Information Integrity Policies

Peng Li   Yun Mao   Steve Zdancewic*

University of Pennsylvania

Abstract. Information integrity policies are traditionally enforced by access control mechanisms that prevent unauthorized users from modifying data. However, access control does not provide end-to-end assurance of integrity. For that reason, integrity guarantees in the form of noninterference assertions have been proposed. Despite the appeals of such information-flow based approaches to integrity, that solution is also unsatisfactory because it leads to a weaker notion of integrity than needed in practice.
This paper attempts to clarify integrity policies by comparing and contrasting access control vs. information flow, integrity vs. confidentiality policies, and integrity vs. availability policies. The paper also examines data invariants as a way to strengthen integrity. The result is a better classification of information-integrity policies.

1 Introduction

Information integrity is a critical issue in computer security, and integrity policies that seek to prevent accidental or malicious destruction of information have long been recognized as important. However, the concept of integrity is difficult to capture and context dependent. Pfleeger's textbook, *Security in Computing*, [26] defines integrity to mean that data is:

– precise	– modified only in acceptable ways
– accurate	– modified only by authorized people
– unmodified	– modified only by authorized processes
– consistent	– meaningful and correct

Traditionally, information integrity has been supported by security models based on *discretionary access control* via access control lists or capabilities. These models mainly provide the authorization component of integrity requirements. However, such enforcement mechanisms are not adequate to deal with situations in which important data in the system may be affected by untrusted sources of information.

For example, the programmer may use a format string provided by the user to print some information, possibly creating a vulnerability to the well-known *buffer overflow attack*. Other examples include using a value received from the network as an array index, or executing a piece of code downloaded from untrusted Web

* Email: `{lipeng, maoy, stevez}@cis.upenn.edu`

sites. Prior to using dangerous data, the programmer must carefully verify it against possible attacks to assure safety.

Because these potential integrity violations stem from how untrustworthy data propagates through the system, it is tempting to generalize access-control models of integrity to *information-flow* models, which have been widely applied to the problem of protecting confidential information [28]. In fact, it has been observed that information-flow based integrity can be treated as the formal dual to confidentiality [2]. Yet it is also known that doing so yields a somewhat unsatisfactory (and potentially inadequate) definition of security [6, 22].

The goal of this paper is to investigate integrity policies and their relationship to other existing information security policies. The paper is intended to serve as map of the technical landscape surrounding integrity policies; it suggests that integrity be classified into *program correctness*, *noninterference*, and *data invariant* policies. The main contributions of the paper include:

- A critical comparison between confidentiality and integrity when treated as formal duals. This comparison is carried out in the context of Myers and Liskov's *decentralized label model* (DLM) [22].
- A comparison of information-flow style integrity policies as enforced by static program analysis to data invariants (in the style of Clark and Wilson [6]). This comparison is motivated because treating integrity as purely dual to confidentiality leads to weak guarantees.
- A comparison of integrity policies to availability policies.
- Some suggestions for possible future research directions.

2 An aside on program correctness

Ideally, integrity, like any other program property, would be described by a specification and enforced by proving that the implementation satisfies the specification. Such an approach captures the "meaningful and correct" data requirement of Pfleeger's definition. These specifications are especially difficult to pin down and can perhaps best be formalized by requiring *program correctness*, which can be defined as the following [3]: Let π denote (a specification of) a computational decision or search problem. For x an input to π, let $\pi(x)$ denote the output of π. For a deterministic program f, we say that

- The program f is *correct*, if and only if $\forall x, f(x) = \pi(x)$.
- The output $f(x)$ is *correct*, if and only if $f(x) = \pi(x)$.

Program verification [15, 4] aims at proving the correctness of programs with respect to their specifications. *Program correctness checkers* [3] are programs that verify the correctness of the program output.

Due to their expense, program verification and correctness checkers have not been widely applied in practice. This paper instead focuses on formal definitions of integrity policies based on *noninterference* and *data invariants*. These definitions can cover almost all of the meanings of integrity given by Pfleeger and have more tractable enforcement strategies.

3 Information flow

3.1 Access control vs. information flow

In computer security, access control has been widely used to manage the permissions of users to access the objects. *Capabilities* [8, 36] and *access control lists* [17] are often used to implement such policies. In *discretionary access control* (DAC) [17, 13], the owner of an object controls the access to the object by granting and revoking access to other users. *Stack inspection* [35, 11] is a more recently developed dynamic mechanism for enforcing access control checks to sensitive system calls from untrusted code. Both DAC and stack inspection are insufficient in the context of a running system—they do not control how the data is used after granting access to the user.

Information-flow policies [12, 20, 28] are end-to-end security policies that provide more precise control of information propagation than access control models. With access control, a file is accessible to some principal only if the appropriate read permission is specified in the access control list. However, once the file is read, the data can be used in any arbitrary way. Information-flow policies aim to solve this problem by granting file access to only those processes that will not leak confidential data. The policy enforcement is thus extended to the end systems, providing more security than access control alone.

Information flow can be defined as a set of *noninterference* assertions [12, 20]. Intuitively, noninterference requires that high-security information does not affect the low-security observable behavior of the system. Such policies are especially useful to protect data confidentiality, where the goal is to ensure that secret data does not influence public data.

