructure* put in place to support the requirements may be unreliable. For example, experience tells us that a system's infrastructure should include a suitable backup and restore subsystem. While not part of the essential requirements, it is a necessary part of the implementation, since the infrastructure can corrupt data. Infrastructure is everything that serves the requirements: software, hardware,

Manuscript received April 2002; Revised August 2002.

Author's address: Department of Computer Science, University College Cork, Ireland (email: s.foley@cs.ucc.ie).

users, and so forth. Even where a system is functionally correct, the infrastructure is likely to fail: software fails, users are dishonest, do not follow procedures, and so forth. The system must be designed to be resilient to these infrastructure failures. To properly characterize integrity we must consider, therefore, how the infrastructure can affect the behavior of the system.

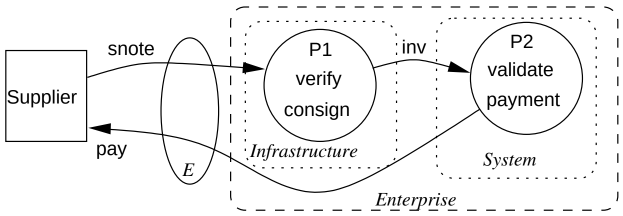

Fig. 1. A simple payment enterprise

Example 1: A simple enterprise receives shipments, and generates associated payments for a supplier. For simplicity, the value of a shipment is abstracted out of the description by assuming that shipments have equal value. Requirements Analysis identifies the events snote and pay corresponding to the arrival of a shipment (note) and its associated payment, respectively. The *enterprise* must ensure that the number of payments issued do not exceed the number of shipments received. This is specified by the CSP process $ConsReq_0$.

$$\begin{aligned} ConsReq_0 &= (\mathsf{snote} \rightarrow ConsReq_1) \\ ConsReq_i &= (\mathsf{pay} \rightarrow ConsReq_{i-1}) \\ &\quad \Box\ (\mathsf{snote} \rightarrow ConsReq_{i+1}) \qquad (i > 0) \end{aligned}$$

Figure 1 outlines a possible implementation of this requirement. A clerk verifies shipment notes and enters invoice details (event inv) to a computer system which, in turn, generates payment (pay) to the supplier. This is specified as

$$\begin{aligned} Clerk &= (\mathsf{snote} \rightarrow \mathsf{inv} \rightarrow Clerk) \\ System &= (\mathsf{inv} \rightarrow \mathsf{pay} \rightarrow System) \end{aligned}$$

and the enterprise design is specified as $ConsImp = System \| Clerk$. Intuitively, integrity is maintained since $ConsImp$ supports requirement $ConsReq_0$ at its external interface E with the supplier. △

The example above illustrates integrity characterized as a form of refinement—$ConsImp$ refines $ConsReq_0$. In the traces model of CSP, process S is a (safety) refinement of process R if $\alpha R = \alpha S$ and $traces(S) \subseteq traces(R)$, that is, every possible trace of S is permitted by R [21]. For example, the process $P = (\mathsf{snote} \rightarrow \mathsf{pay} \rightarrow P)$ which alternates between snote and pay, is a refinement of process $ConsReq_0$.

The Supplier (Example 1) is oblivious to the 'internal' event (inv) and interacts with $ConsImp$ abstracted through interface $\{\mathsf{snote}, \mathsf{pay}\}$, that is, $ConsImp@\{\mathsf{snote}, \mathsf{pay}\}$, where for process S and set of events E,

$$S@E = \{\, t : traces(S) \bullet t \upharpoonright E \,\}$$

and $t \upharpoonright E$ is the trace t with events absent from E removed. Every trace the supplier can observe from $ConsImp@\{\mathsf{snote}, \mathsf{pay}\}$ is permitted by $ConsReq_0$ and we say that *ConsImp locally refines* $ConsReq_0$ at that interface, that is, $ConsReq_0 \sqsubseteq^{\{\mathsf{snote},\mathsf{pay}\}} ConsImp$.

Definition 1: (Local Refinement) R is locally refined by S at event interface E iff $R \sqsubseteq^E S$, where, $R \sqsubseteq^E S \Leftrightarrow E \subseteq \alpha R \subseteq \alpha S \wedge S@E \subseteq R@E$. ◇

Example 2: Continuing Example 1, assume that the computer system will behave reliably as $System$. However, it is unreasonable to assume that the clerk will always act reliably as $Clerk$. In practice, an unreliable clerk ($\overline{Clerk}$) may take on any behavior involving events snote and inv.

$$\begin{aligned} \overline{Clerk} &= Run_{\{\mathsf{snote},\mathsf{inv}\}} \\ ConsImp2 &= System \| \overline{Clerk} \end{aligned}$$

$ConsImp2$ is a more realistic representation of the actual enterprise. It more accurately reflects the reliability of its infrastructure than the previous design $ConsImp$. However, for external interface $E = \{\mathsf{snote}, \mathsf{pay}\}$, since $t = \langle \mathsf{inv}, \mathsf{pay} \rangle \in traces(ConsImp2)$, and $t \upharpoonright E = \langle \mathsf{pay} \rangle \notin traces(ConsReq)$ then $ConsReq \not\sqsubseteq^E ConsImp2$, that is, when operating within its expected infrastructure, the design of $System$ is not sufficiently robust to be able to safely support the original requirements of $ConsReq$. △

In [16] integrity is given as one attribute of *dependability*, that is, integrity is defined as *"dependability with respect to absence of improper alterations"*. Dependability is characterized as a *"property of a computer system such that reliance can be justifiably placed on the service it delivers"* [16]. Dependability is not an absolute property, in the sense that its justification will always be conditional on the proper behavior of some components. Therefore, we choose to define dependability as a relative property that can be used to determine whether one system is at least as dependable as another system in the services it provides.

