Asset-Oriented Access Control through Object-Oriented Principles

Anonymous Author(s)

ABSTRACT

The decision of whether to grant access to an asset has traditionally been considered a matter for the system in which the asset resides. Centralized approaches to access control can, however, be problematic; in particular, for assets that are not always bound to one administrated system. Examples of such assets are beginning to be observed in Internet of Things (IoT) settings, and there are instructive examples present in the context of file sharing, where data assets leave the system for which they were originally specified. As an alternative to centralized approaches, we propose a framework for empowering assets with the ability to carry out their own access control. This approach is quite distinct from, for example, DRM and laissez-faire. To facilitate the shift in perspective from systems to assets, we adapt principles from object-orientation. We use these principles to make sense of an indicative IoT problem present in the literature. After considering what lessons can be learned from assets in the file sharing domain, we develop a further set of principles that ensure that assets can make decisions according to their context. Collectively, all of the principles come together to form what we call Asset-Oriented Access Control. In the concluding part of the paper, we consider some well-known policy models, give evidence of how they can fit into the overall framework, and consider how they differ form our approach. We also consider future directions for the work, such as providing a logical foundation and a prototype implementation.

ACM Reference format:
Anonymous Author(s). 2019. Asset-Oriented Access Control through Object-Oriented Principles. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 12 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Access control (AC) has typically been understood at the level of systems, often explicitly involving a notion of a central arbiter that decides the sort of access that is allowed for each asset. Centralized reference monitors, however, are ill-placed for dealing with assets that are not bound to one administrated system. For example, in the context of file sharing, where data assets need to regularly cross system boundaries, there is no assurance that when an asset leaves the system for which it was originally specified, it will continue to be used in ways consistent with that specification.

Digital rights management (DRM) technologies have attempted to address this issue, but they often rely on access to a central provider to receive a license (something not always possible) and a narrow range of controls that are not interoperable with other providers (something undesirable) [4, 32]. An alternative view is that rather than attempting to make assets secure by locking them down to systems and restricting their use, assets should have their intended security and usability in as many contexts as possible.

It has also been amply demonstrated that the lack of flexibility in current AC schemes drives non-compliance [6, 49]. For example, when users are forced to sacrifice sharing rights to introduce files into rigidly specified AC systems, they are naturally led to other sharing methods (e.g., personal, unencrypted USB sticks) resulting in weaker security. Similarly, readily-available access to public WiFi hotspots means that is easy to work around strict policies insecurely. Pallas [43] has analytically demonstrated the need for a more flexible approach to AC in modern distributed organizations.

The laissez-faire policy model given in [34] is a non-centralized AC approach for file sharing. Broadly, it suggests that control over files should reflect current controls for emails. In particular, owners should not need to sacrifice rights in order to share a file (requirement of ownership), owners should be able to pass on rights to whomever they please (requirement of delegation), and owners should be able to see all changes to a file (transparency). While we recognize that diminishing rights can lead to insecure behaviour, these properties provide only minimal protection for files.

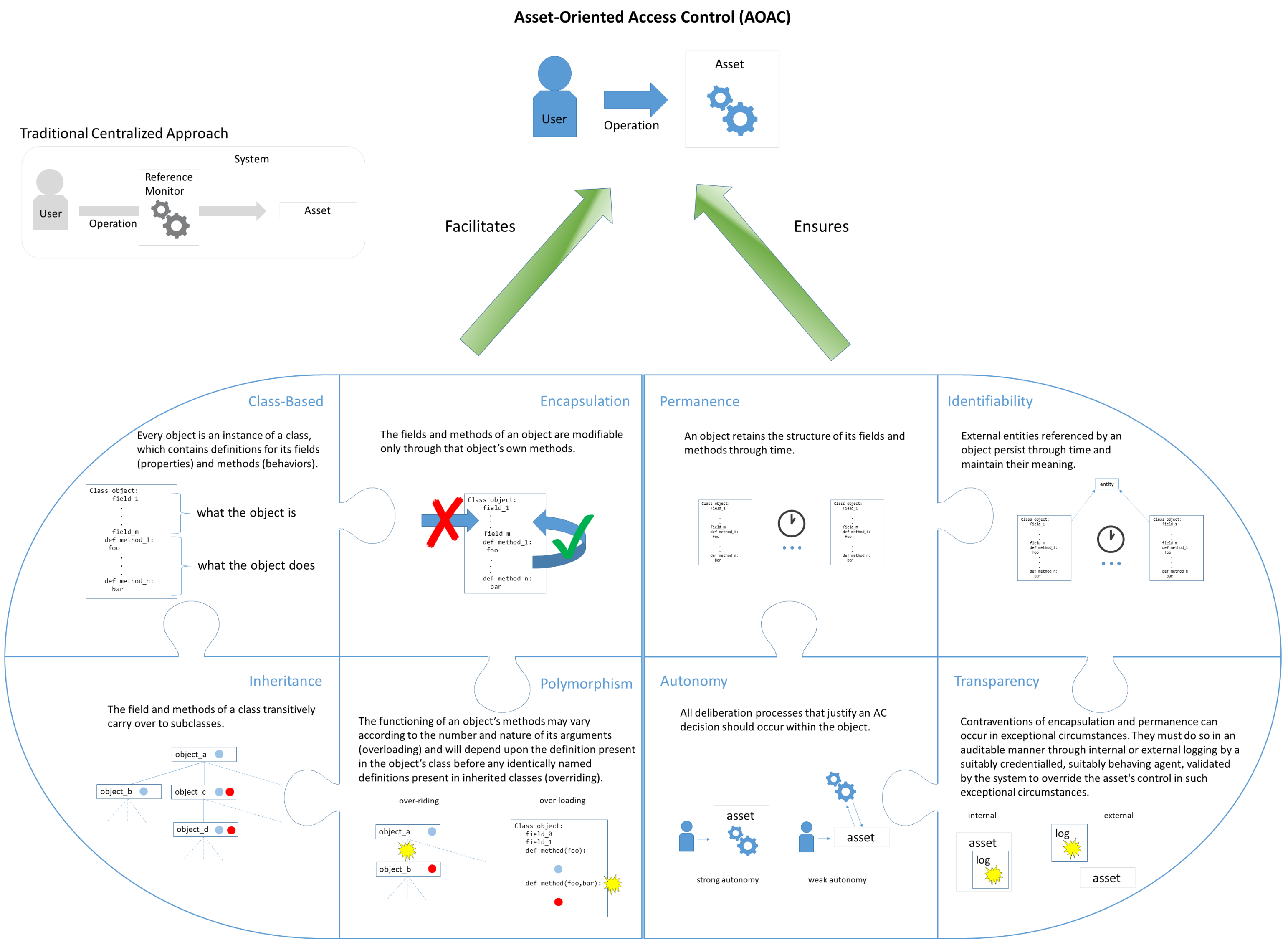
More recently, related issues have begun to emerge in the context of Internet of Things (IoT) devices, either because they are free from any one centrally administrated system, or because the AC schemes they fall under are not adequately dynamic to allow for fast and easy policy modification [20].

These examples show the need for a rethinking of AC from the ground up, one that allows assets to be made secure dynamically according to their context.

As an alternative approach, we propose inverting the control vector of AC by understanding AC not in terms of what can be done to assets — but what assets can do themselves. Instead of passive entities governed by centralized authorities, we seek to empower assets to be capable of performing their own AC decisions based on their context. Put another way, we want to empower each asset to have the necessary structure for storing and processing its access based on its properties, its relationship to other assets, and its broader environment. In a sense, then, assets should contain their own reference monitors. Such an approach will allow dynamically secure access contextually, in line with that given by Calo et al., [13].

To make this possible, we provide a framework with which to specify the design of assets. This framework is in part based on principles of object-oriented design that are used explicitly in the security domain. These principles provide assets with the features they need to perform their own AC.   In order to mitigate potential

Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



Class-Based

Every object is an instance of a class, which contains definitions for its fields (properties) and methods (behaviors).

