CS 6600 Project Report: Unmanned Aircraft System

Matthew Whitesides and Bruce M. McMillin
Department of Computer Science
Missouri University of Science and Technology, Rolla, MO 65409-0350

Abstract—The abstract goes here.

1 Infrastructure

1.1 Infrastructure Description

In its simplest form, an unmanned aircraft system (UAS), is an aircraft system that operates without an onboard pilot. UAS are either controlled remotely through some form of wireless radio communication, semi-autonomously in conjunction with a remote pilot, or fully autonomously using some form of computational intelligence as navigation. These intelligent crewless vehicles have many potential uses in the defense sector, including surveillance, strategic mission execution, aerial sustainability support, and training systems, to name a few. These aircraft fall under the umbrella of cyber-physical systems (CPS), merging the intelligent navigation processes, system health monitoring, and communication with the physical mobile aerial vehicle.

We will design a proposed infrastructure security policy for the Boeing MQ-25 unmanned aircraft system, but it will also apply to similar mission support drones. The MQ-25 is an unmanned aircraft system designed for the U.S. Navy, and it provides autonomous refueling capability for the Boeing F/A-18 Super Hornet, Boeing EA-18G Growler, and Lockheed Martin F-35C fighters. This capability extends the combat range of the supported aircraft, seamlessly and semi-anonymously navigating to the plane, refueling, and returning to base. MQ-25 is the first unmanned aircraft to support aerial refueling another aircraft and is currently in the flight test phase of development [1], making it the perfect system to analyze security impacts for current and future unmanned aircraft systems. For the purposes of this document we will take the basic idea of the MQ-25 and model a UAS system referred to as the MX-01 UAS.

1.1.1 Infrastructure Security Policy

Our infrastructure security policy breaks down the various actions a system user can perform using the following terms.

- *Subject*: Any entity that contains the proper rights can request the UAS perform operations, access objects, or grant rights to another subject.
- *Object*: An entity that is part of the UAS functionality or data that does not have control over another entity.
- *Rights*: A property assigned to a subject that defines its right to access an object or grant permissions to another subject.

TABLE 1
Description of rights over objects in the UAS.

Right	Description
Owns (O)	The owner of the given object.
Read (R)	Can observe the given object.
Write (W)	Can modify the given object.
Execute (E)	Can execute the functionality of the given object.
Grant (G)	Can grant a given right to another subject.
Control (C)	Can control a given system object.
Delete (C)	Can delete a given object or right.
Create (C)	Can create a new subject or object.

Table 1 describes the rights and their associated functionality. Table 2 breaks down the subject roles involved in operating the UAS during a refueling mission. Table 3 contains the access control matrix (ACM) showing each Subject's rights over the objects.

1.2 HRU

The Harrison, Ruzzo, Ullman security model (HRU) establishes a finite set of mono-operational procedures our system can perform on subjects and objects. Given our set of rights and ACM, we will establish a set of commands available to the system that acts upon the subjects and objects in the system. The commands will consist of mono-operational modifications and pre-condition checks. Therefore given these sets, we can show how a specific set of commands can create a rights leakage.

The following shows the basic HRU commands related to our UAS mission. A fundamental UAS refueling mission follows these basic steps.

1) UAS is verified flight-ready by the MC.
2) The PC plans the mission.
3) The PC and IP execute the mission.
4) During the flight, the PC and IP execute the ANC and RO as needed.
5) After the refueling operation, the UAS returns to base, and the flight is debriefed.
6) PC, IP, or MC download the flight data from the UAS flight recorder.

TABLE 2
Description of actor subject roles during a UAS refueling mission.

Subject	Description
Pilot Commander (PC)	The primary remote pilot of the UAS during the mission.
Instructor Pilot (IP)	Assists the Pilot Commander and can pilot the UAS if given permission from the PC or MC.
Maintenance Crew (MC)	Handles work orders created by the PC, IP, or FDA, responsible for the maintenance of the UAS.
Flight Data Admin (FDA)	Handles and analyses all mission flight data.
External Contractor (Bad Actor) (EC)	Has a similar job to the MC however only has read rights to the FED.

