Object Capability Patterns: Policies and Specifications

Shu-Peng Loh
Imperial College London
shu.loh16@imperial.ac.uk

Sophia Drossopoulou
Imperial College London
s.drossopoulou@imperial.ac.uk

Abstract

We propose a set of higher-order predicate logic to formally specify object-to-object interactions which can then be used to describe reference dynamics in an object-oriented computational model. Using these predicates, we attempt to formally specify the policies of well-established Object-Capability (OCap) patterns within the OCap literature which we have implemented in the capability-safe language Pony. We also offer some preliminary insights on how such specifications can be used in the context of a non-Ocap model by describing the security properties of a pattern built on the Ethereum smart contract programming language Solidity, which we argue implements a form of stack-based access control.

1 Introduction

Recent widespread adoption of distributed ledger technology (blockchain) has created multiple decentralized, distributed computational platforms where millions of dollars are transacted over codified constructs called smart contracts[1]. The power of distributed modern computing hence lies in facilitating cooperation between multiple agents, but it comes with risk as an agent is vulnerable to *unexpected* outcomes[2] from participating in these smart contracts. This might generally arise from two issues:

- oversight or misconception of the outcomes of executing a piece of *known* code
- failure to defend against malicious execution of *unknown* code components

These two issue are often closely intertwined in any system of execution that has both trusted and untrusted code components (the second issue is often a result of the first).
In recent years the Object-Capability (OCap) model has received increasing attention as a compelling approach to building robust, distributed systems that promote what Miller[3] calls *cooperation without vulnerability*. The OCap model attemps to address these two issues by alleviating security as a separate concern from the mind of the programmer, by leveraging the object-oriented programming paradigm and imposing certain prohibitions.

[1]For example, as of 10 Aug 2017, Ethereum is a US$28 billion blockchain platform with an in-built Turing-complete programming language that can be used to create and deploy such contracts.

[2]Representing in general any outcome arising from a piece of code execution that has deviated from an agent's original intention or objective independent from code.

2 OCap Model

The OCap model uses the reference graph of the objects as the access graph, and strictly requires objects to interact with each other only by sending messages on object references[4].

2.1 From Capability to Object-Capability

2.2 OCap Languages

- Joe-E (inspired by Java)
- Emily (inspired by OCaml)
- Caja (inspired by JavaScript)
- E
- Pony

2.2.1 Language Restrictions

2.3 OCap Patterns

An OCap pattern is a concrete representation of the OCap model and comprises a set of objects

connected to each other by capabilities. Objects interact with each other by sending messages on capabilities. An OCap pattern may be visualised as a directed graph—nodes represent objects, and each edge from an object o to another o' represents o holding a capability that allows it to directly access o'.

3 Formal specifications

3.1 Definitions

We borrow liberally the definitions of runtime state, module and arising configurations from the appendix of [2].

Runtime state: σ consists of a stack frame φ, and a heap χ. A stack frame is a mapping from receiver (this) to its address, and from the local variables (VarId) and parameters (ParId) to their values. Values are integers, the booleans true or false, addresses, or null. The heap maps addresses to objects. Objects are tuples consisting of the class of the object, and a mapping from field identifiers onto values.

$\sigma \in$ state = frame $\times$ heap
$\varphi \in$ frame = StackId $\rightarrow$ val
$\chi \in$ heap = addr $\rightarrow$ object
v $\in$ val = {`null`, `true`, `false`} $\cup$ *addr* $\cup \mathbb{N}$
StackId = {*this*} $\cup$ VarId $\cup$ ParId
object = ClassId $\times$(FieldId $\rightarrow$ val)

Module:

M $\in$ Module = ClassId $\cup$ SpecId
$\rightarrow$
(ClassDescr $\cup$ Specification)

Reach and Execution:

Arising Configurations

Domination:

3.1.1 MayAccess Definitions

We define in total four flavours of MayAccess predicates that describe the relation between two entities in a system (of arity 2 that represent the identifiers of these entities). These four flavours represent a combination of space (distance) and time (state):