Encapsulation

The fields and methods of an object are modifiable only through that object's own methods.

```
Class object:
    field_1
      .
      .
      .
    field_m
  def method_1:
    foo
      .
      .
      .
  def method_n:
    bar
```

Permanence

An object retains the structure of its fields and methods through time.

Identifiability

External entities referenced by an object persist through time and maintain their meaning.

Inheritance

The field and methods of a class transitively carry over to subclasses.

Polymorphism

The functioning of an object's methods may vary according to the number and nature of its arguments (overloading) and will depend upon the definition present in the object's class before any identically named definitions present in inherited classes (overriding).

```
Class object:
  field_0
  field_1
  def method(foo):

  def method(foo,bar):
```

Autonomy

All deliberation processes that justify an AC decision should occur within the object.

Transparency

Contraventions of encapsulation and permanence can occur in exceptional circumstances. They must do so in an auditable manner through internal or external logging by a suitably credentialled, suitably behaving agent, validated by the system to override the asset's control in such exceptional circumstances.

Figure 1: Principles for enabling the movement from traditional to asset-oriented access control

problems with this approach, we provide details for a further set of principles that ensure that assets behave appropriately through time. Collectively, the principles work together to form what we call Asset-Oriented AC. We refer to this framework as 'asset-oriented' as opposed to 'object-oriented' AC as the latter hascome to mean AC for object-oriented systems, rather than AC that is object-oriented (see, e.g., [24, 50]). An overview of the approach is given in Figure 1.

There has been a great deal of work on logical models of access control régimes, perhaps the most influential being [2] and its developments. Logical models provide frameworks in which policies can be specified and within which the compliance of system behaviour with policy can be reasoned about, perhaps even verified or falsified. We suggest that logical models can be very helpful in the setting of AOAC and, Section 5, we discuss the kinds of logical approaches that we think will be necessary, These include agency, location and resource, announcement, and epistemic and doxastic ideas.

The structure of the paper is as follows: in the next section, we give definitions for the principles of asset-oriented access control and use them to understand a motivating IoT problem posed in [13]. In doing so, we provide a set of basic features that assets require in order to make their own access decisions. In Section 3, we broaden our scope to explore data assets, using the principles as a conceptual analysis tool to understand the related problems users face with files. This leads us to generate a further set of principles needed for ensuring assets behave appropriately in any context through time. We use this set to refine the original IoT scenario, and give an example of robots that are operating independently and being updated, showing the breadth of design space that asset-oriented access control applies to. We give a basic simulation implementation (in Python). In Section 4, we show that currently existing AC policy models can be expressed using the framework. We also give an indication of how faithfully they capture the spirit of our approach. In conclusion, we consider future directions for the work in Section 5 as well as some limitations.

2 THE PRINCIPLES OF ASSET-ORIENTED ACCESS CONTROL (AOAC)

From an external perspective, AC involves three key elements: operations, assets, and principals [48]. More specifically, AC can be understood as the management of operations on assets performed by principals. Internally, the details of how to carry out this management has been explored in different ways. In role-based access control (RBAC), for example, the properties of principals (specifically, their roles) provide the decision-making basis [25]. In attribute-based AC (ABAC), the decisions depend on properties of the asset, principal, and environment [31]. In Usage Control (UCON), obligations are also considered, which add the fulfillment of certain actions to the access calculation [47]. All of these approaches have, at their core, ideas present from the 1970s [37].

At a similar time, ideas surrounding object-orientation were beginning to develop, initially for use in programming languages with the goal of making code more robust, extensible, reusable, portable, and easier to use [41]. These ideas have since emerged as a more general set of principles for use in the analysis and design of software [9]. Bringing these two fields together, we can lay foundations for AC that is suited for assets that carry out their own AC, as called for in [13]. We define the principles of AOAC as facilitating the management of principal-asset operations that is class-based and exhibits inheritance, encapsulation, and polymorphism. We unpack this definition while analysing the following scenario developed in [13]:

It is already common for newly manufactured cars to have in-built navigation systems capable of storing, and providing directions to, home addresses and previous locations. Leaving access to these systems open is a single point of failure security risk: an intruder can break into the car, gather the additional sensitive information, and use it to commit further crime.[1] *As cars get smarter, the consequences of this open access vulnerability get more severe; keyless ignition systems, remote control garage fobs, and other IoT devices present in the car increase the amount of damage criminals can do.*

Given that a car is, by its nature, independent of other systems (it may even, from time to time, lose GPS signal), it cannot defer its AC decisions to some external centralized authority; it must instead make the relevant decisions itself. Furthermore, these decisions will be contextual, covering a wide range of use cases. Most obviously, the car will need to allow for different sorts of access to passengers as opposed to the owner. It may need to allow for temporary access, for example in the case of car hires, and it may even need to allow for exceptional circumstances, for example, when emergency services need access because of an accident.

We will now see how the principles of AOAC can help to understand this problem. For each principle, we will provide:

- a general definition explaining what the principle means for arbitrary assets;
- a concrete, specific example drawing out the relevant features of the principle;
- code written in the object-oriented language Python (2.7) that, when executed according to its operational semantics, exhibits behaviour that is interpreted as implementing the given concept;
- comments to clarify the principle and its intended meaning.

2.1 Class-Based

2.1.1 Definition

Every object is an instance of a class, which contains definitions for its fields (properties) and methods (behaviours).

2.1.2 Example

To determine its own AC, the smart vehicle needs to keep track of whether it is currently open and modify this property based on the presence of the owner. As a class-based object, it can do this by having a field that stores its access state together with the methods necessary for changing this state under the right conditions.

2.1.3 Code Example

```
class SmartCar:
   def __init__(self, owner):
      self.access = False
      self.owner = owner

   def unlock(self, user):
      if self.authentication(user):
         self.access = True

   def authentication(self, user):
      if user == self.owner:
         return True
```

[1]Real-world cases of such events are already being reported [11].

The class *SmartCar* has a constructor method (*def __init__*) that takes one argument, the owner. It is used to create an instance of a smart car (an object) that has an owner attribute and an access state (by default the car is closed). The class has two other methods: *authentication*, which checks whether the user is the owner, and *unlock*, which opens the car if the authentication succeeds.

2.1.4 Remarks

By being class-based, assets (as objects) carry with them all of their security relevant properties and behaviours. This is a crucial first step towards assets being able "to generate and manage their own access control policies dynamically on their own." [13, p.39]. For simplicity, we have abstracted away the complexities of the authentication process by assuming that the car has some way of identifying its owner. In practice, this could take place through facial recognition technology, fingerprint scanners, password protection, or some other authentication procedure.

2.2 Inheritance

2.2.1 Definition

The fields and methods of a class transitively carry over to subclasses.

2.2.2 Example

The car's ignition and navigation systems are both entities that require at least the same level of protection as the car itself — otherwise, the car is at risk from the open access vulnerability. The principle of inheritance ensures this by giving the ignition and navigation systems all the security properties and behaviours present in the overall car.

2.2.3 Code Example

```
class IgnitionSystem(SmartCar):
   pass

class NavigationSystem(SmartCar):
   def __init__(self, owner, home_address):
      SmartCar.__init__(self, owner)
      self.home = home_address
```

The class *ignitionSystem* references the *SmartCar* class as an argument, inheriting its security fields and methods. It has no additional fields and methods (hence *pass*). The *NavigationSystem* also inherits from *SmartCar*, but also has the *home* address attribute.

2.2.4 Remarks

Inheritance allows the car-ignition relationship to work analogously to the way it does in traditional car security: the owner is needed to enter the car and also to start it. Traditionally this has involved two separate uses of a key; here, this involves two separate authentications. As a result, gaining illegal entry into the car does not grant immediate access to the ignition system. Access to the car navigation system works in much the same way, though this system contains additional information, in particular the home address location.

2.3 Encapsulation

2.3.1 Definition

The fields of an object are modifiable only through that object's own methods.