Formally, noninterference for confidentiality can be defined as the following[1]:

- Suppose there are two security levels of data: *secret* and *public*. Let the program f take as input (i_{secret}, i_{public}) and produce the output (o_{secret}, o_{public}).
- Define *noninterference* as a property of the program: f is noninterfering iff

$$\forall a, a' : \ f(a, b) = (c, d) \ \Rightarrow \ f(a', b) = (c', d)$$

That is, nothing about i_{secret} can be learned by only observing o_{public}.

Recent studies have shown that language-based techniques are a promising approach to enforcing noninterference. In particular, static program analysis [7, 33, 14, 21, 27] has demonstrated advantages of little run-time overhead, the capability of managing implicit information flows, and provable guarantees.

3.2 Decentralized label model

As a concrete point of comparison between information-flow definitions of confidentiality and integrity policies, we use Myers and Liskov's *decentralized label*

[1] This simple definition of noninterference is suitable for our discussion—many other more refined variants of noninterference exist in the literature. See the survey by Sabelfeld and Myers [28] for a summary.

model (DLM) [22]. The DLM addresses weaknesses of previous information-flow control models in a distributed setting containing untrusted code or users. It allows users to precisely control information flows, and it also accommodates mutual distrust and selective declassification.

Principals and Labels To address privacy[2] for mutually distrusted users and groups, the DLM uses principals as a central concept. *Principals* are the authority entities such that information is owned, read and written by them. For example, each user or group account in a Windows/Unix machine is a principal.[3]

Labels are the confidentiality requirements that the principals state about their data. A label consists of several policies, each of which is enforced by the decentralized label model. A *policy* is written as $\{o : r_1, r_2, \cdots, r_n\}$, where o represents the *owner* principal who owns the policy; $r_1, \cdots, r_n$ are the *readers*, meaning that the owner o gives $r_1, \cdots, r_n$ permissions to read the data.

A *composite label* consists of zero, one or several policies, e.g. $\{o_1 : r_1, r_2; o_2 : r_1, r_3\}$. This label says the data is owned by both o_1 and o_2. Owner o_1 permits r_1 and r_2 to read it, and owner o_2 permits r_1 and r_3 to read it. To satisfy both requirements, the only effective reader is r_1.

Labels represent different security levels. It is easy to see that label $L_1 = \{o_1 :\}$ represents a stricter policy than label $L_2 = \{o_1 : r_1\}$ because o_1 does allow r_1 to access the data in L_2, but does not in L_1. Such a label relationship is written as $L_2 \sqsubseteq L_1$, and is read as L_1 is more restrictive than L_2. It is shown by Myers and Liskov that all the labels form a distributive lattice, and have a partial order relation in the lattice [22].

When a program tries to compute a result from two values labeled with L_1 and L_2, the result should have the least restrictive label L that enforces all the privacy concerns specified by L_1 and L_2. Namely, $L_1 \sqsubseteq L, L_2 \sqsubseteq L$ and $\forall L'$ such that $L_1 \sqsubseteq L'$ and $L_2 \sqsubseteq L'$ we have $L \sqsubseteq L'$. This least restrictive label is the *least upper bound* or *join* of L_1 and L_2, written as $L_1 \sqcup L_2$. The *greatest lower bound* or *meet* of L_1 and L_2, written as $L_1 \sqcap L_2$ is defined to be the largest label less than both L_1 and L_2.

Declassification During computation, the labels on program results become increasingly restrictive, making the data unreadable. Consequently, the owners of the data sometimes need to relax their policies so that other parties can read it. This kind of label relaxation is called *declassification* [39, 38].

To make declassification reasonable, the decentralized label model permits it only when the current process is authorized to act on behalf of the principals whose policies are weakened. Because a principal can weaken only his own policy, the other owners in the label are safe—their policies are still enforced. No centralized declassification process is needed, making the DLM well suited to distributed, heterogeneously trusted systems.

[2] In this paper, the terms *privacy* and *confidentiality* are considered synonyms.

[3] For the purposes of this paper, we ignore the DLM *actsfor* relationship, which allows one principal to delegate rights to another.

4 Integrity and noninterference

We have seen noninterference policies for protecting data confidentiality. Such policies constrain: (1) Who can *read* the secret data. (2) Where the secret data will flow *to* (in the future).

Dually, integrity policies constrain: (1) Who can *write* to the data. (2) Where the data is derived *from* (in the past, the history of the data).

The analog of "public" is "untainted" and the analog of "private" is "tainted". This style of integrity policy can be defined formally using the same definition of noninterference used for confidentiality:

– Suppose there are two security levels of data: *tainted* and *untainted.* Let the program f take the input $(i_{tainted}, i_{untainted})$ and produce the output $(o_{tainted}, o_{untainted})$.
– Define *noninterference* as a property of the program: f is noninterfering iff

$$\forall a, a' : \; f(a, b) = (c, d) \;\Rightarrow\; f(a', b) = (c', d)$$

That is, the value of $i_{tainted}$ does not affect the value of $o_{untainted}$.

Integrity policies based on noninterference are useful when we want to control the source of important data. For example, in an encryption algorithm, we may want to generate randomized numbers from trusted sources and to make sure that the adversary cannot affect the value of these numbers. Information-flow control and program analysis can be used to enforce such policies based on noninterference.

Biba [2] first observed that integrity and confidentiality are duals in this way. Consequently, both can be enforced by controlling information flows.