TABLE 3
Initial UAS Refueling Mission Access Control Matrix

	PC	*IP*	*MC*	*FDA*	*EC*	**ANC**	**RO**	**FED**	**RTD**	**FRS**
Pilot Commander (PC)	O,R,W	R,W	R,W	R,W	R,W	O,R,W,E,G,C	O,R,W,E,G,C	R,W,E,G,C	R,W,E,G,C	R,W,E,G,C
Instructor Pilot (IP)	R	O,R,W	∅	∅	∅	R,W,E,C	R,W,E,C	R,E	R,E	R,E
Maintenance Crew (MC)	∅	∅	O,R,W	∅	R	∅	∅	R,E	R,E	R,E
Flight Data Admin (FDA)	∅	∅	R	O,R,W	O,R,W	∅	∅	O,R,W,E,G,C	O,R,W,E,G,C	O,R,W,E,G,C
External Contractor (Bad Actor) (EC)	∅	∅	∅	∅	O,R,W	∅	∅	∅	R	∅
Autonomous Navigation Control (ANC)	∅	∅	∅	∅	∅	R,W,E,C	R	R	R	∅
Refueling Operation (RO)	∅	∅	∅	∅	∅	∅	R,W,E,C	R	R	∅
Flight Engine Data (FED)	∅	∅	∅	∅	∅	∅	∅	R,W,E,C	∅	∅
Refueling Tank Data (RTD)	∅	∅	∅	∅	∅	∅	∅	∅	R,W,E,C	∅
Flight Record System (FRS)	∅	∅	∅	∅	∅	∅	∅	∅	∅	R,W,E,C

TABLE 4
ACM After Create Flight Record

	PC	*IP*	*MC*	*FDA*	*EC*	**ANC**	**RO**	**FED**	**RTD**	**FRS**	**FR**
PC	O,R,W	R,W	R,W	R,W	R,W	O,R,W,E,G,C	O,R,W,E,G,C	R,W,E,G,C	R,W,E,G,C	R,W,E,G,C	R,W,E,G,C
IP	R	O,R,W	∅	∅	∅	R,W,E,C	R,W,E,C	R,E	R,E	R,E	R,E
MC	∅	∅	O,R,W	∅	R	∅	∅	R,E	R,E	R,E	R,E
FDA	∅	∅	R	O,R,W	O,R,W	∅	∅	O,R,W,E,G,C	O,R,W,E,G,C	O,R,W,E,G,C	O,R,W,E,G,C
EC	∅	∅	∅	∅	O,R,W	∅	∅	∅	R	E,W	∅
ANC	∅	∅	∅	∅	∅	R,W,E,C	R	R	R	∅	∅
RO	∅	∅	∅	∅	∅	∅	R,W,E,C	R	R	∅	∅
FED	∅	∅	∅	∅	∅	∅	∅	R,W,E,C	∅	∅	∅
RTD	∅	∅	∅	∅	∅	∅	∅	∅	R,W,E,C	∅	∅
FRS	∅	∅	∅	∅	∅	∅	∅	∅	∅	R,W,E,C	∅
FR	∅	∅	∅	∅	∅	∅	∅	∅	∅	∅	R,W,E,C

7) A flight record (FR) is created that contains flight tracking information, engine usage data, and various UAS health status.
8) This flight record is uploaded by the PC, IP, MC, or EC to the record-keeping system.
9) Mission is completed.

With this in mind, we have the following basic HRU commands available for post-flight maintenance (steps 5 - 9). We will then show how improper use of these commands can lead to a rights leakage with our external contractor standing in as our "bad actor" gaining a leak of Integrity and confidentiality rights beyond our initial ACM.

The first command is a generic *Grant r Rights* that allows a subject to grant any right they have over a subject/object to another subject/object.

```
command grant_r_right(r, o, p, q)
  if grant in A[p, o] and r in A[p, o]
  then
    enter r into A[q, o];
end
```

Next we have command *Make Owner* allowing a subject p to make another subject q the owner of a object o they currently have owner rights over.

```
command make_owner(p, q, o)
  if owns in A[p, o]
  then
    enter owns into A[q, o];
end
```

When the *PC* and *IP* return from a flight they or a *MC* will read the flight data and create a new object **Flight Record (FR)** holding the flight data, available to subjects who have **FED** access. After running this command, our

ACM would transition to a new state similar to Table 4, representing the subjects and objects involved in the create flight record procedure. This command will also give rights to the FRS to control the flight record data once uploaded.

```
command create_flight_record(p)
  if create in A[p, FED]
  then
    create object FR;
    enter own into A[p, FR];
    enter delete into A[p, FR];
    enter read into A[p, FR];
    enter grant into A[p, FR];
end
```

After creating a flight record which may be done by a PC, IP, or MC, they may choose to give the task of processing and uploading the flight record to an external external contractor and execute the following command to provide them with access to the record. In the command *Grant Flight Record Access*, *p* is the subject granting the right to subject *q*, for the *fr* flight record.

```
command grant_flight_record_access(p, q,
    fr)
if own in A[p, fr]
then
  enter read into A[q, FR];
  enter write into A[q, FR];
  enter execute into A[q, FR];
end
```