2.3.2 Example

The remote control garage fob is a standalone device that may either be present in the car or simply carried by the owner. In the latter case, the fob should assume access, but in the former, it should base its decision on the car's access. The principle of encapsulation ensures that only the fob's request sets its new access state and, likewise, only the car's send method gets the car's access state.

2.3.3 Code Example

```
class SmartCar:
   ...
   def send_access(self):
      return self.access

class GarageFob:
   def __init__(self):
      self.access = True

   def request(self, nearest_object=None):
      if nearest_object != None:
         self.access = nearest_object.send_access()
```

The *GarageFob* class has a method *request* that looks for a nearby object. If an instance of the *SmartCar* class is nearby, it will give its *access* state to the remote control garage fob through the *send_access* method.

2.3.4 Remarks

This principle emphasizes the fact that central to object-orientation is the idea of messaging between objects. In the context of security broadly, encapsulation is vital to ensuring that external entities cannot modify the inner states of an object. For smart assets in particular, encapsulation meets Research Challenge 4 of allowing assets "in the environment to be aware of other devices in the environment" [13, p.46] and doing so in a standard way that avoids obvious security exposures. In more complex situations, the remote control could defer access to other devices and base the decision on further conditions. For example, it could request access via Bluetooth through an app on the owner's phone that authorizes access with a fingerprint.

2.4 Polymorphism

2.4.1 Definition

The functioning of an object's method may vary according to the number and nature of its arguments (overloading) and will depend upon the definition present in the object's class before any identically named definitions present in inherited classes (overriding).

2.4.2 Example

Access to a car is typically not limited to the car's owner; passengers should also be able to enter the car, as long as the owner is with them. By default, however, passengers should not have access to the ignition system. The principle of polymorphism allows for both of these situations: firstly, access to the car can depend upon the number and nature of the users, and secondly, access to the ignition system can behave differently from its car superclass.

2.4.3 Code Example

```
class SmartCar:
  ...
  def unlock(self, *users):
    for user in users:
      if self.authentication(user):
        self.access = True

class IgnitionSystem(SmartCar):
  def __init__(self, owner):
    SmartCar.__init__(self, owner)

  def unlock(self, user):
    if self.authentication(user):
      self.access = True
```

The *SmartCar* class now has a new *unlock* method that can take arbitrarily many arguments, which will change its behaviour (overloading). The *Ignition* class inherits *unlock* but modifies its behaviour by redefining the method (overriding).

2.4.4 Remarks

The flexibility afforded by polymorphism is vital for dynamic AC. Through overloading, the same method may change its functioning given almost identical stimuli. A more detailed example could be that the ignition system does not grant access if the owner is present but intoxicated. Similarly, if the owner is under duress, perhaps in the presence of an actor judged to be malicious, the car denies access.

Through overriding, inheritance can be limited, allowing many properties and behaviours to be inherited, but not all. By requiring as few changes as possible to be made, this could reduce the policy burden on a human manually changing policy or a computationally-limited machine learning it.

This section has explained how converting assets to objects using the object-oriented paradigm can empower them to make their own AC decisions. In summary: being class-based ensures that assets carry the security relevant properties and behaviours they need in order to perform their own AC within themselves; inheritance allows distinguished assets to transfer their properties and behaviours; encapsulation ensures that objects can behave in a secure manner no matter where they are, through secure messaging; and polymorphism gives assets a level of dynamism in behaviour necessary to be context-aware. Consequently, assets (as objects) are modular, capable of interacting together to perform complex AC.

The general nature of the AOAC principles means that they are also applicable to the other scenarios described in [13]. For example, in the Partner Information Access Scenario, the need for remote access could be met by having each service be class-based, determining access individually without assuming constant connectivity. The need for histories may potentially be met through inheritance, basing AC decisions on previous superservices. The importance of context of operations despite identical end-points suggests that overloading may be a useful principle here. Elastic Data Centers might exhibit encapsulation to "only allow requests from the tier ahead of it" [13, p.41] and exhibit polymorphism broadly to allow for the dynamic updates necessary for each tier. The Client Side IoT Security Scenario also stresses the importance of emergency situations where "security norms may need to be violated" [13, p.42]. This is something that we will look at in more detail later on.

3 FURTHER PRINCIPLES OF AOAC, DATA ASSETS, AND MORE SCENARIOS

In the previous section, we saw that the principles of AOAC facilitate assets to be secure objects that interact with themselves, each other, and their environment in order to make their own AC decisions. In particular, we saw how this could work for device assets in an IoT setting. As the principles apply to assets generally, it is possible to apply them also to data assets, in particular to files. Unlike devices, we do not normally think of files as independent entities free from an underlying system, but we do observe them regularly crossing system boundaries, which means that the same issue of enforcing policies in a decentralized manner arises.

In this section, we will apply the principles of AOAC to the file sharing domain. This will lead us to consider some familiar ways in which security lapses occur, forcing us to consider a further set of principles to mitigate these problems. In the final part of this section, we will apply the principles back to the original IoT scenario and consider a further scenario, with the aim of avoiding analogous problems for non-data assets.

Email was not originally designed for secure file sharing [42]. Despite this, it has since become "by far the most reported method for sharing files" [51, p.223]. That nearly everyone has access to an email account, means that it is often the go-to platform for users wishing to initiate a share for the first time, and there is some evidence that email's universality makes it the common fallback platform when other file systems fail users [19]. While email clients are not particularly object-oriented, emails do exhibit some degree of structure by bundling file attachments together with recipient information and messages. As we shall see, this allows for the expression of intended AC policy, albeit problematically.

Unix-like OSs exhibit some object-oriented design by treating every entity as a file (even attached devices) that carries around its own AC in the form of permission bits. Sub-files can be made to exhibit security inheritance, but encapsulation is not strictly possible because security bits rely on the underlying OS to be interpreted correctly. While intra-system file shares work well — assuming that the required sharers are all present on the system — extra-system shares (i.e., shares from a Unix to another system) can be problematic.

Lastly, a Wiki — in particular, Wikipedia — is a technology and website[40] in which pages act as shared files for collaboration. Files, as understood, exhibit a high degree of structure, storing an article together with its history and discussion pages. Fine-grained AC is possible, and history pages indicate when changes are made and by who, using timestamps and IP or username signatures.

We refer to the set of further principles drawn out by consideration of these cases as permanence, identifiability, autonomy, and transparency. We follow the structure of the previous section, again providing (1) a general definition, (2) a concrete application, (3) an object-oriented implementation, and (4) further comments.

3.1 Permanence

3.1.1 Definition

An object retains the structure of its fields and methods through time.

3.1.2 Example

Alice sends an email to Bob with an attached file. Within the body of the email, she includes the words "FOR INTERNAL USE ONLY" to encourage Bob to only share the file within the company. Bob downloads the attachment to his phone. Three weeks pass, Bob forgets Alice's request, and he shares the file with a non-employee. Permanence denies objects the ability to separate their fields and methods, ensuring that Alice's policy request remains associated with the file.

3.1.3 Code Example

```
class Email:
   def __init__(self, message, attachment):
      self.data = attachment
      self.metadata = message

#  def download(self, device): # improper
#     return self.data         # decoupling

   def download(self, device):
      return self.data, self.metadata
```

The *download* method should return both the attachment (*data*) and the message (*metadata*), rather than just the attachment (as shown in the commented out code).

3.1.4 Remarks

While the principle of encapsulation does put limits on how an asset's fields are modified, it does not stop an asset from removing its own policies or the internal information that help it to make AC decisions. A better approach would be akin to what we see with Wikipedia, where an articles page maintains a strong connection to its history page. While edits can be made to the article, each of these edits are recorded, as well as previous states of the article. For data assets understood as objects, most of the asset's fields will refer to metadata, and "embedded metadata has the advantage of keeping the metadata with the file it describes, ensuring the file is understandable in new contexts." [46, p.14]. Permanence forces all an asset's methods to continue the strong coupling of data and metadata, but relies on encapsulation to ensure that these are the only methods that can modify these fields.