Definition 2: (Dependability) If R gives the behavioral requirements for an enterprise and S is its proposed implementation, including details regarding the nature of the reliability of its infrastructure, then S is at least as *dependably safe* as R at interface E if and only if $R \sqsubseteq^E S$. ◇

In comparing the dependability of two systems, we assume that their specification can include details of potential improper behavior within their infrastructure. This may include details of improper alteration and therefore, based on the interpretation of integrity from [16], local refinement may be used to compare the integrity of two systems. Given suitable specifications R and S, then $R \sqsubseteq^E S$ means that for the service provided via interface E then S is at least as dependable, with respect to improper alterations, as R. $R \sqsubseteq^E S$ can be interpreted to mean that S provides at least as much integrity protection for its interface E as does R.

Example 3: $ConsReq$ and $ConsImp2$ are functional specifications (Example 2). $ConsReq$ assumes the correct behavior of its components, while $ConsImp2$ includes an unreliable clerk who can make improper alterations to invoices. We have $ConsReq \not\sqsubseteq^E ConsImp2$, which means therefore, that relative to $ConsReq$, $ConsImp2$ does not achieve integrity for services provided via the interface $E = \{\mathsf{snote}, \mathsf{pay}\}$. △

Another interpretation of integrity is that the system concerned provides *external consistency*, that is, the *"correct correspondence between data objects and the real world"* [4]. An alternative viewpoint is that an external entity can achieve consistent interactions with the enterprise, even in the presence of failures within the infrastructure of the enterprise. We characterize this notion of external consistency in terms of dependability.

Definition 3: (External Consistency) Let $S||I$ and $S||\overline{I}$ describe the behavior of system S operating within reliable, and unreliable, infrastructures I and $\overline{I}$, respectively. We say that S is *externally consistent* at interface E if $S||\overline{I}$ is as dependably safe as $S||I$, that is, $S||I \sqsubseteq^E S||\overline{I}$. ◇

Example 4: Given the nature of an unreliable clerk (Example 2), *System* is not externally consistent at E, that is, $System||Clerk \not\sqsubseteq^E (System||\overline{Clerk})$. △

III. Integrity Mechanisms

A. Separation of Duties

Separation of duties is a common implementation technique used to achieve integrity. While fault-tolerant techniques replicate an operation, separation of duties can be thought of as a partitioning of the operation.

Example 5: When a shipment arrives, a clerk verifies the consignment at goods-inwards (entering details cons into the system). When an invoice arrives, a different clerk enters details into the system, and if the invoice matches a consignment, a payment is generated. While as the operations are separate then a single clerk entering a bogus consignment cons or invoice inv can be detected by the system. For simplicity, we assume that both consignment and invoice details are contained within the shipment note snote; this is depicted in Figure 2.

To distinguish shipments, events are prefixed with identifiers drawn from $\mathcal{N}$, the set of shipment-identifiers. For example, $n.\mathsf{pay}$ corresponds to the payment resulting from shipment-note $n.\mathsf{snote}$. While shipment-identifiers are intended to be unique, it is possible that a supplier may re-use identifiers. Thus, $n{:}ConsReq_0$ (process $ConsReq_0$ with events prefixed by n) describes the behavior required when processing shipments identified by $n \in \mathcal{N}$. The top-level requirement is

$$ConsReq = ||_{n:\mathcal{N}}(n{:}ConsReq_0)$$

The system allows arbitrary clerks u and v to verify consignment ($n.\mathsf{cons}.u$) and invoice ($n.\mathsf{inv}.v$) for consignment n, after which payment is generated.

$$\begin{aligned} AppSys = &\Box_{\substack{n:\mathcal{N}\\u:U}} (n.\mathsf{cons}.u \\ &\qquad\to \Box_{v:U}\ (n.\mathsf{inv}.v \to n.\mathsf{pay} \to AppSys))\end{aligned}$$

This system allows the same clerk to perform both operations, and a separation of duty mechanism is required to limit certain behaviors. Specification

$$Sep_u^v = Stop_{\{\mathsf{cons}.u,\mathsf{inv}.v\}}||Run_{\{\mathsf{cons}.v,\mathsf{inv}.u\}}$$

separates clerks u and v who may process invoices and consignments, respectively, but not vice-versa. If we assume that the infrastructure has only two clerks $U = \{x, y\}$ then a dynamic separation of duty mechanism, allowing a clerk to vary operations between shipments is specified as *DynaSep*. Separation of duty is succinctly expressed using the the CSP external choice operator "□" [9], [23].

$$\begin{aligned} DynaSep &= ||_{n:\mathcal{N}}(n{:}Sep_x^y \Box n{:}Sep_y^x) \\ StatSep &= (||_{n:\mathcal{N}}n{:}Sep_x^y) \Box (||_{n:\mathcal{N}}n{:}Sep_y^x)\end{aligned}$$

StatSep describes a static separation of duty mechanism requiring a clerk to perform the same operation for all shipments. The overall (reliable) system is described as

$$SepSys = AppSys||DynaSep$$

A reliable clerk u processing shipment n behaves as process $n{:}Clerk^u$, where

$$\begin{aligned} Clerk^u = &(\mathsf{snote} \\ &\quad\to ((\mathsf{cons}.u \to Clerk^u) \Box (\mathsf{inv}.u \to Clerk^u)))\end{aligned}$$