- distance: directly (*Dir*) or indirectly (*Ind*)
- state: now (*Now*) or eventually (*Eve*)

and are broad enough to describe both non-OCap and OCap models:

$$\begin{aligned}&M,\sigma \vDash \text{MayAccess}^{Dir,Now}(x,e) \iff \\ &\quad \exists f.\ \lfloor x.f \rfloor_\sigma = \lfloor e \rfloor_\sigma \vee \\ &\quad (\sigma(\text{this}) = \lfloor x \rfloor_\sigma \wedge \exists y.\ \sigma(y) = \lfloor e \rfloor_\sigma)\end{aligned}$$

$$\begin{aligned}&M,\sigma \vDash \text{MayAccess}^{Dir,Eve}(x,e) \iff \\ &\quad \exists \sigma' \in \text{Arising}(M,\sigma). \\ &\quad M,\sigma' \vDash \text{MayAccess}^{Dir,Now}(x,e)\end{aligned}$$

$$\begin{aligned}&M,\sigma \vDash \text{MayAccess}^{Ind,Now}(x,e) \iff \\ &\quad \exists \bar{f}.\ \lfloor x.\bar{f} \rfloor_\sigma = \lfloor e \rfloor_\sigma \vee \\ &\quad (\sigma(\text{this}) = \lfloor x \rfloor_\sigma \wedge \exists y.\ \sigma(y.\bar{f}) = \lfloor e \rfloor_\sigma)\end{aligned}$$

$$\begin{aligned}&M,\sigma \vDash \text{MayAccess}^{Ind,Eve}(x,e) \iff \\ &\quad \exists \sigma' \in \text{Arising}(M,\sigma). \\ &\quad M,\sigma' \vDash \text{MayAccess}^{Ind,Now}(x,e)\end{aligned}$$

A note on f and $\bar{f}$: While f can considered as a field, it can also represent a method that returns a val. Similarly $\bar{f}$ can be considered a series of fields, or methods that return a val, or a combination of both.

We surmarise the relationships between the four flavours of MayAccess in *Table 1*:

Table 1: *MayAccess*,*(x,e) Relations*

	Now		**Eventually**
Direct	$\text{MayAccess}^{Dir,Now}$	$\Longrightarrow$ $\not\Longleftarrow$	$\text{MayAccess}^{Dir,Eve}$
	$\Downarrow \Uparrow$		$\Downarrow \Uparrow$
Indirect	$\text{MayAccess}^{Ind,Now}$	$\Longrightarrow$ $\not\Longleftarrow$	$\text{MayAccess}^{Ind,Eve}$

Let us assume that both x and e are well-defined. Note that without imposing any further assumptions (such as those from the OCap model), we have defined *both* $\text{MayAccess}^{Dir,Now}$(x,e) and

MayAccess^{Ind,Now}(x,e) to mean forms of very weak access—that a directed path exists from x to e, but we do *not* imply that such a path is traverseable by x (it might or might not be traverseable). Indeed, these definitions by themselves represent mere *possibilities* of interaction (or possible authorities), they do *not* represent that interaction (or authority) would succeed. The difference between MayAccess^{Dir,Now}(x,e) and MayAccess^{Ind,Now}(x,e) is only the computational distance between x and e on the reference graph, where the latter involves possibly intermediate entities (or objects in an object-oriented model).

What do the definitions mean then for non-OCap and OCap models in an object-oriented world? Again, let us assume both o and o′ are well-defined, valid object references. In non-OCap models, the possible presence of a global ambient authority means the predicates above say *nothing* about whether any interaction between an object o and o′ would succeed. This is true even if o possesses directly the reference of o′, where MayAccess^{Dir,Now}(o,o′) holds. For all we know, we could easily have in a non-OCap language a feature to completely restrict access to o′ using a global ambient authority, such that any object in the programming world which possesses references of o′ cannot use them, and all paths leading to o′ represent merely possibilities of interaction, but from which no authority can possibly arise.