3.2 Identifiability

3.2.1 Definition

External entities referenced by an object persist through time and retain their meaning.

3.2.2 Example

Every file on a Unix-like system carries with it its operation permissions (read, write, and execute) for three types of user (owner, group, and everyone). When a file is transferred to another system, however, there is no guarantee that the second system will interpret the permissions correctly, because the meaning of 'owner', 'group', and 'everyone' will likely not be consistent between the two systems. Consistency will not occur even if both systems are Unix-like under the likely circumstance that the two systems have different users. Identifiability requires that such user references always point to the same entities irrespective of where the asset is and for as long as necessary for the intended usage of the asset.

3.2.3 Code Example

```
   def UnixFile:
      def __init__(self, data):
         self.data = data

#     def read(self, user_alice): # non-permanent
#        self.data = new_data     # reference

      def read(self, permanent_reference):
         self.data = new_data
```

The *read* method requires a $permanent\_reference$ argument as opposed to the specific *user_alice*.

2.2.4 Remarks

There are a number of different sorts of entity an asset may refer to. The example above focuses on principal entities and shows that Unix-like file sharing is only reliable within one system: "while content placed in a shared folder can be restricted to allow access only by certain users, such restrictions can only be done for hosts in a single administrative domain." [51, p. 222] While throwaway email accounts are common, so too are permanent ones, owned and maintained by the same user for decades [19]. Email addresses would therefore be more suitable than references to users, if indefinite sharing of a file is acceptable.

More broadly, the identifiability requirement suggests that contextual references should be preferred over definite descriptions. Spatial and temporal references fit well into this category. In 2.3.2, the remote control garage fob deferred its AC to nearby devices. Other proximity-based approaches have been proposed [35], but this approach continues to rely on a central specification of trusted agents and their locations. Similarly, restricting access to times of the day or for a set period (transient access) is in keeping with the requirement of identifiability. The properties of assets may also be considered identifiable features; for example in the form of tagging in photo sharing [30, 36, 54].

3.3 Autonomy

3.3.1 Definition

All deliberation processes that justify an AC decision should occur within the object.

3.3.2 Example

A file in any of the systems mentioned is operated upon only through the system within which it is situated. As a result, access to the file is fundamentally constrained by access to the system: access to an email assumes access to the client; access to a Unix file assumes access to the OS; and access to a Wikipedia webpage assumes access to Wikipedia. Autonomy is the requirement that only the file itself is the barrier to access.

3.3.3 Code Example

```
#class Entity: # e.g., underlying system
#  def __init__(self):
#     self.access = False
#     self.file = "data"

#  def access(self):          # access relies
#     if self.access == True: # on external
#        return self.file     # entity

class File:
   def __init__(self, data):
      self.data = data
```

```
        def access(self):
            return self.data
```

Access to the file is dependent only on the *File* method *access*, not also on the commented-out *access* method of the *Entity* class.

3.3.4 **Remarks**
AOAC empowers files with the mechanisms they need to make their own AC decisions, but it does not stop them being placed within systems that further constrain their access. For certain files (e.g., system files) this might be appropriate, but it is superfluous and unhelpful for files that are intended to leave the system. More broadly, as assets cannot rely on the presence of other entities, be they systems or other assets, autonomy is required. They can, like the example in 2.3.2, base their AC on other assets, but they should be capable of working without them, in principle. In practice, a balanced relationship between day-to-day autonomy and irregular updates to AC appears most sensible. An asset that is behaving insecurely, for example, will need modifying, perhaps through DRM-style technologies.

3.4 Transparency

3.4.1 **Definition**
Contraventions of encapsulation and permanence can occur in exceptional circumstances. They must do so in an auditable manner through internal or external logging by a suitably credentialled, suitably behaving agent, validated by the system to override the asset's control in such exceptional circumstances.

3.4.2 **Example**
Anyone can edit almost[2] any Wikipedia article and, in keeping with the permanence requirement, all previous states of an article are stored in its history page. As a result, it is possible for illegal, highly sensitive information to be stored indefinitely on a page, such as the leaking of military secrets. Transparency allows the deletion of such sensitive information, even from an article's history page, as long as the act of deletion and justification is recorded by a suitably credentialled user (administrator).

3.4.3 **Code Example**

```
class WikiFile:
    def __init__(self):
        self.data = ""
        self.history_meta = []
        self.talk_meta = []

    def edit(self, new_data, cause=None):
        if cause == None:
            self.history_meta.append(self)
            self.data = new_data
        else:
            self.data = new_data
            self.history_meta.append(cause)
```

The *WikiFile* class has a method *edit* that will append an old page (*data*) to its history page *history_meta* unless a *cause* (i.e., justification) is given, in which case it will not append the old page and simply record the justification.

3.4.4 **Remarks**
This is a requirement that Wikipedia already meets, but it is not widespread across file sharing platforms. The ability of system administrators on Unix to arbitrarily raise privileges (i.e., in non-exceptional circumstances), for example, means that it is prone to misuse. While email is often used for sharing when other platforms fail, the sharing is similarly non-transparent. 'Break-glass'[3] mechanisms have been deemed necessary by [43], both for individuals as well as organizations. The work of [12] demonstrates how traditional AC approaches can be extended to accommodate such thinking.

It is also important to stress the exceptional nature of transparency. The case of Windows UAC, where users bombarded with permission requests are more likely to accept them blindly, shows that break-glass mechanisms cannot be the norm [21]. A further consideration is that when a file needs deleting, there is little point in recording this fact to a log stored in the file. In such cases, an external log is necessary. When we provide a full definition of AOAC shortly, we provide further comments about how transparency relates to the other principles.

This section has outlined a further set of principles that assets need in order to mitigate issues that arise when viewing assets as objects. Each of them build on the original principles of AOAC to ensure security. Permanence requires that assets retain the features that allow them to make their AC decisions through time; identifiability requires that their relationship to external entities is reliable in any context; autonomy requires that assets are self-governing; and transparency allows for break-glass mechanisms in exceptional circumstances through responsible means.

> In summary, an AOAC specification consists in the following:
> - having all of the the object-oriented principles of being *class-based*, exhibiting *inheritance*, *encapsulation*, and *polymorphism*;
> - having all of the additional principles of *permanence*, *identifiblility*, and *autonomy*.
>
> Additionally, the *transparency* principle can be used — by suitably credentialled, suitably behaving agents — to *override* permanence and encapsulation, which principles are able to hide control of access from agents requiring access in *exceptional circumstances*.

We have seen an example of override of permanence in our discussion of Wikipedia and we shall give an example of override of encapsulation in Section 3.6. In future work, we will explore the possible need for override of other principles.

These principles are actually very difficult to meet for data assets. This is in part because the system-centred view has been so dominant in the field of AC, and in part because of the nature[4] of data. Having said this, there have been attempts to secure data assets in ways

[2]Exceptions include articles relating to current events being editable only by logged in users, and the IP banning of users who repeatedly damage pages [52].

[3]As described in [44, p.198], "the term 'break-glass' is derived from fire alarms that require breaking a glass cover for triggering an alarm."

[4]An analogy can be made to physical documents. The prototypical way of trying to ensure a document is protected from mis-shares is to use classification labelling; for example, by having "CONFIDENTIAL" written in big letters at the top of the document. Permanence is hard because it is easy to remove such labelling; identifiability is hard because it cannot be assumed that the labelling will be understood correctly by everyone who encounters the document; autonomy is hard because the labelling does not affect the use of the document (e.g., a wax seal would be better at ensuring autonomy); transparency is hard because it is easy to share the document without generating an audit trail.

to meet the principles. Sticky Policies [14] is one approach. They enforce permanence by encrypting files together with their machine-readable policies for later decryption with a key issued from a trusted authority. This is not an ideal solution for two reasons. Firstly, the trusted authority is a third party that, like DRM, may not always be available. Secondly, permanence is only enforced up to the point of decryption, meaning that once the file is accessible to the user it remains accessible from that point onwards. Consequently, if a user wants to share on the file without any controls, they are free to do so. Data Usage Control [38, 39] enforces permanence by first establishing that the file is received by a system that has comparable controls to the sending system. This is clearly only possible if the receiving system is known beforehand, and as such is only appropriate for a small proportion of circumstances.