However, we make the assumption that, of our two clerks x and y, then one may take on an unreliable or arbitrary behavior. Thus, the unreliable infrastructure behavior is $\overline{Clerks}$, where

$$\begin{aligned} \overline{Clerks} = ||_{n:\mathcal{N}}\ n{:}((&Clerk^x||Run_{\alpha Clerk^y}) \\ &\Box\ (Clerk^y||Run_{\alpha Clerk^x}))\end{aligned}$$

Since the system and separation mechanism ensure that one failing clerk cannot influence the generation of a payment without the assistance of the other clerk, then, we can prove that for any $n : \mathcal{N}$ and $n{:}E = \{n.\mathsf{snote}, n.\mathsf{pay}\}$,

$$ConsReq \sqsubseteq^{n:E} (SepSys||\overline{Clerks})$$

As currently defined, our specification favors the payment-enterprise, not the supplier: payments may be very late, or not made at all, but are never bogus. If a clerk fails then payment is not made. In reality, the infrastructure contains many additional components; audit logs to record failures and supervisors, who make judgments and rectify these inconsistencies. △

Example 6: Example 5 illustrates how separation of duties may be regarded as an implementation technique for achieving dependability. The implementation also maintains external consistency on shipments, since,

$$SepSys||Clerks \sqsubseteq^{n:E} SepSys||\overline{Clerks}$$

where $Clerks = ||_{n:\mathcal{N}}n{:}(Clerk_i^x||Clerk_i^y)$ characterizes a reliable infrastructure comprised of honest clerks. △

B. Cryptographic Checksums

Example 7: Suppose that the application system and the supplier (Example 1) share a secret cryptographic key that is unknown to the clerk. The supplier provides a cryptographic checksum/Message Authentication Code (MAC) with each shipment note. This checksum (entered by the clerk with the shipment details) is used by the application system to ensure that the shipment details are authentic and have not been tampered with by the clerk.

Let $\mathcal{M}$ be a data-type representing shipment-identifiers plus associated MAC fields. Let $\mathcal{N}$ be the set of all values from $\mathcal{M}$

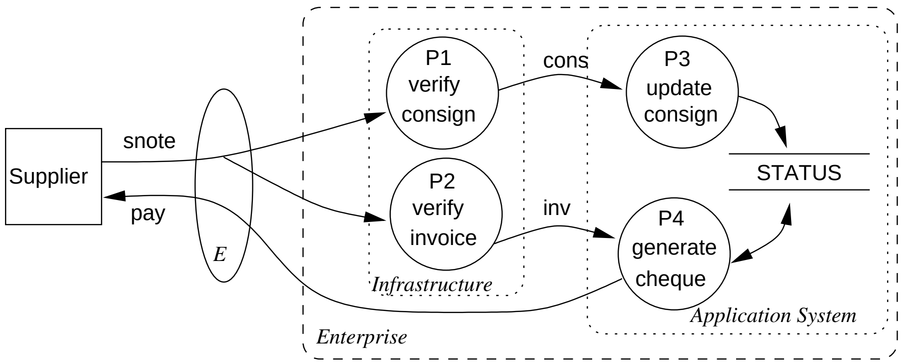

Fig. 2. Supporting separation of duties

that represent cryptographically secured shipment-identifiers, that is, the MAC component corresponds correctly to the identifier. Values in $\mathcal{N}$ cannot be generated/forged by an entity that does not know the secret MAC key. Let $\overline{\mathcal{N}}$ represent all other values in $\mathcal{M} \setminus \mathcal{N}$. The top-level requirement is as before, except that we expect the supplier to use only cryptographically secured shipment-identifiers.

$$ConsReq = \|_{n:\mathcal{N}}(n{:}ConsReq_0)$$

The system is expected to generate payments for valid invoices that it has not seen before. A system that has processed $P \subseteq \mathcal{N}$ shipment-identifiers to date has behavior

$$\begin{aligned} MacSys_P = \; & (\Box_{n:\mathcal{N}\setminus P}\ (n.\mathsf{inv} \to n.\mathsf{pay} \to MacSys_{P\cup\{n\}})) \\ & \Box \\ & (\Box_{n:\overline{\mathcal{N}}\cup P}\ (n.\mathsf{inv} \to MacSys_P)) \end{aligned}$$

Invoices processed in the past (shipment identifiers from P), or with invalid identifiers (from $\overline{\mathcal{N}}$) are processed, but payment is not generated. Since the system knows the secret MAC key it then has the ability to test the validity of a shipment-identifier.

A reliable clerk simply takes shipment details (including the checksum) and enters the details as invoices to the system. This behavior is described as $MClerk = \|_{n:\mathcal{N}}(n{:}Clerk)$, where *Clerk* is defined in Example 1. If a cryptographically strong checksum scheme is used and since the clerk does not know the secret key, then it is reasonable, to assume, therefore, that it is not feasible for the clerk to forge messages in $\mathcal{N}$. Thus, an unreliable clerk engaging in arbitrary events, can feasibly generate messages only from $\overline{\mathcal{N}}$, or messages from $\mathcal{N}$ that it has processed in the past. Thus, an unreliable clerk that has processed $P \subseteq \mathcal{N}$ messages in the past has the following behavior.

$$\begin{aligned} \overline{MClerk}_P = \; & (\Box_{n:\mathcal{N}}\ (n.\mathsf{snote} \to Clerk_{P\cup\{n\}})) \\ & \Box \\ & (\Box_{n:\overline{\mathcal{N}}\cup P}\ (n.\mathsf{inv} \to Clerk_P)) \end{aligned}$$

Given this defintion, we can prove that the resulting enterprise is as dependably safe as the original requirement:

$$ConsReq \sqsubseteq^{n:E} MacSys_{\{\}} \| \overline{MClerk}_{\{\}}$$

△

Studying the effects of normal versus abnormal infrastructure behavior has been successfully applied to cryptographic security protocols, for example [5], [19], [20]. Analysis is based on a generic behavior (called 'Spy' [20] or adversary [5]) characterizing the untrusted network (abnormal infrastructure) over which a security protocol operates. Our integrity analysis generalizes this by considering many 'spys', each one characterizing the infrastructure threats that a protection mechanism must withstand.