Could we say more about OCap systems? In OCap systems, there can be no global ambient authority so that an object reference by itself represents both the designation and the authority to use the object. This therefore leads us to be able to make a crucial distinction between MayAccess^{Ind,Now}(o,o′) and MayAccess^{Dir,Now}(o,o′) in the OCap model:

- MayAccess^{Ind,Now}(o,o′) means—*only*
    1. a directed path from o to o′ exists (possible authority)
- MayAccess^{Dir,Now}(o,o′) means—*both*
    1. a directed path from o to o′ exists (possible authority) *and*
    2. o's authority to use o′ *will* succeed

We elaborate the distinction with the following OCap example where there is a particular state σ of the system where, o_1.next points to o_2, and o_2.next points to o_3. o_2.next is a private method that can only be called internally by o_2. In this example, M,$\sigma \vDash$ MayAccess^{Ind,Now}(o_1,o_3) holds, regardless of whether o_2.next is traverseable by o_1. We say here that a path from o_1 to o_3 exists, but is not traverseable by o_1. On the other hand, MayAccess^{Dir,Now}(o_1,o_3) does not hold true, because o_3 is not reachable from o_1 in a single step—o_2 sits between them on the reference graph as an intermediate object, and can possibly prevent or allow traversal from o_1 to o_3 (in this example, o_2 prevents such a traversal).

What does it mean then for MayAccess^{Dir,Now}(*,*) to hold in an OCap model? From the same example, MayAccess^{Dir,Now}(o_1,o_2) holds and implies a stronger form of access—it means that a path exists from o_1 to o_2, and that such a path *is* traverseable. This is because, by the definition of the predicate and configuration of the example, the object reference of o_2 exists within o_1's state. Therefore, o_2 is guaranteed therefore to be accessible, and its authority exercisable, by o_1, without the interference of any ambient authority. Notice how this o_1-o_2 relationship differs from the o_1-o_3 relationship, where o_1 cannot guarantee that it can exercise the authority of o_2.next which points to o_3, since o_2.next does not exist within o_1's state—it is possible that o_2.next is protected by x_2 through encapsulation and data-hiding.
Within an OCap model, we can now be convinced that the stronger

MayAccess^{Dir,Now}(o_1,o_2) $\simeq$
    o_1 has the capability of o_2

while the weaker MayAccess^{Ind,Now}(o_1,o_3) does not say anything about whether o_1 has the capability of o_3, but that only a directed path exists. It does however, represent a necessary condition for capability.

MayAccess^{Ind,Now}(o_1,o_3) $\simeq$
    there is a directed path from o_1 to o_2

o_1 has the capability of o_3 $\implies$
    MayAccess^{Ind,Now}(o_1,o_3)

3.1.2 MayAffect Definitions

With our access predicates in place, we introduce a set of predicates that describe changes to the state of a system. Again, we highlight that these predicates are broad enough to describe both non-OCap and OCap models.

$M,\sigma \vDash \text{MayAffect}^{Now}(x,e) \iff$
$\quad \exists \bar{m},\bar{a},\sigma'.\ x.\bar{m}(\bar{a}) \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{\sigma} \neq \lfloor e \rfloor_{\sigma'}$

$M,\sigma \vDash \text{MayAffect}^{Eve}(x,e) \iff$
$\quad \exists \sigma \in \text{Arising}(M,\sigma).$
$\quad \exists \bar{m},\bar{a},\sigma'.\ x.\bar{m}(\bar{a}) \rightsquigarrow \sigma' \wedge \lfloor e \rfloor_{\sigma} \neq \lfloor e \rfloor_{\sigma'}$

If e is an object:
$\forall e \in \text{Object}.\ \lfloor e \rfloor_{\sigma} \neq \lfloor e \rfloor_{\sigma'} \iff$
$\quad \exists f.\ \lfloor e.f \rfloor_{\sigma} \neq \lfloor e.f \rfloor_{\sigma'}$