3.5 Returning to the Car Scenario

Devices, however, have a significant advantage over data assets in that they are already understood as independent entities with the potential to control their own interactions.

Returning to our initial car IoT scenario, designs must take the permanence requirement seriously so that car functions do not suppress or remove access conditions. This suggests that car users — even the owner — should be inhibited from removing AC features arbitrarily. For example, if the owner wishes to delegate access to another driver, the car itself should facilitate this securely, instead of the owner being forced to temporarily remove access conditions, resulting in diminished security.

Identifiability becomes relevant to the design of the authentication and authorization mechanisms by ensuring that owner attributes and environmental conditions are as stable and contextual as possible. If, for example, facial recognition hardware and software are used, it will presumably only be as accurate as similar technology present at border controls, so profile updates will need to be as regular as those for passport photos.

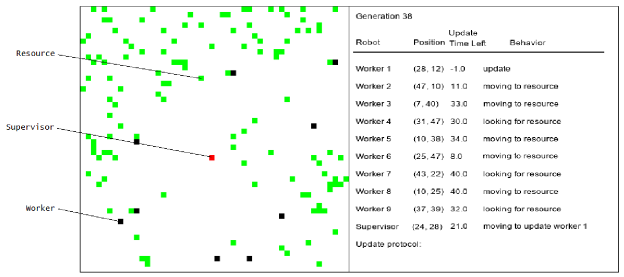
Taking autonomy seriously means that such mechanisms should not be designed as separate objects that could potentially become disconnected. Instead, face and voice recognition should be designed as properties of the car as opposed to communicable assets; in terms of access, these are the eyes and ears of the car, and must be strongly coupled to the brain that makes the AC decisions. Pragmatic autonomy can occur in the case of hire cars, where, if there is time to sign documentation for a temporary driver, there is also time to update the AC aspects of the car for the new profile.

Transparency may possibly enter into the design process to allow for the exceptional case of an emergency. If such situations are allowed, then they should be justified, either through request to a next of kin or identification as an emergency worker (examples of suitably credentialled and behaved agents); as much information as possible should be recorded, both in the car and at some remote server.

3.6 Robot Example

Iot devices have more independence than data assets, but less than robots. According to the IEEE [33]:

> *A robot is an autonomous machine capable of sensing its environment, carrying out computations to make decisions, and performing actions in the real world.*



Generation 38

Robot	Position	Update Time Left	Behavior
Worker 1	(28, 12)	-1.0	update
Worker 2	(47, 10)	11.0	moving to resource
Worker 3	(7, 40)	33.0	moving to resource
Worker 4	(31, 47)	30.0	looking for resource
Worker 5	(10, 38)	34.0	moving to resource
Worker 6	(25, 47)	8.0	moving to resource
Worker 7	(43, 22)	40.0	looking for resource
Worker 8	(10, 25)	40.0	moving to resource
Worker 9	(37, 39)	32.0	looking for resource
Supervisor	(24, 28)	21.0	moving to update worker 1

Update protocol:

Figure 2: Basic elements of the robot example simulator showing diagnostic box.

A robot needs perceptive capabilities to sense the relevant features around itself and the ability to change its internal states according to its perceptions. It also needs motive powers to enact observable changes, to move and to signal. All of these idea can come to bear on a robot carrying out AC decisions.

More concretely, consider a robotics manufacturer wanting to design multiple robots for use in an arbitrary environment where the majority of the robots, the workers, perform a primary task (collecting resources, say), and one robot, the supervisor, performs checks on the workers to make sure they are functioning correctly. These checks, or updates, need to be carried out at regular intervals and securely. Occasionally, the supervisor is required to go back to some central station to receive secure updates of their own.

We provide a 2D simulator[5] for this scenario that uses the principles of AOAC we have described. Figure 2 is an image taken from one run of the simulator with annotations showing the basic entities involved. The basic idea is that workers navigate around the map collecting resources until their internal clock tells them they need an update, at which point they stop and request one. The supervisor finds the worker in need of the update and, when they meet, they exchange messages in order to update securely. Once the update is complete, the edited worker goes back to collecting resources and the supervisor looks for another worker in need of an update. More rarely, the supervisor returns to the centre of the map to receive an update from the system.

We will now describe how each of the AOAC principles enable the assets (worker and supervisor robots) to make their own AC decisions.

Class-Based

First, we define a general class *Robot* that contains the basic security features for every robot.

```
class Robot:
    def __init__(self):
        self.access_state = False
        self.internal_clock_start = time.time()
        self.time_up = random.randint(2,30)
        self.log = []
        self.color = (0,0,0)

    def get_access(self, asker):
        if asker.color == (255,0,0):
            if random.randint(1,100) != 1:
```

[5]For the full code and example usage, including videos, see https://github.com/cora711/2D-Simulator-for-Robot-Example. Graphics developed using the Python Image Library (PIL), through Pillow [15].

```
                self.access_state = True
                return True

        def update(self, msg):
            self.access_state = msg[0]
            self.internal_clock_start = msg[1]
            self.time_up = msg[2]
```

In particular,

- the *access_state* field tells the robot whether its internal states can be modified
- the *internal_clock_start* and (random) *time_up* fields tell the robot when it needs an update
- the methods *get_access* and *update* allows the robot to change its access according to what it 'sees' (the color of the robot) and rewrite its fields respectively.

Being class-based, each robot can thus decide whether to allow an update or not (via its access field) and when (via its time fields), depending on whether it receives the right input (via its methods).

Inheritance

Second, we define two subclasses *Worker* and *Supervisor* which inherit the basic security features as described above. On top of this, they can contain their own fields and methods peculiar to their intended operations.

```
class Worker(Robot):
    def __init__(self):
        Robot.__init__(self)

class Supervisor(Robot):
    def __init__(self):
        Robot.__init__(self)
        self.time_up = 30
        self.update_message = [
            False,
            time.time(),
            100]

    def request_access(self, recipient):
        if recipient.color == (0,0,0):
            if recipient.get_access(self):
                recipient.update(self.update_message)
```

In particular,

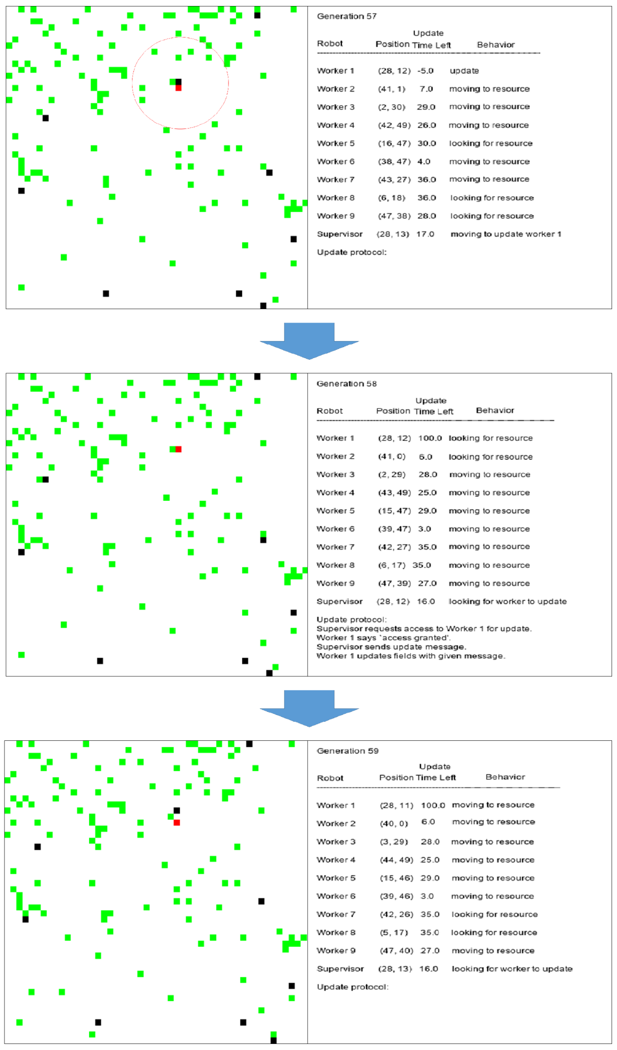
- the workers have no additional security fields or methods
- the supervisors have a different update time setting (*time_up*) and an *update_message* field that is used in updating workers
- the supervisor also has a *request_access* method used to authenticate the worker and update it if it allows.