C. Confidentiality

Confidentiality is a further attribute of dependability [16] and, for the sake of completeness, this section illustrates how multilevel security might be formally characterized in terms of refinement.

Example 8: Using our fault model, the reliable part of a multilevel secure system is the Trusted Computing Base (TCB), while the remaining operating system and applications make up the unreliable infrastructure. Consider a TCB providing interfaces to low and high users. The TCB has to be sufficiently robust to be able to provide an externally consistent interface to a low user regardless of the behavior of a high application, that is, the *TCB* running a high application A_h is as dependably safe as *TCB* running any other high application A'_h. Or, in other words, that *TCB* is externally consistent at the low interface.

$$\begin{aligned} & \forall A_l, A_h, A'_h \mid \alpha A_l = Lo \wedge \alpha A_h = \alpha A'_h = Hi \\ & \quad \bullet\ (A'_h \| TCB \| A_l) \sqsubseteq^{Lo} (A_h \| TCB \| A_l) \end{aligned}$$

This can be shown to simplify to $(TCB \| Stop_{Hi}) \sqsubseteq^{Lo} TCB$, and simplifies further to $(TCB \| Stop_{Hi})@Lo = TCB@Lo$. This corresponds to non-information flow [7], [14] and is not unlike non-deducibility [31]. If *Lo* and *Hi* partition the entire alphabet of *TCB*, then it simplifies further to non-inference [14]: $TCB@Lo \subseteq TCB$. △

D. Fault-Tolerance

Another approach to dealing with unreliable systems (infrastructure) is to replicate the faulty components to make the system fault tolerant. We can make the payment enterprise fault tolerant if we replicate the clerk. We assume that every shipment is processed by $2k+1$ replicated clerks. The system votes (on the $2k+1$ invoices) to decide whether or not a consignment is valid. In this case, the abnormal behavior of the infrastructure is represented by at least $k+1$ clerks having normal behavior, and we argue that the resulting enterprise is as dependably safe as *ConsReq* at interface $n{:}E$.

Non-interference techniques have been used previously to verify fault-tolerance [21], [30], [33]. Faulty behavior is modeled using special *fault* events and the system is fault-tolerant if the fault events are non-interfering with the critical events of the system. In essence, engaging a *fault* event changes the system from normal to abnormal behavior, and what may be thought of as external consistency must be preserved on the critical events that make up the external interface.

E. Security Kernels

In Example 5 we considered the integrity of the enterprise with regard to the external supplier and assumed that *SepSys* was reliable, that is, secure. A secure application system may be built in terms of untrusted (unreliable) applications running on an underlying Trusted Computing Base.

Example 9: Consider the application system used by the payment enterprise (Example 5). Figure 3 depicts a design of this system based on a simplistic model of an assured pipeline [3]. The applications form an infrastructure composed of programs $\mathsf{P1}$, $\mathsf{P2}$ and $\mathsf{P3}$ that run in respective domains $\mathsf{D1}$, $\mathsf{D2}$ and $\mathsf{D3}$ provided by the pipeline. In this example we do not attempt to model the usual confinement properties associated with the domains. The integrity of an assured pipeline based application relies on the separation enforced between domains, and the 'correctness' of the applications along the pipeline.

We specify a model of the assured pipeline. It is highly simplistic, but serves as an illustration. It does not consider the usual confinement properties associated with the domains. Event $n.\mathsf{d1}$ represents entry into domain $\mathsf{D1}$ by program $\mathsf{P1}$ (processing shipment n). Events $n.\mathsf{d2}$ and $n.\mathsf{d3}$ have similar interpretations. The pipeline enforces a strict ordering on domain entry.

$$Pipeline = \Box_{n:\mathcal{N}}\ (n.\mathsf{d1} \rightarrow n.\mathsf{d2} \rightarrow n.\mathsf{d3} \rightarrow Pipeline)$$

When a cons event is engaged the program enters domain $\mathsf{D1}$, and similarly for inv (these events will eventually be prefixed by shipment identifier).

$$\mathsf{P1} = \Box_{u:U}\ (\mathsf{cons}.u \rightarrow \mathsf{d1} \rightarrow P1)$$
$$\mathsf{P2} = \Box_{u:U}\ (\mathsf{inv}.u \rightarrow \mathsf{d2} \rightarrow P2)$$

The payment program $\mathsf{P3}$ behaves slightly differently. Once the pipeline enters domain $\mathsf{d3}$ a payment may be generated.

$$\mathsf{P3} = \mathsf{d3} \rightarrow \mathsf{pay} \rightarrow \mathsf{P3}$$

Our failure model assumes that one of the programs $\mathsf{P1}$ or $\mathsf{P2}$ may fail and engage in events arbitrarily. Failure of program $\mathsf{P3}$ can result in multiple payments and therefore it is necessary to treat the payment program $\mathsf{P3}$ as a reliable component. This is not an unreasonable assumption: for example, a typical guard pipeline regards that part which generates the output as trusted [11]. Thus, the infrastructure is modeled as $\overline{Apps} = ||_{n:\mathcal{N}}(n{:}\overline{Trans})$, where $\overline{Trans}$ specifies the unreliable processing of a single shipment.