A related concept for integrity policies is the notion of *separation of duties* (see for example Clark and Wilson's paper [6]). The idea of separation of duties is to increase data integrity by requiring that multiple principals collaborate to produce the data—forging a data item is thus more difficult. Such separation is only useful if the two parties are noninterfering in some sense. Shockley [31] and Lee [19] showed how the enforcement parts of the Clark-Wilson paper could be implemented using Bell and LaPadula [1] noninterference mechanisms. They used the Bell and LaPadula model with "partially trusted subjects" to represent the trusted program components.

More recent work by Foley gives a framework [9] for describing label-based policies that supports separation of duties. Foley [10] has also shown how to define integrity in terms of a refinement-based notion of dependability, which is closely related to standard definitions of noninterference.

5 Extending the DLM for integrity

This section discusses an integrity variant of the DLM and compares it to other integrity label models.

5.1 Integrity label models

Before introducing the full decentralized label model for integrity, we first give several simpler integrity label models for comparison and to help readers have a gradual introduction to the label models.

1. **Binary Model** In this model, there are only two possible labels: {tainted} and {untainted}. The meanings of these two labels are clear, and the order relation is {untainted} $\sqsubseteq$ {tainted}. Although the binary model is very simple, it has been used to detect format string exploits [30].
2. **Writer Model** In this model, labels are sets of principals: $\{p_1, p_2, \cdots, p_n\}$. This label means that every principal p_i in the label may have modified the data, and principals not in the set have not affected the data. Therefore, the partial order relation can be defined as: $L_1 \sqsubseteq L_2$ if and only if $L_1 \subseteq L_2$. For example, L_1={Alice} and L_2={Alice, Bob}, we have $L_1 \sqsubseteq L_2$ because data with label L_2 could be tainted by Bob, who is not a writer in L_1.
3. **Trust Model** Here, labels are again sets of principals: $\{p_1, p_2, \cdots, p_n\}$, but the meaning differs from the writer model. The interpretation is that a principal *trusts* the data with the label if and only if it is in the label set. Therefore, the partial order relation can be defined as: $L_1 \sqsubseteq L_2$ if and only if $L_2 \subseteq L_1$. The trust model is used in secure program partitioning [40, 41] to ensure data quality and to meet the robust declassification [39] requirement.
4. **Distrust Model** The distrust model is very similar to the trust model. In the distrust model, a label, being a set of principals, means that principals in the set do not trust the data. Consequently, the partial order relation is the opposite to the trust model: $L_1 \sqsubseteq L_2$ if and only if $L_1 \subseteq L_2$

Some of these models can be reduced to others. For example, to represent the binary model in the in the trust model, the label {} can represent {tainted}, and {root} can represent {untainted}. Because root should be trusted by every principal, everyone trusts that the data is untainted. It is not hard to see that trust model and distrust model can be reduced to each other.

5.2 The DLM for integrity

Label Definition Myers and Liskov [22] show how the DLM extends to support integrity in addition to confidentiality. Like a privacy label, an *integrity label* also consists of several policies. Each *policy label* is written as $\{o : w_1, w_2, \cdots, w_n\}$. Here o represents the *owner* principal, meaning that he owns the data; $w_1, \cdots, w_n$ are the *writers*, meaning that the owner o believe that only $w_1, \cdots, w_n$ could have modified the data. The owners and writers are principals drawn from the same set as in privacy labels.

A *composite label* consists of zero or more policies, e.g., $\{o_1 : w_1, w_2; o_2 : w_1\}$. This label says that the data is owned by both o_1 and o_2. Owner o_1 believes that w_1 and w_2 have written to it, and owner o_2 believes that w_1 have written to it. To satisfy both integrity constraints, the only effective writer is w_1.

We intentionally choose this integrity label definition to use the same representation syntax as that of privacy labels to make the duality between them explicit. To avoid confusion, unless the readers can easily tell from the context, we use the convention that L denotes representations, L^P denotes privacy labels and L^I denotes integrity labels. We define $\mathcal{L}$ as the set of all possible label representations. For convenience, we also define a function $R(L)$ to convert an integrity label or a privacy label to its representation, namely, $\forall L \in \mathcal{L}, R(L^P) = R(L^I)$.

Integrity Label Ordering Integrity labels have a partial order relation that expresses their relative security. For privacy labels, $L_1 \sqsubseteq L_2$ if and only if L_2 is as restrictive as L_1 or more restrictive than L_1. For integrity labels, we define $L_1 \sqsubseteq L_2$ if and only if L_2 is as tainted as L_1 or more tainted than L_1. For example, $L_1 = \{o : w_1, w_2\}$ is dirtier than $L_2 = \{o : w_1\}$ because w_2 may have tainted the data in L_1, but not in L_2. Therefore, $L_2 \sqsubseteq L_1$. If we treat L_1 and L_2 as privacy labels, and w_1, w_2 as reader principals, then $L_1 \sqsubseteq L_2$. To summarize: $\forall L_1, L_2 \in \mathcal{L}, L_1^P \sqsubseteq L_2^P$ iff $L_2^I \sqsubseteq L_1^I$ [22].

By this order definition, integrity labels also form a distributive lattice that is exactly the dual of the privacy lattice. That is, if we turn integrity lattice up side down, the two lattices will perfectly match. This property can help us to derive the integrity relabeling rules and computation rules.