Inheritance allows the security features common to all robots to be held by the *Robot* class and specific features to be held by the subclasses.

Encapsulation

Third, we make sure that updates are in principle only possible by the robot that is being updated. More specifically, each update works according to the following protocol:

- supervisor authenticates worker
- supervisor requests access to worker
- worker receives request and authenticates supervisor
- if authentication is successful, worker sends accept signal to supervisor
- supervisor receives accept signal and sends update message
- worker receives update message and updates its own setting based on the message.



```
Generation 57

                         Update
Robot          Position Time Left      Behavior
-------------------------------------------------------
Worker 1       (28, 12)  -5.0     update
Worker 2       (41, 1)    7.0     moving to resource
Worker 3       (2, 30)   29.0     moving to resource
Worker 4       (42, 49)  26.0     moving to resource
Worker 5       (16, 47)  30.0     looking for resource
Worker 6       (38, 47)  4.0      moving to resource
Worker 7       (43, 27)  36.0     moving to resource
Worker 8       (6, 18)   36.0     looking for resource
Worker 9       (47, 38)  28.0     looking for resource
Supervisor     (28, 13)  17.0     moving to update worker 1
Update protocol:
```

```
Generation 58

                         Update
Robot          Position Time Left      Behavior
-------------------------------------------------------
Worker 1       (28, 12)  100.0  looking for resource
Worker 2       (41, 0)    6.0   looking for resource
Worker 3       (2, 29)   28.0   moving to resource
Worker 4       (43, 49)  25.0   moving to resource
Worker 5       (15, 47)  29.0   moving to resource
Worker 6       (39, 47)  3.0    moving to resource
Worker 7       (42, 27)  35.0   moving to resource
Worker 8       (6, 17)  35.0    moving to resource
Worker 9       (47, 39)  27.0   moving to resource
Supervisor     (28, 12)  16.0   looking for worker to update
Update protocol:
Supervisor requests access to Worker 1 for update.
Worker 1 says `access granted'.
Supervisor sends update message.
Worker 1 updates fields with given message.
```

```
Generation 59

                         Update
Robot          Position Time Left      Behavior
-------------------------------------------------------
Worker 1       (28, 11)  100.0  moving to resource
Worker 2       (40, 0)    6.0   moving to resource
Worker 3       (3, 29)   28.0   moving to resource
Worker 4       (44, 49)  25.0   moving to resource
Worker 5       (15, 46)  29.0   moving to resource
Worker 6       (39, 46)  3.0    moving to resource
Worker 7       (42, 26)  35.0   looking for resource
Worker 8       (5, 17)   35.0   looking for resource
Worker 9       (47, 40)  27.0   moving to resource
Supervisor     (28, 13)  16.0   looking for worker to update
Update protocol:
```

Figure 3: Three consecutive states of a simulation showing the supervisor updating a worker. Note the new update time for the worker.

The results of worker update are shown in Figure 3. In the first state, the stationary worker is waiting for an update; in the second, the supervisor and worker follow the protocol described above; in the third, the worker goes back to finding and collecting resources, and the supervisor looks for more workers to update.

More broadly, encapsulation means that every security method of a robot is protected in a way that restricts direct manipulation.

Polymorphism

Fourth, we let the robots have to have one update method to cover two situations.

```
class Supervisor(Robot):
        ...
    def update(self):
        self.update_message = system_update_message
        self.internal_clock_start = time.time()
        self.time_up = 100

system_update_message = [
        False,
    time.time(),
    300]
```

In particular,

- the *update* method in *Supervisor* overrides the one defined in after a call to a supervisor, allowing the supervisor to return to the central station and update according to the system's specification rather than wait to be updated like the worker
- worker update is unaffected.

Polymorphism gives the designer flexibility to define such a method twice, changing its behaviour according to the context.

Through these principles, we make it possible for the robots to determine their own access based on their context. To mitigate potential problems that may still arise, however we also implement the following further principles.

Permanence
While the robots need methods to modify their access, they should not have methods that remove it, or otherwise render it useless. For example, the robots should be precluded from

- removing their access states
- removing their ability to authenticate other robots.

Permanence forces the robots to continue to base their access decisions on their internal features through time.

Identifiability
The means by which the robots authenticate other robots should reliable in any context, as should the location of the supervisors central station. More specifically,:

- recognition of a robot is done locally (i.e., by the relevant robot) through visual means
- the central station has fixed coordinates.

More broadly, any of the conditions that a robot is basing its access decision on will need to be intelligible whatever the context.

Autonomy
The barriers to access should be limited to the robots and not also external entities. In particular:

- the ability to authenticate robots exists in each robot visually, and not through some system command (e.g., the $entity.\_\_class\_\_$ system function);
- supervisors only need to ask workers for permission and not also a central authority.

Autonomy requires that there are not further barriers to entry than those present in the robots themselves. Note that this does not preclude updates to the supervisor's AC.

Transparency
The design should also allow for exceptional circumstances. Suppose a supervisor attempts to update a worker, but the worker (incorrectly) denies access. (Note: this is possible, although unlikely, given how we defined *get_access* in the class *Robot* (see specifically the line $if random.randint(1, 100)! = 1$)). Owing to encapsulation, in principle there is nothing more the supervisor can do; the worker will be stuck operating abnormally. In such extreme circumstances, the suitably credentialled and well-behaved supervisor has no choice but to:

- contravene encapsulation to force the worker to update, and
- record the incident in its log and the worker's log.

This can be allowed for with the following code:

```
class Supervisor(Robot):
        ...
    def force_update(self, recipient):
        recipient.access_state = True
        recipient.internal_clock_start = time.time()
        self.log.append(
        (time.time(),
        recipient,
        action = "Forced␣Update",
        reason= "Denied␣Access"))
        recipient.log.append(
        (time.time(),
        self,
        action= "Forced␣Update",
        reason= "Denied␣Access"))

    def request_access(self, recipient):
        if recipient.color = (0,0,0):
            if recipient.get_access(self):
                recipient.update(self.update_message)
            else:
                self.force_update(recipient)
```

Transparency allows design to cover such exceptional circumstances in an auditable manner.

In terms of independence, data assets lie at one end of the spectrum, robots at the other, with devices somewhere in between. AOAC provides a framework for all three. The set of properties ensure that the principles of AOAC allow assets to continue to make decisions based on their context through time.

4 AOAC AND POLICY MODELS

We have considered a theory of how AC can work from the asset's perspective, but in practice, of central importance is the analysis of specific policies. We want to know whether a given asset is following the prescribed behaviour intended by a policy. This is where AC policy models come in: they allow us to reason — formally, in some cases — about whether a system correctly implements a given policy. From this perspective, schemes like RBAC and ABAC are general frameworks through which policies can be expressed; in themselves, they express no official dogma, but they do frame the discussion in such a way as to facilitate some. It is in this sense that we say that AOAC has facilitated assets to make their decisions in any context. In the previous sections, we generated a further set of properties. Unlike the principles of AOAC, these do dictate some amount of policy. Consequently, it is possible to state classical models using AOAC and show how they disagree with the properties of the previous section.