$$\overline{Trans} = ((\mathsf{P1}||Run_{\alpha\mathsf{P2}}) \Box (\mathsf{P2}||Run_{\alpha\mathsf{P1}}))||\mathsf{P3}$$

And we can prove that $AppSys \sqsubseteq^{\alpha AppSys} Pipeline||\overline{Apps}$. △

IV. Evaluating Dependability

A. Non-Functionality

If we take the view that refinement is used to characterize the property of interest [13], then trace refinement and local refinement may be regarded as functional and non-functional properties, respectively. This observation is explained as follows.

Trace refinement is predicate on traces and is, therefore, considered a functional property in the sense of [17], [28]: the refinement

$$R \sqsubseteq S \Leftrightarrow \forall\, t \in traces(S) \bullet t \in traces(R)$$

means that the validity of t (as a trace of S) can be determined by testing the validity of t (as a trace of R) independent of the other traces. Therefore, building an implementation mechanism S that upholds the requirements of R is straightforward in the sense that, in any state of S then determining the validity of an operation request is based on the past operations (trace) that led to this state.

Local refinement, on the other hand, is expressed as a predicate on sets of traces and is, therefore, considered a non-functional property in the sense of [17], [28]: the refinement

$$R \sqsubseteq^{E} S \Leftrightarrow \forall\, t \in traces(S) \bullet \\ (\exists\, t' : traces(R) \bullet t' \upharpoonright E = t \upharpoonright E)$$

means that it is necessary to consider the set of traces of R to determine the validity of t (as a trace of S). Building mechanisms which uphold non-functional properties is difficult, since determining the validity of an operation request (in some state of S) is based on testing all possible similar traces that could have led to this state.

Since local refinement is a non-functional property, then it falls into the category of security properties such as non-interference [10], deducibility [31], and so forth. This is not surprising given that Example 8 cast (non) information flow in terms of local refinement. The reader is referred to [17] and [28] for consideration of the difficulties associated with building general mechanisms that uphold such properties.

Trace refinement preserves local refinement:

$$\frac{R \sqsubseteq S}{R \sqsubseteq^{E} S}\quad [\, E \subseteq \alpha R\,]$$

However, the converse does not necessarily hold, as illustrated by the following example.

Example 10: Consider processes P and Q, where

$$P = (a \rightarrow b \rightarrow P) \Box (a \rightarrow P)$$
$$Q = (a \rightarrow b \rightarrow P)$$

We have $Q \sqsubseteq^{\{a\}} P$ since both processes may be willing to engage in any number of a's through this interface. However, $Q \sqsubseteq P$ does not hold since $traces(P) \not\subseteq traces(Q)$. △

Trace refinement can be used as an approximation of local refinement. For Example 8, suppose that TCB was designed such that $TCB||Stop_{Hi} \sqsubseteq TCB$ holds. By the law above, this implies that $TCB||Stop_{Hi} \sqsubseteq^{Lo} TCB$ holds. However such a TCB

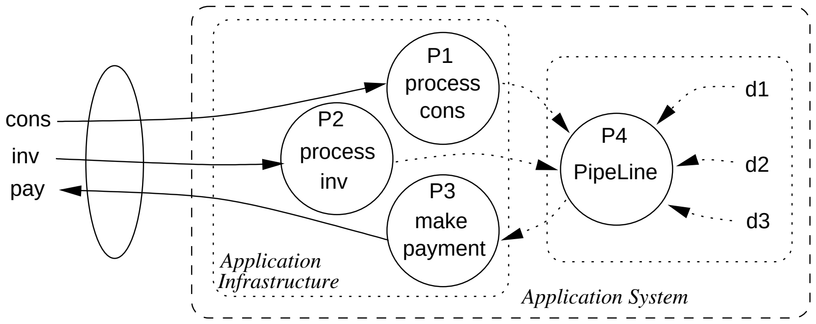

Fig. 3. Application running on a TCB

is not of much use—for every trace t of *TCB* then $t \upharpoonright Hi = \langle\rangle$—it is not willing to engage in *any Hi* event.

While an implementation system may trace refine a requirement, it is not necessarily the case that when the infrastructures are considered then the overall enterprise (system plus infrastructure) dependably refines its requirement. This is illustrated in the following example, which shows that trace refinement does not necessarily preserve the external consistency of a system.

Example 11: Consider system requirement P and implementation Q from Example 10. Let I and $\overline{I}$ define the normal and abnormal infrastructures, respectively.

$$I = (a \rightarrow a \rightarrow b \rightarrow I)$$
$$\overline{I} = Run_{\{a,b\}}$$

Under the trace model of a process we have $P||I = P||\overline{I} = P$ and thus P is externally consistent at interface $\{a\}$. Specification Q is also a trace refinement of P ($P \sqsubseteq Q$). However, we have $Q||I = (a \rightarrow Stop)$ and $Q||\overline{I} = Q$ and thus $Q||I \not\sqsubseteq^{\{a\}} Q||\overline{I}$, that is, Q is not externally consistent at this interface. △