Finally, the flight record needs to be uploaded to our flight record system using the following *Upload Flight Record* command with *p* being the subject executing the command and *fr* being flight record to upload.

```
command upload_flight_record(p, fr)
if own in A[p, fr] and read in A[p, FRS]
then
  enter read into A[FRS, FR];
  enter write into A[FRS, FR];
  enter execute into A[FRS, FR];
  enter control into A[FRS, FR];
end
```

Similar to *Create Flight Record* a subject may need to delete or update a flight record from the system.

```
command
    delete_flight_record_from_system(p, FR)
if delete in A[p, FR]
then
  delete read from A[FRS, FR];
  delete write from A[FRS, FR];
  delete execute from A[FRS, FR];
  delete control from A[FRS, FR];
end
```

If any modifications need to be made to the FRS, the following command begins an update transaction and ends one for a given subject and flight record.

```
command update_flight_record_system(p, FR)
if own in A[p, FR]
  then
    enter read into A[p, FED];
    enter read into A[p, FR];
    enter write into A[p, FED];
    enter write into A[p, FR];
  end
```

1.3 Rights Leakages

1.3.1 Confidentiality

For our example of a confidentiality attack, we simulate a scenario where our *External Contractor* is a bad actor seeking leaked rights beyond the initial ACM utilizing the following HRU commands available for the UAS.

The following sequence of commands is typical among a mission debriefing process.

```
create_flight_record(Maintenance Crew
    (MC));
grant_flight_record_access(Maintenance
    Crew, External Contractor (EC), Flight
    Record (FR));
upload_flight_record(External Contractor
    (EC), Flight Record (FR));
```

However, when running *upload_flight_record*, the EC will find they do not have "own" rights over the record, which is required so that they will go back to the original MC and as to grant them execute privileges to the FRS system. The MC will execute the commands to make EC have permission to upload the FR.

```
make_owner(MC, EC, FR);
command grant_r_right(execute, FRS, MC,
    EC);
```

This command, unfortunately, will lead to a leak as the contractor (EC) now has the execute privileges over the FRS, which they did not initially and is not intended and can lead to other flight records confidential information being exposed to the EC. Table 4 shows the leaked rights in red.

1.3.2 Integrity

In an attempt at an integrity leak, our EC may seek to modify existing data and, to get permission to do so, may attempt the following sequence.

```
create_flight_record(Maintenance Crew
    (MC));
upload_flight_record(External Contractor
    (EC), Flight Record (FR));
grant_flight_record_access(Maintenance
    Crew, External Contractor (EC), Flight
    Record (FR));
make_owner(MC, EC, FR);
update_flight_record_system(EC, FR);
```

However, upon reviewing the FR, the MC notices an issue in the data and wants the EC to update the flight record system. Unfortunately, *Update Flight Record System* checks only for rights to the given flight record and not the initial systems, therefore, giving the EC *write* access to the FRS system, which could lead them modifying the FRS data ruining its integrity. Table 4 shows the result of executing the confidentiality and integrity leaks.