For example, while the Bell-LaPadula model [8] can be expressed through the principles of AOAC, it does not meet the set of further properties. In particular, as classification levels do not have universal meaning, they are not identifiable. This means that implementing Bell-LaPadula for assets that are free to leave their system boundaries will be problematic. For example, if we apply Bell-LaPadula to the Robot Example of the previous section by insisting that every robot and resource in the system has a classification label, and interaction is determined by the "read-down rule", the robot will be unable to interact with resources outside of the system. This problem is analogous to the one experienced when labelled documents leave some secure system and are used insecurely. We can use AOAC to recognize the problem as well as its potential solution, namely that the robot would itself need to determine the classification level of the resource, assigning it a label based on the context.

Similar remarks can be made for the Clark-Wilson [16] and Laissez-Faire [34] models. As supplementary material, we include code that expresses the three policy models.[6]

5 FURTHER WORK

We have identified, exemplified, and discussed the core concepts of AOAC, starting from object-oriented principles. In this section, we consider two main directions for further work.

First, in order to test the practical viability of AOAC, a protoype should be implemented. This could take shape in several forms:

- Simulation of an IoT device, and its security protocols, that interacts with its environment according to the principles and properties described above;
- A modelling tool that can be directly applied to the design and analysis of devices in distributed systems;
- A prototype operating system that implements an AOAC file system and/or email environment (cf. [34]).

Such tools could be a basis for usability studies in the sense of [53].

Second, in order to reason about the properties and behaviours of systems of access control that embody these principles, it would be helpful to have formal logical models of AOAC. Such models might support the characterization of specific access control policies and the verification and falsification of the compliance of the behaviour of systems with policies.

There is a substantial literature on logical models of access control. We consider here the most pertinent contributions for our purposes. The work of Abadi, Burroughs, Needham, & Plotkin [2] introduces the *says* modality, facilitating reasoning about delegation in the distributed systems setting. Subsequent work of Abadi [1] gives further refinements to the logic to prevent undesirable (from the point of view of security) logical consequences in the original system.

The work of Collinson and Pym [18] employs a resource–process algebra, in which systems are modelled as processes which evolve relative to available resources, together with its associated modal logic. In this context, the 'says' modality can be seen to characterize a process combinator that represents the one principal acting in the role of another. This set-up can be enriched with a notion of location that enables the modelling of architecture of distributed systems [17].

More recent modal logic based work of Genovese & Garg [27] and Genovese [26] (in collaboration with many others [10, 28, 29]) considers new access control modalities in the distributed systems setting (corresponding to *control*, *permission* and *ratification*), and gives a unified approach to the semantics and proof theory of authorization logics via modal correspondence theory and labelled deduction.

While this work provides a useful starting point for logical modelling of AOAC, some key features of AOAC are not readily handled. First, interaction between an asset and its local environment in order to support local about about access to the asset by entities in the environment. This requires a representation of the local environment in some part of the broader global context, in a manner similar to that which is captured by the frame rules of separation logics [45, 55], which enable reasoning about the effects of program execution by restricting attention to that part of the global memory state that is manipulated by the program. Second, local reasoning by assets about their interaction with their environments requires representations of knowledge and beliefs in the sense of epistemic and doxastic logics [23]. Third, assets must maintain models of their state and environment as they and their environments evolve. This requires the logic to incorporate a notion of 'model update', such as in dynamic epistemic logics [22]. Further, capturing the notion of message passing requires more sophisticated notions of update from the literature, for example, private announcements [5]. Last, the representation of assets is based on object-oriented principles, so the associated logics must handle at least some aspects of object-oriented design. Approaches can be seen in Beckert & Platzer [7], Abadi & Leino [3], among others.

The existing work on logic for access control fails to meet our properties for the following reasons: first, we require a suitable framework for local reasoning; second, the modalities we require must reason about local properties, unlike the 'says' modality of [1, 2, 27] which makes global assertions; third, the logic must incorporate a notion of 'model update'; and, finally, reasoning about some aspects of object-oriented structure is necessary. We require a logic which appropriately incorporates all of these features. Such a logic would be used to reason about the properties and dynamics of systems that implement AOAC — for example, questions of correctness and emergent behaviour through techniques such as model checking.

This program of logical work is substantial and beyond the scope of this paper.

6 CONCLUSION

In this paper, we have set out a framework for designing assets that can make their own AC decisions. The framework (AOAC) is in part based on principles adapted from object-orientation. These principles allow assets (as objects) to store and perform AC (by being class-based), pass over security features (through inheritance), message indirectly (through encapsulation), and have dynamic behaviours (through polymorphism). We have applied these principles to make sense of a problem in the literature that is indicative of the type that IoT devices are beginning to face. More specifically, we have addressed the need for assets to make decisions based on their context dynamically, a problem recognized by the community (cf. [13]).

We have also shown how the principles can be applied to data assets, and have pointed out specific ways the approach must be refined to mitigate security issues. This has led us to formulate a further set of properties that ensure that assets retain their class-based structure (permanence), remain capable of interacting with their environment (identifiability), are not limited by external access conditions (autonomy), and behave exceptionally responsibly (transparency). We have provided definitions for each of the principles and properties, examples that show in what circumstances they apply, and implemented source code in an object-oriented language showing their behaviour. Collectively, the principles and properties form what we have termed Asset-Oriented Access Control (AOAC).

By describing traditional access control policy models — such as Bell–LaPadula and Clark–Wilson, as well laissez-faire — within our framework, we have begun to demonstrat the flexibility of the AOAC

[6]See also the github repository: https://github.com/cora711/aoac-policies.

approach. This work also indicates some ways in which these models can be made to accommodate asset-oriented ideas.

Finally, we have described some potential directions for future work, specifically some approaches prototype implementations and logical modelling as formal basis for AOAC.

REFERENCES

[1] M. Abadi. 2008. Variations in access control. LNCS 5076: 96–109.

[2] M. Abadi, M. Burrows, B. Lampson and G. Plotkin. 1993. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst.* 15, 4, 706–734.

[3] M. Abadi and K. Rustan and M. Leino. 2003. A logic of objected oriented programs. In *Verification: Theory and Practice*, LNCS 2772, 11–41.

[4] Adobe Systems Incorporated. 2015. *Adobe Content Server.* Retrieved February 7, 2019 from http://www.adobe.com/solutions/ebook/content-server.html.

[5] A. Baltag, L. Moss and S. Solecki. 1998. The logic of public announcements, common knowledge and private suspicions. *Proc. TARK '98*, Morgan Kaufmann, 43–56.

[6] A. Beautement, M. Angela Sasse and M. Wonham. 2009. The compliance budget: managing security behaviour in organisations. In *Proceedings of the 2008 New Security Paradigms Workshop (NSPW'08)*. ACM Press, New York, NY, 47–58.

[7] B. Beckert and A. Platzer. 2006. Dynamic logic with non-rigid functions: a basis for object-oriented program verification. *Proc. IJCAR '06*, LNAI 4130:266–280.

[8] D. Bell and L. LaPadula. 1973. *Secure computer systems: mathematical foundations.* MITRE Technical Report 2547, Vol. I.

[9] S. Bennett, S. McRobb and R. Farmer. 1999. *Object-oriented systems analysis and design using UML* (2nd Ed.). McGraw-Hill Publishing Co., New York.

[10] G. Boella, D. Gabbay, V. Genovese, L. van der Torre. 2009. Fibred security language. *Studia Logica* 92, 395–436.

[11] British Broadcasting Corporation. 2019. Hundreds of popular cars 'at risk of keyless theft'. Retrieved February 7, 2019 from https://www.bbc.co.uk/news/business-47023003.

[12] A. Brucker and H. Petritsch. 2009. Extending access control models with break-glass. *SACMAT '09: Proceedings of the 14th ACM on Symposium on Access Control Models and Technologies*, ACM, New York, NY, USA, 197–206.

[13] S. Calo, D. Verma, S. Chakraborty, E. Bertino, E. Lupu and G. Cirincione. 2018. Self-Generation of Access Control Policies. *Proc. SACMAT '18.* ACM, New York, 39–47.