B. Incremental Evaluation

We interpret $R \sqsubseteq^{\alpha R} S||I_S$, to mean that the system S is sufficiently resilient to the faults in I_S to be able to (safely) support the requirements R. This dependable component may then be used in place of R which, in turn, may be used in place of some other more abstract requirement. In general,

$$\frac{R \sqsubseteq^{E} S||I_S \ , \ S \sqsubseteq^{\alpha S} P||I_P}{R \sqsubseteq^{E} (P||I_P||I_S)}$$

Example 12: We have from Examples 5 and 9 for, $n \in \mathcal{N}$ and $E = \{\mathsf{snote}, \mathsf{pay}\}$,

$$ConsReq \sqsubseteq^{n:E} AppSys||DynaSep||\overline{Clerks}$$
$$AppSys \sqsubseteq^{\alpha AppSys} PipeLine||\overline{Apps}$$

and it follows that

$$ConsReq \sqsubseteq^{n:E} PipeLine||\overline{Apps}||DynaSep||\overline{Clerks}$$

Thus, a trusted computing base composed of the pipeline and dynamic separation of duty mechanism is sufficiently resilient to infrastructure failures (clerks and programs) and supports the original requirement *ConsReq*. △

C. Composition

Under certain circumstances, if systems S and S' are dependable (according to requirements R and R') then so is their composition.

$$\frac{R \sqsubseteq^{E} S \ , \ R' \sqsubseteq^{E} S'}{R||R' \sqsubseteq^{E} S||S'} \quad [\ \alpha R \cap \alpha R' \subseteq E\ ]$$

We note, however, that if the side-condition $\alpha R \cap \alpha R' \subseteq E$ does not hold, then $R||R' \sqsubseteq^{E} S||S'$ does not necessarily hold, since synchronization on events in $(\alpha R \cap \alpha R') \setminus E$ may result in behavior restrictions in $R||R'$ that are not restricted in $S||S'$.

V. Discussion: Safety and Liveness

Based on the CSP traces model, the proposed definition of integrity is a *safety* style property in the sense that fraudulent (or abnormal) activity can be *detected* by the system, and the integrity violation *prevented*. However, no guarantees can be given regarding the subsequent *liveness* of the system. A trace based analysis cannot verify whether the detected fraud will be *corrected* and that the system will continue to provide service. Considering Example 1, an application system $Stop_{\{\mathsf{inv},\mathsf{pay}\}}$ that provides no service cannot be defrauded by a dishonest clerk and, thus, relative to *ConsReq*, it has integrity.

The failures divergences [21] model of CSP may be used to encode a 'liveness' style definition of integrity. The *lazy abstraction* [21], [22] of process P via interface E is defined as the interleaving of P with events not in E, that is, $P\ |||\ Run_{\alpha P \setminus E}$. This provides abstraction by effectively camouflaging events that are not in E. Lazy abstraction is applicable within the failures divergences model of CSP and local refinement is redefined as:

$$R \sqsubseteq^{E}_{FD} S \Leftrightarrow (R\ |||\ Run_{\alpha R \setminus E} \sqsubseteq_{FD} S\ |||\ Run_{\alpha S \setminus E})$$

where $\sqsubseteq_{FD}$ is failures divergences refinement. This characterization means that guarantees can be given regarding the liveness of the system. That is, fraudulent (or abnormal) activity *will be detected* and *will be corrected*.

VI. Conclusion

By considering the nature of the entire enterprise we provide a meaningful and implementation-independent definition for integrity. This systems view has not been taken by conventional integrity models which limit themselves to the boundary of the computer system and tend to define integrity in an operational/implementation-oriented sense.

We expect that the proposed interpretation of integrity may be used in a number of ways. It may be used to verify the effectiveness of existing and new mechanisms. For example, validate that the design of a mechanism that uses a combination of separation of duty and protection domains upholds integrity for certain classes of abnormal infrastructure. Alternatively, given a system built using verified integrity mechanisms, one could verify that integrity is upheld given some proposed infrastructure configuration. For example, given the intended authorizations of clerks and supervisors, will the security configuration of the system ensure external consistency of bank balances? Such analysis goes beyond existing analysis techniques such as [27] which searches for possible conflicts between separation of duty, user role assignment and role inheritance rules. Since our approach provides a semantics for integrity, it may be used to determine whether the separation of duty controls actually achieve external consistency and whether the application uses integrity mechanisms correctly. Using the proposed framework to provide a formal basis for understanding the scope of ad-hoc techniques such as [27] is a topic for future research.

This paper has sought to facilitate a better understanding of integrity; further research is necessary to develop practical approaches to verifying the integrity of large complex systems. For the sake of ease of exposition, the simple traces model of CSP was used to define integrity in this paper, The result is a non-functional interpretation of integrity that is closely related to existing security properties such as non-interference. We expect that our approach could be re-cast in terms of other non-interference style frameworks such as [6], [22], [24]. This would provide access to a wide range of practical results on composition, verification, and so forth.

References

[1] E. Bertino et al. An authorization model and its formal semantics. In *Proceedings of the European Symposium on Research in Computer Security*, pages 127–142. Springer LNCS 1485, 1998.

[2] K.J. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153 Rev 1 (ESD-TR-76-372), MITRE Corp Bedford MA, 1976.

[3] W.E. Bobert and R.Y. Kain. A practical alternative to hierarchical integrity properties. In *Proceedings of the National Computer Security Conference*, pages 18–27, 1985.

[4] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security models. In *Proceedings Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, April 1987.

[5] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[6] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.

[7] S.N. Foley. A universal theory of information flow. In *Proceedings 1987 IEEE Symposium on Security and Privacy*, pages 116–121. IEEE Computer Society Press, 1987.