Fig. 1. Initial Iteration T-G Model Graph

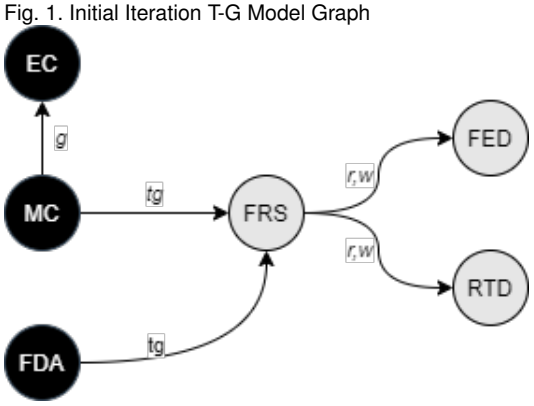


Fig. 2. Iteration 2 T-G Model Graph

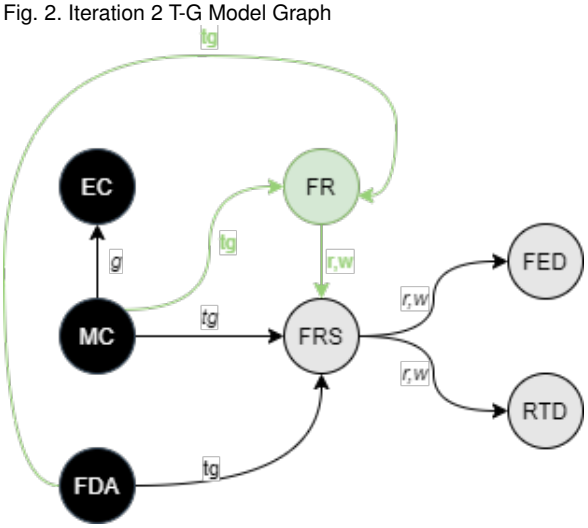
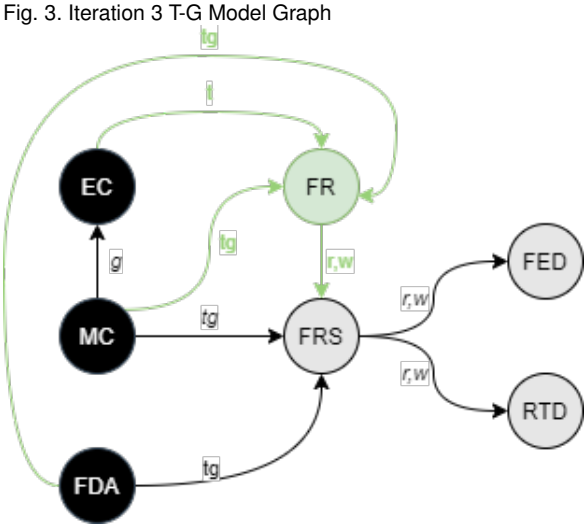


Fig. 3. Iteration 3 T-G Model Graph



2 Survey

2.1 Take-Grant

Next, we take a look at the Take-Grant (TG) model interpretation of our rights leakages. This model represents our system as a directed graph, with our subject and objects represented as vertices (black for subjects, white for objects). Edges represent the rights on vertex has over another, where special take (*t*) and grant (*g*) representing the ability of a subject or object to give or obtain rights from another subject or object. In our case, subjects can execute a grant HRU command representing the grant right, and take can be achieved by rights automatically allocated by granting access to specific objects (i.e., the FRS).

This models our HRU example in terms of a TG model where the *Flight Record* created object has *read, write* rights over the *Flight Record System* that our **External Contractor (EC)** can utilize to gain *read, write* access to the *Flight Engine Data*.

Our bad actor could achieve this leak through the following TG commands.

1) **MC** creates object **FR**.
2) **MC** grants (*t* to **FR**) to **EC**.
3) **EC** takes (*r,w* to **FRS**) from **FR**.
4) **EC** takes (*r,w* to **FED**) from **FRS**.

Figure 1 shows the initial state of rights among actors and objects in the **FRS** system.

Figure 2 shows the state of rights after the **MC** runs the HRU command *create_flight_record(MC)* which a new **FR** object.

Figure 3 shows the state after the **EC** leads the **MC** into giving them *t* rights over **FR** they can exploit the system to take our leaked rights to **FED** through the **FRS**. This would be established after the command *grant_flight_record_access(MC, EC, FR)*.

Therefore allowing EC to execute *update_flight_record_system(EC, FR)* which leaks access to the FED as shown by the state in Figure 4.

Using this TG protection model instead of the simple HRU commands, we demonstrate that the safety question is decidable in linear time based on our graph consisting of seven nodes.

2.2 Bell-LaPadula

The Bell-LaPadula Model (BLP) focuses on establishing access control confidentiality through security levels and categories. BLP focuses on two simple rules in a security policy within a given ACM. The simple security property states a subject can not read an object at a higher security level. The star property states that a subject may not write to any object at a lower security level.

The primary limitation we see when implementing a BLP model into our existing policy is our external contractor and maintenance controllers. We need access to the flight records, so they require at least *confidential* clearance levels. These restrictions make it hard to read the flight record system at a *secrete* level and our FRS to write to the flight records.