Label Computation To keep track of the integrity information-flow property in the program, it is necessary to define the least upper bound operation $\sqcup$ and greatest lower bound operation $\sqcap$ in terms of integrity labels. Because of the dual relation of integrity labels and privacy labels, $R(L_1^I \sqcup L_2^I) = R(L_1^P \sqcap L_2^P)$ and $R(L_1^I \sqcap L_2^I) = R(L_1^P \sqcup L_2^P)$. For composite label $L = \{P_1; P_2; \cdots; P_n\}$, where $L_1 = \{P_1\}, L_2 = \{P_2\}, \cdots, L_n = \{P_n\}$, we have $L^P = L_1^P \sqcup L_2^P \sqcup \cdots \sqcup L_n^P$, and $L^I = L_1^I \sqcap L_2^I \sqcap \cdots \sqcap L_n^I$

Representation Power of the DLM The DLM for integrity is able to represent all the other models we have given. For example, {root: Alice, Bob} in the DLM represents {Alice, Bob} in the writer model. {Alice: ; Bob:} in the DLM represents {Alice, Bob} in the trust model. Because the distrust model is representable by the trust model, it is also representable by the DLM.

Endorsement An analog to declassification also exists for integrity labels. We call this integrity security relaxation *endorsement*. The motivation is that along with the information flow in the program, more and more data may be tainted, and the final results may be useless. It is desirable to cast tainted data to untainted data, perhaps after performing some check on that data. For example, the program may read data from the Internet, a very tainted data source. Enforcing pure noninterference would prevent such data from affecting any untainted data in the program, which is usually too restrictive in practice. Endorsement is intended for such situations. If the program can verify that the data from the network is safe and not tainted, e.g. the digital signature is correct, the program may endorse the data and allow it to flow to untainted places.

In the extended DLM, the endorsement operation is a set of relabeling rules to change the integrity labels. Because we can express mutual distrust integrity constraints, it is important that the endorsement can only weaken the integrity constraints of the current authority. In particular, if the policy of a principal does not exist in the label, he has the right to add to it.

The endorsement rule for integrity labels can be formalized as follows: If $L_2 \sqcap \{a :\} \sqsubseteq L_1$ and a is a current authority then L_1 can be endorsed to L_2.

Examples We have described how to modify the DLM to support integrity constraints. Now we present several examples in Jif [23], which is a variant of Java language that implements the DLM, to help readers understand how the model works. To distinguish integrity labels from privacy labels, we put "!" in front of labels to represent integrity labels.

```
Example 1
1: boolean !{Alice: Bob} x=true;
2: boolean !{Alice:} y = false;
3: x = y;
4: y = x; //this is wrong!
5: if (x) {
6:    y = true;  //wrong,  too!
7: }
```

```
Example 2
1: public int !{Alice:} foo(int !{} in)
2:      where authority(Alice)
3: {
4:  int result = (in>0) ? in:0;
5:  return(endorse(result, !{Alice:}));
6: }
```

In Example 1, variables `x` and `y` are given two different labels. By our integrity label definition, `!{Alice: }` $\sqsubseteq$ `!{Alice: Bob}`. Therefore, `x` is considered tainted data while `y` is untainted. Thus, assigning `y` to `x` is legal (line 3) but assigning `x` to `y` is illegal (line 4). Line 6 illustrate the implicit information flow case. Although `y` is not directly modified by any tainted data, `y`'s value depends on the condition value of the control statement in line 5. In fact, if `y` is false before line 5, then line 5 and 6 is equivalent to `y=x`. The compiler rejects such programs by adding another constraint that `y` must be dirtier than the current program-counter (`pc`) label, and in line 5 the `pc` label is augmented to `x`'s label so that the compiler can detect the fault.

In Example 2, the function `foo` takes an argument, tests whether the value is larger than 0, and returns a non-negative result. The interesting part is that the input data could be written by anyone. The program will be rejected without endorsement. With the endorsement statement, the return value of the function is fully trusted by Alice. Note that the endorsement requires Alice to be a current authority. If the current authority is Bob or someone else, the endorsement will not be granted.

6 Comparison of label models

The DLM can be symmetrically defined for both confidentiality labels and integrity labels, but they are not completely symmetric in applications. The problem lies in the motivation to use the DLM: to enforce security policies in presence of mutual distrust. Figure 1 is a comparison of the label models for both confidentiality and integrity; the details are explained in the following subsections.

	Confidentiality Labels				Integrity Labels		
	Trusted Code		Untrusted Code		Trusted Code		
	RM	DLM	RM	DLM	WM	TM/DM	DLM
Mode 1	√	√	√	√	×	×	×
Mode 2	√	√	√	√	√	√	√
Mode 3(a)	×	×	×	√	×	×	×
Mode 3(b)		√		√		√	√

RM = Reader Model TM/DM = Trust/Distrust Model
WM = Writer Model DLM = Decentralized Label Model

Fig. 1. Comparison of confidentiality and integrity label models under various failure modes (see Sections 6.1 and 6.3).

6.1 Revisiting confidentiality

To better understand the motivation of using the DLM to protect confidentiality, we compare the DLM with the *reader model*. The reader model is the dual of the writer model: each label is simply a list of principals allowed to access the information. We consider both trusted (self-written) code, and untrusted (downloaded) code, assuming all the code is typechecked before execution.