[8] S.N. Foley. The specification and implementation of commercial security requirements including dynamic segregation of duties. In *ACM Conference on Computer and Communications Security*, pages 125–134, 1997.

[9] S.N. Foley. Evaluating system integrity. In *Proceedings of the ACM New Security Paradigms Workshop*, 1998.

[10] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proceedings 1984 IEEE Symposium on Security and Privacy*, pages 75–86. IEEE Computer Society, 1984.

[11] P. Greve, J. Hoffman, and R.E. Smith. Using type enforcement to assure a configurable guard. In *Proceedings of the 13th. Annual Computer Security Applications Conference*, pages 146–153. ACM Press, 1997.

[12] J.L. Jacob. The basic integrity theorem. In *Proceedings of the Computer Security Foundations Workshop IV*, pages 89–97. IEEE Computer Society Press, June 1991.

[13] J.L. Jacob. The varieties of refinement. In J. M. Morris and R. C. Shaw, editors, *Proceedings of the 4th Refinement Workshop*, pages 441–455. Springer-Verlag, 1991.

[14] J.L. Jacob. Basic theorems about security. *Journal of Computer Security*, 1(4):385–411, 1992.

[15] S. Jajodia et al. A logical language for expressing authorizations. In *Symposium on Security and Privacy*, pages 31–42. IEEE Press, 1997.

[16] J.C. Laprie(ed.). *Dependability: Basic Concepts and Terminology*. Springer Verlag, 1992. IFIP WG 10.4-Dependable Computing and Fault Tolerance.

[17] J. McLean. A general theory of composition for trace sets closed under selective interleaving functions. In *Proc. IEEE Symposium on Research in Security and Privacy*, pages 79–93, 1994.

[18] R. Needham. Later developments at Cambridge: Titan, cap and the cambridge ring. *Annals of the History of Computing*, 14(4):57, 1992.

[19] L.C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[20] A.W. Roscoe. Using intensional specifications of security protocols. In *Proceedings of the Computer Security Foundations Workshop*, pages 28–38. IEEE Press, 1996.

[21] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[22] A.W. Roscoe, J.C.P. Woodcock, and L. Wulf. Non-interference through determinism. *Journal of Computer Security*, 4(1), 1995.

[23] P.Y.A. Ryan. Mathematical models of computer security. In *Foundations of Security Analysis and Design*, number 2171 in LNCS. Springer, 2000.

[24] P.Y.A. Ryan and S.A. Schneider. Process algebra and non-interference. In *IEEE Computer Security Foundations Workshop*, pages 214–227, 1999.

[25] R. Sandhu. A perspective on integrity mechanisms. In *Proceedings of the Computer Security Applications Conference*, page 279, 1989. Panel Discussion paper.

[26] R Sandhu et al. Role based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[27] A. Schaad and D. Moffett. The incorporation of control principles into access control policies. In *Workshop on Policies for Distributed Systems and Networks*, Bristol, UK, 2001.

[28] F.B. Schneider. Enforcable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, February 2000.

[29] R. Simon and M. Zurko. Separation of duty in role based environments. In *Proceedings of the Computer Security Foundations Workshop*, pages 183–194. IEEE Press, 1997.

[30] A.C. Simpson. *Safety through Security*. PhD thesis, Oxford University, Computing Laboratory, 1996.

[31] D. Sutherland. A model of information. In *Proceedings 9th National Computer Security Conference*. U. S. National Computer Security Center and U. S. National Bureau of Standards., 1986.

[32] U. S. Department of Defense. Integrity-oriented control objectives: Proposed revisions to the trusted computer system evaluation criteria (TCSEC). Technical Report DOD 5200.28-STD, 1991.

[33] D. Weber. Specifications for fault-torerance. Technical Report 19-3, Odyssey Research Associates, Ithaca, NY, 1988.

Appendix

A. Communicating Sequential Processes

In the traces model of CSP [21] the behavior of a process is represented by a prefix-closed set of event traces. If P is a process then $traces(P) \subseteq (\alpha P)^*$ where A^* is the set of all possible sequences of events from the set of events A. The alphabet αP of process P is the set of events that P is defined in terms of. In the traces model two processes are considered equivalent if their traces are the same, that is, $P = Q \Leftrightarrow traces(P) = traces(Q)$.

The subset of the CSP notation used in this paper is defined as follows.

$$traces(Stop_A) = \{\langle\rangle\}$$

The process $Stop_A$ is a deadlocked process that is not willing to engage in any event from its alphabet A. Note that we often omit subscript A if no ambiguity arises.

$$traces(Run_A) = A^*$$

Process Run_A is willing to engage in any event in alphabet A.

$$traces(a \to P) = \{\, t : traces(P) \bullet \langle a \rangle \frown t \,\} \cup \{\langle\rangle\}$$

This process is willing to first engage in event a and then behave like process P. For example, the process $(a \to b \to Stop_{\{a,b\}})$ describes a process that is willing to first engage in event a, and then to engage in event b and then deadlock. Processes may be defined recursively, for example, $P = (a \to b \to P)$. This equation has a unique fixed-point: the process that repeatedly alternates between being willing to engage in event a and willing to engage in event b.