For our BLP implementation, we first define the security classifications for our subjects and objects. Given we are dealing with military records, it makes sense to use the standard government security levels. In addition to the security level, subjects and objects will fall into categories based upon the appropriate work unit. Table 5 shows the security clearance levels and the subjects/objects at the

Fig. 4. Final Iteration T-G Model Graph

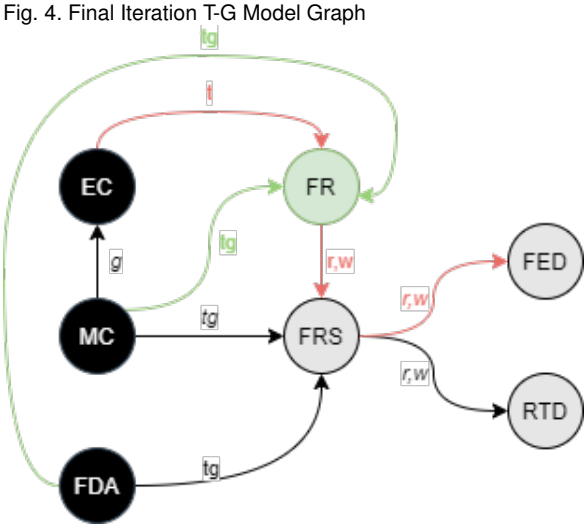


TABLE 5
Security Classifications for the UAS

Clearance Level	Subjects	Objects
Top Secret (TS)	PC, FDA	ANC, RO
Secret (S)	IP	FRS, RTD
Confidential (C)	MC, EC	FED, FR
Unclassified (UC)		

given levels. Table 6 shows the work unit categories of each subject/object.

Now that we've established security classifications for our subjects and objects, we can see how the system and security policy could still allow leakage and further theft of flight record data.

```
create_flight_record(Maintenance Crew
    (MC));
grant_flight_record_access(Maintenance
    Crew, External Contractor (EC),
    Flight Record (FR));
upload_flight_record(External Contractor
    (EC), Flight Record (FR));
make_owner(MC, EC, FR);
update_flight_record_system(EC, FR);
```

1) **MC** executes *create flight record* creating flight record object **FR**.
   - This write is allowed as the object **FR** created has clearance level of *Confidential* which is greater than or equal to **MC's** level of *Confidential* and **FR** dominates **MC** by categories.

TABLE 6
Work Unit Categories for the UAS

Category	Subjects	Objects
Flight Ops (FO)	PC,IP	ANC,RO
Maintenance Ops (MO)	PC,IP,MC,FDA	FED,RTD,FR,FRS
Flight Data Ops (FDO)	PC,IP,FDA,EC	FED,FR,FRS

2) **MC** executes *upload flight record* granting the object **FRS** access to the flight record object **FR**.
   - This write is allowed as the object **FRS** created has clearance level of *Secret* which is greater than **FR's** level of *Confidential* and **FRS** dominates **MC** by categories.
3) **MC** executes *grant flight record access* granting **EC** access flight record object **FR**.
   - This read is allowed as the subject **EC** created has clearance level of *Confidential* which is greater than or equal to **FR's** level of *Confidential* and **MC** dominates **EC** by categories.
4) **MC** executes *make owner* making **EC** have owner rights over flight record object **FR**.
   - This write is allowed as the subject **EC** created has clearance level of *Confidential* which is greater than or equal to **FR's** level of *Confidential* and **FR** dominates **EC** by categories.
5) **EC** executes *update flight record system* allowing **EC** write to the flight record system object **FRS**.
   - This write is allowed as the subject **EC** created has clearance level of *Confidential* which is less than or equal to **FRS's** level of *Secret* and **FRS** dominates **EC** by categories.

As you can see, while this system does keep our bad actor **EC** from reading the confidential flight records from the flight record system, it does not prevent an integrity attack of writing up to the **FRS** and **FR**.

We can setup our BLP system state in the context of the command leakages as follows:

- $S = \{MC, EC\}$
- $O = \{FRS, FR\}$
- $P = \{owner, read, write\}$
- $C = \{Secret(S), Confidential(C)\}$
- $K = \{MO, FDO\}$
- $f_c(s) = \{(C, \{MO\}), (C, \{FDO\})\}$
- $f_c(o) = \{(S, \{MO, FDO\}), (C, \{MO, FDO\})\}$

Then as the commands are executed we can see the resulting states in Table 7.

2.3 Biba

The Biba model focuses on data integrity as opposed to the BLP model, which enforces confidentiality by restricting access. The Biba model ensures the data objects once created remain unmodified by untrusted sources by establishing integrity levels. The model achieves this by ensuring ACM transitions do not allow subjects to write/modify data above their integrity level. Subjects do not read data below their level to avoid being influenced by data below their trusted level.