We compare the two models in the following failure modes (see the summary for confidentiality labels in the left half of Figure 1):

1. *Wrong Computation*: This is the case where the code performs some wrong computation. For the trusted, self-written code, it may be a bug in the program. For the untrusted code, it may be some malicious statements trying to leak secret information. As long as the code is annotated and typechecked, both the DLM and the reader model are safe, because the typechecking guaranteed that no information will be leaked. In fact, performing the wrong computation on the secret data only make it safer. The secret data only becomes more useless after the wrong computation.
2. *Violating Flow Constraints*: This is the case where the flow of information violates the annotated constraints. Both models are safe in this aspect, because the typechecker will detect such problems.
3. *Wrong Declassification*: This is the case where the code declassifies some secret information that should not be leaked. The reader model is not safe in this case, because there is no relationship between the authority of the code and the principals in the label. If we allow declassification here, there is no way to restrict the declassification in the code that we do not trust. For example, anyone can declassify root's password to the public.
   The DLM works better here, because the authority of the code is associated with the owners in the policies, and each principal may only weaken his/her own policy. There are two cases:
   (a) *Wrong Declassification with sufficient authority*: If a principal has the authority to weaken a policy, such a wrong declassification is not safe. For example, if root wants to declassify his own password to the public, this is a serious mistake and the typechecker will not detect it.

(b) *Wrong Declassification without sufficient authority*: If a principal does not have the authority to weaken a policy, it is safe because such a mistake can be detected by the typechecker. For example, if an applet wants to declassify root's password, such a program will be rejected because the applet does not have root's authority.

This comparison gives a clear view of the motivation to use the DLM. In the reader model, it is not safe to allow declassification when the code is not trusted. In the DLM, declassification is safe as long as each principal does not compromise his own confidentiality.

6.2 Code must be trusted to assure integrity

We know that the decentralized label model and information-flow control can be used to assure data confidentiality, even when part of the code is not trusted. As long as the program is typechecked, it is guaranteed that no secret information can be leaked.

However, when untrusted code exists, the information-flow control approach is not immediately sufficient to assure integrity. Consider the following examples:

```
Example 1
int !{untainted} add(
   int !{untainted} a,
   int !{untainted} b )
{
   return a+b;
}
```

```
Example 2
class Evil {
 int !{untainted} fileHandle;
 void setFileHandle(int !{untainted} fh){
      fileHandle = fh - fh; // Evil here
 }
 int !{untainted} getFileHandle() {
      return fileHandle;
}}
```

In the first example, the method `add` does not violate any information-flow control constraints. It takes two `untainted` arguments and calculate the sum of them, so that the result can also be annotated as `untainted`. But if the code is downloaded from untrusted sources, the integrity of the result still cannot be guaranteed, because it is easy to compromise data integrity without violating information-flow constraints. For example, the `a+b` can be modified to `a-b`, or even `a-a`, so that the result can always be garbage. More generally, we can infer that, if the code is not trusted, then we have no integrity guarantee on any information returned from it, because the adversary can manipulate the data in many ways without violating information-flow control constraints. In the second example. The `Evil` class was written by the adversary. It was supposed to store the value of a file handle and later return it. Apparently, we want to assure the integrity of that file handle, but the direct information-flow control approach cannot provide such a guarantee. The `Evil` class can be typechecked, but the value returned from the `getFileHandle()` method is completely garbage.

Therefore, the information-flow control approach must be augmented to assure the integrity of information returned from untrusted code. If such untrusted code exists, it may taint any data flowing out of it. One solution is to annotate these data as tainted by the authority of the code, which is equivalent to treating the untrusted code as tainted input channels.

6.3 Comparison of integrity label models

Now we compare the three integrity label models described earlier in this paper: the *writer model*, the *trust/distrust model*, and the *decentralized label model*. We exclude the untrusted code cases from our comparison, because none of these models can guarantee the integrity of data coming from untrusted code. We assume that all the code is written by a trusted person.

We compare the two models in the following failure modes (see the summary for integrity labels in the right half of Figure 1):

1. *Wrong Computation*: This is the case where the code performs some wrong computation. Clearly, such a mistake will break the integrity of the results, and it is not safe for any model.
2. *Violating Flow Constraints*: Just as for confidentiality, all models are safe.
3. *Wrong Endorsement*: This is the case where the code endorses some data that is actually tainted.
   The writer model is not safe in this case, because there is no relationship between the authority of the code and the principals in the label. If the programmer misuses a endorsement, the typechecker cannot find this mistake.
   In both the trust/distrust model and the DLM, the authority of the code is associated with the principals in the policies, and each principal may only weaken his/her own policy. There are two cases:
   (a) *Wrong Endorsement with sufficient authority*: If a principal has the authority to weaken a policy, such a wrong endorsement is still not safe. For example, if root wants to endorse an arbitrary integer and use it as the index to access an array, the typechecker will not detect it.
   (b) *Wrong Endorsement without required authority*: If a principal does not have the authority to weaken a policy, both the DLM and the trust/distrust models are safe. Such mistakes can be detected by typechecking.

This comparison shows that the writer model is not as powerful as the DLM, because the authority of the code is not associated with the principals in the label. However, the comparison does not show any difference between the trust model and the DLM. In fact, it seems that the trust model is sufficient to use in practical applications.

We have shown that although the DLM can be defined, manipulated and implemented in symmetric ways for integrity as well as for confidentiality, it does not provide more benefits than simpler models as the trust model or distrust model. The motivation and the power of DLM is significantly weakened by the assumption that the code is trusted.