$$traces(P \,\Box\, Q) = traces(P) \cup traces(Q)$$

Process $P \,\Box\, Q$ is willing to behave like P or like Q. If the events that P and Q are initially willing to engage in are different, then the environment of $P \,\Box\, Q$ selects between P and Q by engaging in the associated event. For example, the process

$$S = (a \to S1) \,\Box\, (b \to S2)$$

is willing to initially engage in either event a or b; if the environment selects (engages in) b then the subsequent process behavior is $S2$. In the traces model, if the environment selects an event that both P and Q are willing to engage in, then the resulting behavior is a choice between the subsequent behaviors within P and Q. For example, given process S above and

$$T = (b \to T1) \,\Box\, (c \to T2)$$

then if the environment of the process $S \,\Box\, T$ engages in b then the subsequent behavior of the process is $S2 \,\Box\, T1$.

$$\begin{aligned} traces(P \| Q) = \{\, t : (\alpha P \cup \alpha Q)^* \\ \mid t \upharpoonright \alpha P \in traces(P) \wedge t \upharpoonright \alpha Q \in traces(Q) \,\} \end{aligned}$$

Process $P \| Q$ is the parallel composition of processes P and Q with synchronization on common events in $\alpha P \cap \alpha Q$. For example, given S above, then: $S \| Stop_{\{a,b\}} = Stop_{\{a,b\}}$, since $Stop_{\{a,b\}}$ is not willing to synchronize on any event; $S \| Run_{\{a,b\}} = S$, since $Run_{\{a,b\}}$ is willing to synchronize on any event offered by S, and $S \| Stop_{\{a\}} = (b \to (S2 \| Stop_{\{a\}}))$, since $Stop_{\{a\}}$ deadlocks event a but does not constrain the engaging of other events.

In the paper we use indexed forms of concurrent composition and external choice. Process $(\|_{i:I} P(i))$ corresponds to the concurrent composition of each $P(i)$ indexed over $i : I$; Process $(\Box_{i:I}\, P(i))$ corresponds to the external choice of each $P(i)$ indexed over $i : I$. A prefixed process $(i : P)$ is the process P with its events prefixed by label i. Thus, for example, we have

$$\begin{aligned} &\Box_{i:\{0,1\}}\, (i : (a \to b \to Stop)) = \\ &\quad (0.a \to 0.b \to Stop) \,\Box\, (1.a \to 1.b \to Stop) \end{aligned}$$

B. Proof of Incremental and Composition Rules

Lemma 1: Given processes P and Q and interface E then

$$\begin{aligned} &(P \| Q)@E \subseteq (P@E) \| (Q@E) \wedge \alpha P \cap \alpha Q \subseteq E \\ &\quad \Rightarrow (P \| Q)@E = P@E \| Q@E \end{aligned}$$

PROOF: Follows from the trace semantics of $\|$. □

Theorem 1: Given systems S and P, and their corresponding infrastructures $\overline{I}_S$ and $\overline{I}_P$, then

$$(R \sqsubseteq^E S \| \overline{I}_S \wedge S \sqsubseteq^{\alpha S} P \| \overline{I}_P) \Rightarrow R \sqsubseteq^E (P \| \overline{I}_P \| \overline{I}_S)$$

PROOF: If $S \sqsubseteq^{\alpha S} P \| \overline{I}_P$, then $t \in traces(P \| \overline{I}_P) \Rightarrow t \upharpoonright \alpha S \in traces(S)$, implies that $t \in traces(P \| \overline{I}_P \| \overline{I}_S) \Rightarrow t \upharpoonright (\alpha S \cup \alpha \overline{I}_S) \in traces(S \| \overline{I}_S)$. Thus, $S \sqsubseteq^{\alpha S} P \| \overline{I}_P$ implies that $(P \| \overline{I}_P \| \overline{I}_S)@(\alpha S \cup \overline{I}_S) \subseteq traces(S \| \overline{I}_S)$, and since $E \subseteq \alpha R \subseteq \alpha S \cup \alpha \overline{I}_S$, then it follows that $(P \| \overline{I}_P \| \overline{I}_S)@E \subseteq (S \| \overline{I}_S)@E$ and from the hypothesis $(S \| \overline{I}_S) \subseteq traces(R)$, and by transitivity of $\subseteq$ the theorem follows. □

Theorem 2: Given requirements R and R' and systems S and S' and an interface E such that $\alpha R \cap \alpha R' \subseteq E$, then

$$(R \sqsubseteq^E S \wedge R' \sqsubseteq^E S') \Rightarrow R \| R' \sqsubseteq^E S \| S'$$

PROOF: If $S@E \subseteq R@E$ and $S'@E \subseteq R'@E$, then it follows that $S@E \| S'@E \subseteq R@E \| R'@E$. Since, by definition $\alpha R \subseteq \alpha S$ and hypothesis $\alpha R \cap \alpha R' \subseteq E$, we have $E \subseteq \alpha S$ and, similarly, $E \subseteq \alpha S'$. Lemma 1 implies that $(S \| S')@E \subseteq S@E \| S'@E$ and since $\alpha R \cap \alpha R' \subseteq E$ then $(R \| R')@E = R@E \| R'@E$. Thus, $(S \| S')@E \subseteq (R \| R')@E$ and the theorem follows.

We should note that if $\alpha R \cap \alpha R' \subseteq E$ does not hold then, from Lemma 1, $(P \| Q)@E = P@E \| Q@E$ does not necessarily hold and thus $R \sqsubseteq^E S \wedge R' \sqsubseteq^E S' \Rightarrow R \| R' \sqsubseteq^E S \| S'$ does not hold in general. □

Simon N. Foley is a Statutory Lecturer in Computer Science at University College Cork where he teaches and directs research on computer security. He serves on the editorial board of the Journal of Computer Security and has served as Program chair of the IEEE Computer Security Foundations Workshop and the ACM New Security Paradigms Workshop. His reseesearch interests include formal methods, security models and trust management