Table 2: *MayAffect(o,o') Relations in OCap*

MayAffect^{Now}	$\Longrightarrow$ $\not\Longleftarrow$	MayAffect^{Eve}

3.2 OCap Security Implications

3.2.1 What is protection?

In an object-oriented world[3] , security concerns between objects are often a question of whether what one object can do to another object in *any* eventual state of a system. Because an object encapsulates both internal state and behaviour, strictly speaking, security of an object should govern over both the integrity of the object's fields (internal state) and whether the objects' methods can be called (behaviour). Our predicates are broad enough to enable a discussion of both protection of state (MayAffect^{Eve}) and behaviour ($\text{MayAccess}^{Dir,Eve}$)[4]. We emphasize however that protecting either state or behaviour of an object, does *not* necessarily imply the other. In fact, a common feature in OCap patterns is being able to protect a sensitive object's behaviours (they cannot be called directly by untrusted objects), but at the some allowing the same untrusted objects to modify the object's state in some controlled way.

There are however, some flexibility in working with objects, that allows us to simplify our discussion and work with only a broad definition of state protection in terms of our MayAffect^{Eve} predicate, *without* thinking too much about specific fields of the object we want to protect or the protection of behaviour. Moving away from our broad definition of whether an object may be affected (we defined it as being able to change at least one field of the object), to more precise spcifications of which particular field(s) of the object may be affected, should be trivial. We can in theory also stay and reason within our framework by separating the particular concerned field(s) of the object into separately encapsulated objects. We merely have to be careful as to *which* object's state we want to protect. Furthermore, in theory one can easily introduce a field within an object that behaves like a 'signal' which would be modified whenever a specified behaviour is called. Preventing a particular behaviour to be called by an untrusted object then becomes equivalent to denying the untrusted object the ability to modify the particular state of the signal field of the object.

With these simplifications, protection for us then becomes solely a matter of whether we can allow or deny an object to modify the state of another. In the subsections below we build the neccessary conditions for MayAffect^{Eve}, where MayAffect^{Eve} is placed in the antecedent, and we examine which of the family of MayAccess predicates is placed in the consequent. To help us achieve this, we also formalize our assumptions and the rules of OCap, using our predicates, to help guide us in constructing our necessary conditions.

In building our necessary conditions for state protection in the next subsection notice how we have focused on finding the necessary conditions for the weaker predicate MayAffect^{Eve} rather than MayAffect^{Now} in the antecedent. This is because negation on both sides of the implication, would yield a stronger $\neg\text{MayAffect}^{Eve}$

[3] To simplify our discussion, we work with the variables $\{o,o'\} \in \text{Object}$ for this entire section.

[4] Our $\text{MayAccess}^{Dir,Eve}$ is not weak enough to reason specifically which behaviours can be called. This is however not a big issue in *pure* OCap systems, where often giving away the capability of an object typically means allowing *all* public behaviours of the object to be called without distinction.

in the consequent. In practical terms, if we are concerned with the protection of o′ from o, it is also often not very useful to have a policy where $\neg\text{MayAffect}^{Now}(o,o')$ holds but $\neg\text{MayAffect}^{Eve}(o,o')$ does not. Furthermore, by ensuring $\neg\text{MayAffect}^{Eve}(o,o')$ holds, we can also ensure $\neg\text{MayAffect}^{Now}(o,o')$ holds since by contraposition:

$$[\text{MayAffect}^{Now}(o,o') \implies \text{MayAffect}^{Eve}(o,o')]$$
$$\implies$$
$$[\neg\text{MayAffect}^{Eve}(o,o') \implies \neg\text{MayAffect}^{Now}(o,o')]$$

In building the necessary conditions for the predicate MayAffect^{Eve}, we are also more concerned with finding the configuration of $\text{MayAccess}^{*,Now}$, rather than $\text{MayAccess}^{*,Eve}$. This is because it is easier to implement or prove a configuration of relations that holds in *one* specific state and hence much more useful in practice, than think about whether a configuration holds in *all* possible eventual states.