7 Integrity and data invariants

This section considers data invariants as a means of strengthening information-flow based definitions of integrity.

7.1 Limitations of noninterference

Noninterference is not strong enough to represent the integrity policies needed in practice. In our case study of the integrity extension to the Jif language, we simply ruled out the information-flow analysis for untrusted code. We have seen such examples: An untrusted applet may have a method f that takes an untainted argument a and returns an untainted value $f(a)$. This method can be noninterfering, i.e. the output $f(a)$ only depends on a, the untainted input value. But in fact, even if f is noninterfering, $f(a)$ can be anything. It can be $a + a$ or any arbitrary constant. This will lead to serious problems that hurt the integrity of the system. What will happen if we use $f(a)$ as the index to access an array or use it as the format string for `printf()`?

The problem is that noninterference can be used only to control how the data flows in the system, but it cannot be used to control how the data is manipulated. For confidentiality, this suffices, because destroying the data only make the secrets safer, just like shredding a piece of paper with secrets on it only makes it harder to reveal these secrets. For integrity, we must consider both the data flow and its contents.

Consider the Unix file system as an example. Besides noninterference policies, which state that only authorized users can write to the file, we also want to make sure that the file system is manipulated in correct ways. The data structure of the file system should be kept consistent, i.e. the user cannot create cross-linked files or directories, the date and time of each directory item are valid (not June 32nd). Such requirements are far beyond the scope of noninterference policies, yet they are usually considered as integrity requirements.

7.2 Integrity as invariants

For a better understanding of integrity policies beyond noninterference, we need to study the concept of integrity of the contents of data. In practice, there are many examples: array access (indexes not exceeding the boundaries), message authenticity (digital signature or MAC is valid), and file systems (the data structures are consistent).

Depending on the context of the application, the integrity of data may have very different meanings. Nevertheless, they all require some property of the data, which is either "good", i.e. it satisfies the constraint or meets the specification, or "bad", i.e. the data is invalid or inconsistent. To be enforceable, such properties must be computable, so that given a specific value, we can decide whether it is good or bad. Generally, for a piece of data a, we can define the *quality* of a as a computable predicate φ on it:

$$\varphi(a) \equiv a \text{ has good quality}$$

An integrity policy on the value of data can be defined as: *an invariant φ on the quality of data under program execution.* We use the function f to represent a fragment of program such as a method or an instruction. For an integrity policy with invariant φ, we can define the quality of the program as a predicate ψ_φ on f, by checking whether it always maintains the invariant:

$$\psi_\varphi(f) \iff \forall a, \varphi(a) \to \varphi(f(a))$$

We would expect that such integrity policies be enforced by checking $\psi_\varphi(f)$. For example, if the quality of data φ is easy to compute, a simple approach to enforcing the invariant policies would be to rewrite the program f in the form

$$g(a) = \begin{cases} f(a) & \text{when } \varphi(f(a)) \\ b & \text{when } \neg\varphi(f(a)) \wedge \varphi(b) \end{cases}$$

where b is a value with good quality that satisfies φ. Suppose the invariant is a range [1..10] on variable x, then, after the execution of f, x may be 11, which is out of the range. In such a case, g could either return a default value 1 that falls within the range, or raise an exception, indicating the failure. This is exactly the approach people have used in software-based fault isolation [34], where mandatory runtime monitors and assertions are inserted into the programs. In particular, if $\varphi(a)$ holds and we choose $b = a$, the function g has the atomic effect of a transaction—the data a will either persist unchanged or become $f(a)$ if and only if $f(a)$ has good quality.

If we treat the data during program execution as traces, a security policy based on the invariant φ is a safety property [29], specifying that bad things never happen during any execution. Therefore, various enforcement mechanisms for safety properties can be used to assure integrity, such as execution monitoring with security automata [29], software-based fault isolation [34], and proof carrying code [24].

7.3 Enforcing the invariants

Clark and Wilson's model [6] used an access-control style enforcement mechanism for preserving data invariants. The model introduced the concepts of CDI (Constrained Data Item), referring to data with good quality, and TP (Transformation Procedure), referring to programs preserving the invariants on CDI. The invariants are enforced with two kinds of rules: certification rules, which state that all the TPs are certified so that they always preserve the invariants on the CDIs, and enforcement rules, saying the CDI can only be accessed via the TP and the accesses are controlled according to a certified relation (User, TP, CDI).

The certification of TP is exactly the problem of checking whether a program preserves a data invariant. In Clark and Wilson's model, the certification is usually a manual operation performed by the security officer or the system custodian. For large systems, solely relying on the manual proofs would be inefficient and error-prone. Moreover, the emerging application of mobile code brings more challenges on the integrity model: it is impractical to manually check the safety of a downloaded applet before executing it. Therefore, we need automated mechanisms to enforce the data invariants for programs.

There are two kinds of enforcement mechanisms—static and dynamic mechanisms. One tradeoff between them is that they have different information to work with—dynamic mechanisms have access to the run-time state, but static mechanisms typically have access to the full program text. Another tradeoff is that dynamic mechanisms potentially introduce more runtime overhead. The typical dynamic runtime mechanisms include software-based fault isolations [34, 32] and reference monitors [29, 18].