These notions apply to our infrastructure well as the bad actor (External Contractor), in theory, could either read data they're not supposed to (violating confidentiality). Or the EC could modify existing flight record data (violating integrity). If the confidentiality policy is perfectly implemented, no data is accessed that is outside their security

TABLE 7
BLP System State Commands

	i_0	i_1	i_2	i_3	i_4
X	{(r, ∅), (∅, ∅)}	{(r, ∅), (r, o)}	{(∅, r), (∅, ∅)}	{(∅, o), (∅, ∅)}	{(∅, w), (∅, ∅)}
Y	{(y, n), (n, n)}	{(y, n), (y, y)}	{(y, y), (∅, y)}	{(y, y), (∅, y)}	{(∅, y), (y, y)}
Z	V_1	V_2	V_3	V_4	V_5

TABLE 8
Security Categories for the UAS

Security Category	Subjects	Objects
Archived Data (AD)	PC,MC,FDA	FRS
Operational Data (OD)	PC,IP,MC	ANC,RO,FED,RTD
Ready To Load (RTL)	FDA,MC,EC	FR,FRS

TABLE 9
Integrity Categories for the UAS

Integrity Category	Subjects	Objects
On Aircraft (IOA)	PC,IP,MC	ANC,RO,FED,RTD
Flight Record System (IFRS)	FDA,MC,EV	FRS

TABLE 10
Integrity Classifications for the UAS

Integrity Classifications	Subjects	Objects
System Programs (ISP)	∅	ANC,RO,FRS
Operational (IO)	PC,IP,FDA	∅
Maintenance (IM)	MC,EC	FRS,FED,RTD,FR

TABLE 11
Security and Integrity Levels for Work Units

Work Unit Category	Security	Integrity
Flight Ops	({TS,{OD})	(ISP,{IOA})
Maintenance Ops	({S,{AD,RTL})	(IO,{IOA,IFRS})
Flight Data Ops	({S,{OD,RTL})	(IM,{IFRS})

level. However, there's still an amount of "trust" we have in the EC and system preventing the integrity of the data from being violated. The Biba model attempts to quantify this level of trust and maintain the flow of information, inhibiting its modification by lower trusted subjects/objects.

2.4 BLP-Biba Lipner Like

Lipner proposed an integrity model that combined aspects of the BLP and Biba model to fit a more real-world commercial software development environment. Lipner created security levels and categories similar to BLP to prevent various software development lifecycle subjects from accessing different systems. Lipner then added to the security classifications with integrity classifications for system programs and integrity categories to differentiate development and production environments.

In our BLP model we established various security classifications (Table 5) and work unit categories (Table 6) so we need to establish security categories to define the area of impact in the given category (Table 8).

We can now incorporate integrity categories to allow distinction between on aircraft systems and the flight record keeping system (Table 9) and integrity classifications to determine the trust of the data combing from that source (Table 10 highest to lowest level).

This gives us an overall security and integrity clearance levels for subject categories (Table 11) and objects (Table 12).

Ideally, these classifications would block EC from accessing FRS by putting them in a lower classification stopping our rights leakage. However, due to the requirement of MC to allow EC to read a flight record, our system would not operate otherwise.

2.5 Clark Wilson

The Clark Wilson is an integrity verification model that enforces a principle of *separation of duty* in that a subject/object that verifies a data state transition is not the same one that caused it. Data with valid integrity is defined to be in a *consistant state*, and processes can only transform data through valid *transactions* that preserve this consistency. Data that is constrained by the separation of duty and transaction controls are said to be *constrained data items (CDIs)*. In contrast, data not constrained by these are called *unconstrained data items (UDIs)*.

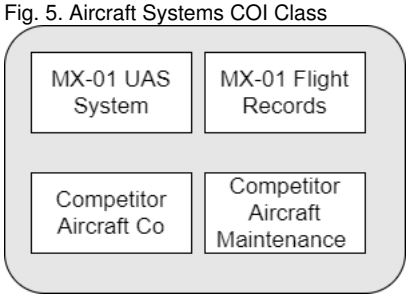
After defining out CDIs, we define two procedures, an *integrity verification procedure (IVP)* that tests the system is in a valid state, and *transformation procedures (TPs)* that will change the state of the system using transactions. To ensure valid TPs occur on valid CDIs, the Clark Wilson model has various *certification rules (CR)* that ensures the TPs and IVPs that operate on CDIs keep a valid state and *enforcement rules ER* that prevent TPs from operating on CDIs that have not been certified.

First, for our scenario, we'll define the CDIs and UDIs.