[14] D. Chadwick and S. Lievens. 2008. Enforcing "sticky" security policies throughout a distributed application. In *MidSec '08: Proceedings of the 2008 workshop on Middleware security*, ACM, New York, NY, USA, 1–6.

[15] A. Clark and Contributors. 2019. Pillow (PIL fork). Retrieved February 7, 2019 from https://pillow.readthedocs.io/en/stable/.

[16] D. Clark and D.. Wilson. 1987. A comparison of commercial and military computer security policies. *Proc. 1987 IEEE Symp. on Security and Privacy*, 184–184.

[17] M. Collinson, B. Monahan and D. Pym. 2012. *A Discipline of Mathematical Systems Modelling.* College Publications.

[18] M. Collinson and D. Pym. 2010. Algebra and logic for access control. *Formal Aspects of Computing* 22, 2, 83–104, 22(3-4), 483-484, 2010.

[19] B. Dalal, L. Nelson, D.Smetters, N. Good, A. Elliot. 2008. Ad-hoc Guesting: when exceptions *are* the rule. In *UPSEC'08 Proceedings of the 1st Conference on Usability Psychology and Security*, USENIX Association, Berkeley, CA, USA, Article 9.

[20] S. Demetriou, N. Zhang, Y. Lee, X. Wang, C. Gunter, X. Zhou, M. Grace. 2017. HanGuard: SDN-driven protection of smart home WiFi devices from malicious mobile apps. *WiSec '17*, ACM, New York, 122–133.

[21] J. DeVaan. 2009. Update on UAC. Retrieved February 7, 2019 from https://blogs.msdn.microsoft.com/e7/2009/02/05/update-on-uac/.

[22] H. van Ditmarsch, J. Halpern, W. van der Hoek and B. Kooi (eds.). 2015. *Handbook of Epistemic Logic.* College Publications.

[23] H. van Ditmarsch, W. van der Hoek and B. Kooi. 2008. *Dynamic Epistemic Logic.* Synthese Library 337, Springer Netherlands.

[24] W. Eßmayr, G.Pernul and A. Min Tjoa. 1998. Access controls by object-oriented concepts. In *Database Security XI: Status and Prospects*, IFIP Conference Procs. 113, Chapman & Hall, 325–340.

[25] D. Ferraiolo, J. Cugini, D. Richard Kuhn. 1995. Role-based access control (RBAC): Features and motivations. In *Proceedings of 11th Annual Computer Security Applications Conference*, IEEE Computer Society Press, 241–48.

[26] V. Genovese. 2012. *Modalities for Access Control: Logics, Proof-Theory and Applications.* PhD Thesis. Universitié du Luxembourg.

[27] V. Genovese and D. Garg. 2011. New modalities for access control logics: permission, control and ratification. *Proc. STM 2011*, LNCS 7170: 56–71.

[28] V. Genovese, D. Garg and D. Rispoli. 2012. Labeled sequent calculi for access control logics: countermodels, saturation and abduction. In *CSF '12: IEEE 25th Computer Security Foundations Symposium*, IEEE, 139–153.

[29] V. Genovese, L. Giordano, V. Gliozzi and G. Pozzato. 2011. A conditional constructive logic for access control and its sequent calculus. *Proc. TABLEAUX 2011*, LNCS 6793: 164–179.

[30] M. Hart, C. Castille, R. Johnson and A. Stent. 2009. Usable privacy controls for blogs. In *2009 International Conference on Computational Science and Engineering*, IEEE, 401–408.

[31] V. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller and K. Scarfone. 2014. *Guide to Attribute Based Access Control (ABAC) Definition and Considerations.* NIST Special Publication 800–162.

[32] G. Heileman and P. Jamkhedkar. 2005. DRM interoperability analysis from the perspective of a layered framework. In *DRM '05: Proc. of the 5th ACM Workshop on Digital Rights Management*, ACM, New York, 17–26.

[33] IEEE. 2018. What is a robot?. Retrieved February 7, 2019 from https://robots.ieee.org/learn/.

[34] M.Johnson, S. Bellovin, R. Reeder and S. Schechter. 2009. Laissez-faire file sharing: Access control designed for individuals at the endpoints. *Proc. NSPW'09*, ACM, New York, 1–10.

[35] M. Kirkpatrick, M. Damiani and E. Bertino. 2011. Prox-RBAC: a proximity-based spatially aware RBAC. In *SIGSPATIAL GIS '11: Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ACM, New York, NY, USA, 339-348.

[36] P. Klemperer, Y. Liang, M. Mazurek, M. Sleeper, B. Ur, L. Bauer, L. Cranor, N. Gupta and M. Reiter. 2012. Tag, you can see it!: Using tags for access control in photo sharing. *Proc. of the SIGCHI*, ACM, New York, 377–386.

[37] B. Lampson. 1971. Protection. In *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, 437-443. Reprinted in *Operating Systems Review* 8, 1, January 1974, 18–24.

[38] F. Kelbert. 2013. Data usage control for the cloud. In *CCGRID 2013: Proceedings of the 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, IEEE Computer Society, 156–159.

[39] F. Kelbert and A. Pretschner. 2013. Data usage control enforcement in distributed systems. In *CODASPY '13: Proceedings of the third ACM conference on Data and Application Security and Privacy*, ACM, New York, NY, USA, 71–82.

[40] B. Leuf and W. Cunningham. 2001. *The WIKI WAY. Quick Collaboration on the Web. Addison Wesley.*

[41] B. Meyer. 1997. *Object-Oriented Software construction* (2nd Ed.). Prentice Hall.

[42] C. Partridge. 2008. The technical development of internet email. *IEEE Annals of the History of Computing* 30, 2, 3–29.

[43] F. Pallas. 2009. *Information Security Inside Organizations: A Positive Model and Some Normative Arguments Based on New Institutional Economics.* PhD Thesis, TU Berlin.

[44] H. Petritsch. 2014. *Break-Glass: Handling Exceptional Situations in Access Control.* Springer.

[45] J. Reynolds. 2002. Separation Logic: A logic for shared mutable data structures. *Proc. LICS '02*, IEEE Computer Society, Washington, DC, 55–74.

[46] J. Riley. 2017. Understanding metadata. *National Information Standards Org.*. Retrieved http://www.niso.org/publications/press/UnderstandingMetadata.pdf.

[47] R. Sandhu and J. Park. 2003. Usage control: A vision for next generation access control. *Proc. MMM-ACNS 2003*, LNCS 2776: 17–31.

[48] R. Sandhu and P. Samarati. 1994. Access Control: Principles and Practice. *IEEE Communications Magazine* 32, 9, 40–48.

[49] D. Smetters and N. Good. 2009. How Users Use Access Control. In *SOUPS '09: Symposium on Usable Privacy and Security*, ACM, New York, Article 1.

[50] T. Tachikawa, H. Higaki, and M. Takizawa. 1997. Purpose-oriented access control in object-based systems. In *Austral. Conf. Information Sec. and Privacy.*

[51] S. Voida, W. Keith Edwards, M. Newman, R. Grinter and N. Ducheneaut. 2006. Share and share alike: exploring the user interface affordances of file sharing. *Proc. CHI '06* ACM, New York, 221–230.

[52] Wikipedia contributers. 2019. *Wikipedia Banning Policy.* Retrieved on February 7, 2019 from https://en.wikipedia.org/wiki/Wikipedia:Banning_policy.

[53] T. Whalen, D. Smetters, and E. Churchill. 2006. User Experiences with Sharing and Access Control. (CHI'06).

[54] C. Au Yeung, L. Kagal, N. Gibbins and N. Shadbolt. 2009. Providing access control to online photo albums based on tags and linked data. In *Social Semantic Web: Where Web 2.0 Meets Web 3.0. AAAI Spring Symposium.*, 9–14.

[55] H. Yang and P. O'Hearn. 2002. A semantic basis for local reasoning. *Proc. FoSSaCS 2002*, LNCS 2303: 402–416.