Static mechanisms, such as type systems and program analyses are also a promising way to specify integrity invariants. The difficulty is in verifying rich integrity invariants—properties involving arithmetic, for instance, quickly become intractable.

An important research direction is to apply these technologies to integrity invariants, potentially by combining static and dynamic approaches: type systems that ensure dynamic checks have been inserted into the system appropriately.

8 Availability vs. integrity

Recently, the Denial-of-Service (DoS) attack, emerging as one of the most serious security problems, has brought availability issues to the security community. Because the idea behind DoS attack is to make the computing resources *unavailable* to break down the system, not confidentiality, integrity or anonymity, but availability policies are violated. However, there are some similarities between integrity and availability policies.

The well-known availability definition is: Availability $= \frac{\text{MTTF}}{\text{MTTF+MTTR}}$.

MTTF stands for Mean Time to Failure and MTTR stands for Mean Time To Repair. One problem of this definition is that it tries to measure the failure of the whole system. In a large, distributed system, even when some hardware or software fails, parts of the system are still functioning. Thus, we need a finer grained definition. We propose that the system may have multiple inputs and outputs, each of which has a different availability concern. This requirement is almost inevitable for the large-scaled distributed system. Now, it is natural to make the analogy of the noninterference notion from integrity to availability. That is, just as we need that untainted data is not affected by tainted data for integrity, we need that highly available data (dependable data) does not depend on the low-available data (undependable data) during the computation.

The formal definition of noninterference for availability is as follows:

– Suppose there are two security levels of data: *undependable* and *dependable*. Let the program f take the input $(i_{undependable}, i_{dependable})$, and produce the output $(o_{undependable}, o_{dependable})$.
– Define *noninterference* as a property of the program: f is noninterfering iff

$$\forall a, a' : \; f(a, b) = (c, d) \;\Rightarrow\; f(a', b) = (c', d)$$

That is, the value of $i_{undependable}$ will not affect the value of $o_{dependable}$.

Given this formal definition, we also need to formalize the model of the multiple inputs/outputs system, and precisely answer what we mean by availability for an output. First, we simplify the system as a real-time function f: f takes a vector $\langle i_1, i_2, \ldots, i_n\rangle$ as input, and outputs another vector $\langle o_1, o_2, \ldots, o_m\rangle$. Assume that for all $\langle i_1, i_2, \ldots, i_n\rangle \in I^n$, $\text{RunningTime}(f(\langle i_1, \ldots, i_n\rangle)) \leq t_0$. That is, the running time of f, no matter what its inputs are, is bounded by time t_0. All the data the system depends on should be specified by the input parameters or have universal availability. If the input channel for i_j is available, the data can be read by the system in time t_j, which should be negligible compared to t_0, i.e. $t_j = o(t_0)$.

Secondly, in this model, we define the availability as a property on each output element of the system. The availability for an output is the a vector of probabilities, each element of which is the probability for the system to get the corresponding output element successfully in time $O(t_0)$. By this definition, in a system one can have different levels of availability for different output elements. Recent research shows that such multi-level tunable availability is useful in system design [37].

The noninterference of this model is useful at least in following two scenarios:

1. There is a well-known trade-off that making data highly available sacrifices performance. For example, in storage systems, making more replicas increases availability, but maintaining consistency degrades the performance. Therefore, noninterference is useful to guarantee that all critical data is dependable while other data may not be, permitting improved performance.
2. One possible way to fight against DoS attack is to allocate some privileged users separate, highly available resources from ordinary users [5]. When some DoS attack happens, the behavior of privileged users is not affected.

There are limitations to this noninterference definition of availability.

1. The system model above is significantly simplified from reality. System running time is typically not bounded and varies according to many factors, such as logic inside the program, input, network conditions, etc. It is unpredictable in general. Strong noninterference, which takes timing channels into account, may have application to availability research, i.e., the undependable data cannot interfere with the timing effects of dependable data.
2. The computing infrastructure, including OS, CPUs, chips, storage devices, power supplies, is heavily influential in availability issues. For confidentiality and integrity, it is natural to put the infrastructure into the trust computing base. For availability, however, how dependable the infrastructure is becomes much more important [25, 16]. One interesting question is how to practically specify the availability of underlying infrastructure.
3. A probabilistic model of availability brings difficulties to both the program analyzer and end-users who specify policies. For program analyzer, it is harder to calculate least upper bounds of security levels because they may or may not correspond to independent events. For end users, it is harder to specify availability of resources in terms of percentages than it is to specify who can see/modify the data in confidentiality/integrity.

Therefore, enforcing noninterference for availability is sometimes necessary for some applications, but definitely not sufficient.

These observations suggest a number of possible future research directions:

– Specifying availability concerns inside programs to enable static program analysis, perhaps using type systems to provide the specifications. We call for concrete specification models to address this problem. In particular, how to specify the underlying hardware availability is an important open question.
– Developing a better metric model for availability than the model defined above. We need to incorporate timing effects, the probability that the service is available, the corresponding distribution, and correlated events.
– Exploring ways to better enforce timing sensitive strong noninterference.

9 Conclusion

We examined similarities and differences between access-control and information-flow based definitions of confidentiality, integrity, and availability policies, focusing mainly on integrity.