TABLE 12
Security and Integrity Levels for Objects

Object	Security	Integrity
ANC	({TS,{OD})	(ISP,{IOA})
RO	({TS,{OD})	(ISP,{IOA})
FED	({S,{AD,OD,RTL})	(IO,{IOA,IFRS})
RTD	({S,{OD})	(IO,{IOA,IFRS})
FRS	({S,{AD,OD,RTL})	(IM,{IFRS})

Fig. 5. Aircraft Systems COI Class



In this, we'll describe the contained data items as data related to the flight record and the unconstrained being the autonomous operations. While on their own, the ANC and RO are vital, and access should be confidential. The integrity check, in this case, is specific to the flight record data.

- $CDIs = \{FED, RTD, FRS, FR\}$
- $UDIs = \{ANC, RO\}$

Next, we need to establish the *integrity constraints* over the flight record data.

- The FRs uploaded to the FRS must be equivalent to the data on the UAS as it is transferred over to the FRS.
- The FRS can only be written via new FRs or deleted, but FRs themselves cannot change.

Next, we will define the IVP commands that run on the system to ensure these constraints have been adhered to. After an executed upload, the first IVP check rereads the FR from the UAS and compares it to the FR in the FRS. The next IVP ensures that the FRS is not modified and will run before and after an upload is executed.

To execute IVP1 a different user would have to re-download the flight record from the UAS and verify against the data in the FRS system. Then to test IVP2 likely a system program would record the state of the FRS before a transaction and compare it to the state of the system after the transaction to verify none of the integrity constraints in IVP2 are invalid. Some examples of incorrect data could include the flight record dates such as the start time being after the end time, the aircraft location data indicating speeds or movements beyond the capabilities of the UAS, corrupted data in the flight record causing the inability to read basic information such as the aircraft identifier, attached components, data header parameters, etc.

- IVP1

```
read uac_flight_record uac_fr

if uac_fr != FRS[fr]
  return invalid
else
  return valid
```

- IVP2

```
read FRS initial_FRS

\\ Execute any TP

read FRS after_FRS

foreach FR in after_FRS
  if FR in initial_FRS
    if FR.FlightData !=
        initial_FRS[FR.ID].FlightData
      return invalid

return valid
```

Now we can define the TPs that our model can execute to preform the FRS functionality. These commands themselves enforce the CR rules.

- TP1: execute create_flight_record(p)
- TP2: execute upload_flight_record(p)
- TP3: execute delete_flight_record_from_system(p)
- TP4: execute update_flight_record_system(p)

2.6 Chinese Wall

The Chinese Wall model is a hybrid approach that enforces both confidentiality and integrity. It establishes a separation or "wall" between subjects and objects that would have a conflict of interest between them. For example, a subject who has read the engine data of our MQ-25 UAS would have a conflict of interest if they were to work on another UAS from another company. Our policy enforces that subjects who have a conflict of interest do not have access to specific data defined in our *conflict of interest classes*. In our scenario, we describe our *objects* related to our UAS data from Table 3. These objects will be contained in our *company datasets (CD)*, containing our various UAS data. Above that, our *conflict of interest class (COI)* defines what CDs our subjects will have access to and how they will be protected if later transferred to another company/organization.

First we'll define our *company datasets (CD)* and our *objects* in our datasets that contain information related to our UAS. These datasets will fall under an **Aircraft Systems** COI class (Figure 5).

$$CD = (\{FlightRecords, \{FED, RTD, FRS\}\}, \{UASOperations, \{ANC, RO\}\})$$

In this scenario, any subject having read a from a competitor dataset could not access a dataset in our COI class. If we wanted to allow access to a specific object in our dataset (i.e., a **Flight Record**), we could implement a sanitization method on the **Flight Records**. We could establish a new **Sanitized Flight Record** and only allow the EC access to this sanitized object. However, a new process would need to create the sanitized record off the original FR after upload and be maintained on a new sanitized FRS, which does not allow the MC and EC to do their jobs thoroughly. Also, this would only allow the EC not to conflict when later moving to a competitor company.

3 Non-Interference (NI)

The Non-Interference (NI) model focuses on information flow to ensure that objects and subjects at each security level do not "interfere" with those at different levels. It achieves this by modeling inputs and outputs at different sensitivity

levels, i.e., High and Low. It intends to ensure that those modifying data at a low level of security clearance can only see the machine's state as if only low-level interactions were occurring regardless of what the high-level entities are doing. For example, in our system, the Pilot Commander executing flight instructions on the ANC should not change their system state or be visible by a Maintenance Contractor reading old flight records. These rules are all in an attempt to ensure high-level activities are not visible by low levels, with the theory being low levels would be able to infer information about high-level activities by these changes.

To detect interference, we will utilize a scenario similar to our BLP situation where the following commands are executed from the MC and EC that produces a rights leakage. Essentially to detect interference, we need to determine if actions that modify the FRS (which is at a *Secret* clearance level) produce outputs that can be seen by the EC (at a lower *Confidential* level).

```
create_flight_record(Maintenance Crew
    (MC));
grant_flight_record_access(Maintenance
    Crew, External Contractor (EC),
    Flight Record (FR));
upload_flight_record(External Contractor
    (EC), Flight Record (FR));
make_owner(MC, EC, FR);
update_flight_record_system(EC, FR);
```

1) **MC** executes *create flight record* creating flight record object **FR**.
   - There is no interference here as MC is the same level as the new FR.
2) **MC** executes *upload flight record* granting the object **FRS** access to the flight record object **FR**.
   - There is no interference here as MC uses a higher-level procedure to write to the FRS but not read from it.
   - However, some potential interference issues are if any errors occur when uploading to the FRS if the FRS returns any faults or exceptions. The MC can read this output that would constitute interference as higher-level outputs would be visible by the MC.
   - The *upload_flight_record()* process would need to sanitize the output of any errors or even successful debugging information.
3) **MC** executes *grant flight record access* granting **EC** access flight record object **FR**.
   - There is no interference here as MC is the same level as the new FR.
4) **MC** executes *make owner* making **EC** have owner rights over flight record object **FR**.
   - There is no interference here as MC and EC are at the same level as the new FR.
5) **EC** executes *update flight record system* allowing **EC** write to the flight record system object **FRS**.
   - While this is a leak, this doesn't immediately infer an interference as the EC only has *write* access which is a rights leak but not necessarily an interference.
   - However, this enables the EC to attempt various changes to the system and observe the successes or failures. Therefore unless the system is accessed through only procedures that do not respond with any success or failure messages, an interface issue will occur here.
   - To solve this interference, the system would have to limit *write* access to the FRS to be only allowed through system procedures, and said procedures do not respond with any return code information to any subject at *Confidential* or lower rights.

4 Non-Inference (NF)

Non-Inference is similar to Non-Interference in that it wants to limit the ability of low-level entities to detect the actions of high-level ones. However, NF allows high-level actions as long as the same output could be achieved by low-level ones, thus obfuscating the guarantee to the low-level entity whether the result came from a high or low-level action.

For example, in our scenario, we want the actions of S and TS clearance levels to be hidden from the MC and EC subjects at the C security level. In the NI example, we showed how the standard flight record creation BLP procedure that produced a leak earlier was still allowed as long as the system does not produce any response to the actions taken by the EC when executing on the FRS.

Our NF scenario will follow the same procedure as before, except we can add some examples of how NF could allow some output to the C level if other C level entities could achieve the same.

ToDo:

1) **MC** executes *create flight record* creating flight record object **FR**.
   - There is no interference here as MC is the same level as the new FR.
2) **MC** executes *upload flight record* granting the object **FRS** access to the flight record object **FR**.
   - There is no interference here as MC uses a higher-level procedure to write to the FRS but not read from it.
   - However, some potential interference issues are if any errors occur when uploading to the FRS if the FRS returns any faults or exceptions. The MC can read this output that would constitute interference as higher-level outputs would be visible by the MC.
   - The *upload_flight_record()* process would need to sanitize the output of any errors or even successful debugging information.
3) **MC** executes *grant flight record access* granting **EC** access flight record object **FR**.
   - There is no interference here as MC is the same level as the new FR.

4) **MC** executes *make owner* making **EC** have owner rights over flight record object **FR**.
    - There is no interference here as MC and EC are at the same level as the new FR.
5) **EC** executes *update flight record system* allowing **EC** write to the flight record system object **FRS**.
    - While this is a leak, this doesn't immediately infer an interference as the EC only has *write* access which is a rights leak but not necessarily an interference.
    - However, this enables the EC to attempt various changes to the system and observe the successes or failures. Therefore unless the system is accessed through only procedures that do not respond with any success or failure messages, an interface issue will occur here.
    - To solve this interference, the system would have to limit *write* access to the FRS to be only allowed through system procedures, and said procedures do not respond with any return code information to any subject at *Confidential* or lower rights.

REFERENCES

[1] A. Erwin, and J. Gibson, Navy, Boeing Make Aviation History with MQ-25 Becoming the First Unmanned Aircraft to Refuel Another Aircraft, Accessed on: Sept. 1, 2021. [Online]. Available: https://www.boeing.com/defense/mq25/

Matthew Whitesides Master's Student at Missouri University of Science and Technology.