Confidentiality, integrity, and availability can be defined as *noninterference* policies in *information-flow control* with varying degrees of success. The *decentralized label model* exhibits the duality between confidentiality and integrity. However, treating integrity as the formal dual of confidentiality leads to a weak notion of security, because integrity demands additional assurance on the quality of the data. This paper suggests *invariants* on quality of the data under program execution as a way to strengthen noninterference based integrity policies.

These observations lead us to propose a definition of integrity as *program correctness, noninterference*, and *data invariant* conditions, which yields the following categorization:

– *Program correctness*: program output is precise, accurate, meaningful and correct with respect to a specification.
– *Noninterference*: data is modified only by authorized people or processes either directly, or indirectly by means of information flow.
– *Data invariants*: data is precise or accurate, consistent, unmodified, or modified only in acceptable ways under program execution.

References

1. D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and Multics interpretation. Technical Report ESD-TR-75-306, MITRE Corp. MTR-2997, Bedford, MA, 1975. Available as NTIS AD-A023 588.
2. K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, MA, April 1977.
3. M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42(1):269–291, 1995.
4. R.S. Boyer and J.S. Moore. *The Correctness Problem in Computer Science*. Academic Press, London, 1981.

5. Jos Brustoloni. Protecting electronic commerce from distributed denial-of-service attacks. In *Proceedings of the eleventh international conference on World Wide Web*, pages 553–561. ACM Press, 2002.
6. David Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *Proc. IEEE Symposium on Security and Privacy*, pages 184–194, 1987.
7. Dorothy E. Denning and Peter J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.
8. J. B. Dennis and E. C. VanHorn. Programming semantics for multiprogrammed computations. *Comm. of the ACM*, 9(3):143–155, March 1966.
9. Simon N. Foley. The specification and implementation of 'commercial' security requirements including dynamic segregation of duties. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 125–134. ACM Press, 1997.
10. Simon N. Foley. A nonfunctional approach to system integrity. *IEEE Journal on Selected Areas in Communications*, 21(1):36–43, January 2003.
11. Cedric Fournet and Andrew Gordon. Stack inspection: Theory and variants. In *Proc. 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 307–318, 2002.
12. J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Computer Society Press, April 1982.
13. M. A. Harrison, W. L Ruzzo, and J. D. Ullman. Protection in operating systems. *Comm. of the ACM*, 19(8):461–471, August 1976.
14. Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 365–377, San Diego, California, January 1998.
15. C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12(10):576–580, October 1969.
16. James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.
17. Butler W. Lampson. Protection. In *Proc. Fifth Princeton Symposium on Information Sciences and Systems*, pages 437–443, Princeton University, March 1971. Reprinted in Operating Systems Review, 8(1), January 1974, pp. 18–24.
18. I. Lee, H. Ben-Abdallah, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Monitoring and checking framework for run-time correctness assurance. In *Proceedings of the Korea-U.S. Technical Conference on Strategic Technologies*, 1998.
19. Theodore M. P. Lee. Using mandatory integrity to enforce "commercial" security. In *IEEE Symposium on Security and Privacy*, pages 140–146. IEEE Computer Society Press, 1988.
20. John McLean. Security models and information flow. In *Proc. IEEE Symposium on Security and Privacy*, pages 180–187. IEEE Computer Society Press, 1990.
21. Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999.
22. Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, 2000.
23. Andrew C. Myers, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java information flow. Software release. Located at http://www.cs.cornell.edu/jif, July 2001.

24. George C. Necula. Proof-carrying code. In *Proc. 24th ACM Symp. on Principles of Programming Languages (POPL)*, pages 106–119, January 1997.
25. David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conf. on Management of Data*, pages 109–116. ACM Press, 1988.
26. Charles P. Pfleeger. *Security in Computing*, pages 5–6. Prentice-Hall, 1997. Second Edition.
27. François Pottier and Sylvain Conchon. Information flow inference for free. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 46–57, September 2000.
28. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.
29. Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 2001. Also available as TR 99-1759, Computer Science Department, Cornell University, Ithaca, New York.
30. Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.
31. W. Shockley. Implementing the clark/wilson integrity policy using current technology. In *Proceedings of NIST-NCSC National Computer Security Conference*, pages 29–37, 1998.
32. Chris Small. MiSFIT: A tool for constructing safe extensible c++ systems. In *Proceedings of the Third Usenix Conference on Object-Oriented Technologies*, June 1997.
33. Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.
34. R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. 14th ACM Symp. on Operating System Principles (SOSP)*, pages 203–216. ACM Press, December 1993.
35. Dan S. Wallach, Andrew W. Appel, and Edward W. Felten. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4), October 2000.
36. W. A. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor system. *Comm. of the ACM*, 17(6):337–345, June 1974.
37. Haifeng Yu and Amin Vahdat. The costs and limits of availability for replicated services. In *Proc. 18th ACM Symp. on Operating System Principles (SOSP)*, Banff, Canada, October 2001.
38. Steve Zdancewic. A type system for robust declassification. In *Proceedings of the Nineteenth Conference on the Mathematical Foundations of Programming Semantics.* Electronic Notes in Theoretical Computer Science, March 2003.
39. Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proc. of 14th IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Canada, June 2001.
40. Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *Transactions on Computer Systems*, 20(3):283–328, 2002.
41. Lantian Zheng, Stephen Chong, Steve Zdancewic, and Andrew C. Myers. Building secure distributed systems using replication and partitioning. In *IEEE 2003 Symposium on Security and Privacy.* ieee, 2003.