3.3 Formal Implications

3.3.1 Formalizing OCap rules

We begin our reasoning of protection in our OCap model, by introducing the assumption that all fields in our objects can only be declared private. Consequently, this implies the necessary condition that an object's state can only be modified if one of it's behaviour is called, and therefore require at least one other object in the system that holds its capability. We do not think this is restrictive in any case, the programmer merely has to write explicitly a method to return an object's field.

---

*Private Fields Assumption (PFA)

$$M,\sigma \vDash \exists X^*{\in}\text{Obj}.\ \text{MayAffect}^{Now}(X^*,o') \implies \exists Y^*{\in}\text{Obj}.\ \text{MayAccess}^{Dir,Now}(Y^*,o')$$

By contraposition,

$$M,\sigma \vDash \forall Y^*{\in}\text{Obj}.\ \neg\text{MayAccess}^{Dir,Now}(Y^*,o') \implies \forall X^*{\in}\text{Obj}.\ \neg\text{MayAffect}^{Now}(X^*,o')$$

This means that in order for an object X*'s state to change, it must be done through some object Y* calling its behaviour (Y* can refer to the same object as X*). Equivalently, denying all objects in the system the ability to call an object's behaviour implies that no object can modify the object's state.

Rule 1: Objects can only interact with each other through sending messages on capabilities. Therefore, if an object o can affect o′, then there must be a path from o to o′:

---

*Necessary Path Condition 1 (NPC1)

$$M,\sigma \vDash \text{MayAffect}^{Eve}(o,o') \implies \text{MayAccess}^{Ind,Eve}(o,o')$$

By contraposition,

$$M,\sigma \vDash \neg\text{MayAccess}^{Ind,Eve}(o,o') \implies \neg\text{MayAffect}^{Eve}(o,o')$$

Here, we immediately see a first defensive outcome of the OCap model. Having no paths from object o to o′ guarantees that the state of object o′ cannot be modified by object o.

Rule 2: Objects cannot forge capabilities, and only connectivity begets connectivity.

---

*Global Path Connectivity (GPC)

$$[\exists Y^*{\in}\text{Object}.\ \text{MayAccess}^{Ind,Now}(Y^*,o')] \iff [\exists X^*{\in}\text{Object}.\ \text{MayAccess}^{Dir,Now}(X^*,o')]$$

By contraposition,

$$[\forall X^*{\in}\text{Object}.\ \neg\text{MayAccess}^{Dir,Now}(X^*,o')] \iff [\forall Y^*{\in}\text{Object}.\ \neg\text{MayAccess}^{Ind,Now}(Y^*,o')]$$

In addition to path being a necessary condition for capability, GPC gives us an additional new relation between the two concepts over the entire system, and says that in a given system of objects, iff there exists an object Y* which has a path to o′, then there exists an object X* that has the capability of o′. Looking from the left to right direction, this is directly derived from Rule 2, since the path from Y* to o′ must either be a direct path (Y*==X*) or if not at the very least an object X* must have a direct path to o′, in order for the path from Y* to o′ to be well-established, where X* is the last object in the path before o′. Moving from the right to left direction, GPC says that if an object X* has a capability then there exist an object Y* with a path to o′. We can make this assertion because we know by definition that it holds when Y* refers to the same object as X*.

We now look for a relation between a path configuration at a state σ (*Now*) and eventual path configurations arising from σ (*Eve*).

---

*Eventual Path Connectivity (EPC)

$M,\sigma \vDash \text{MayAccess}^{Ind,Eve}(o,o')$
$\implies$
$\text{MayAccess}^{Ind,Now}(o,o')$
$\vee$
$\{\exists X^* \in \text{Obj}.[(\text{MayAccess}^{Ind,Now}(o,X^*) \vee \text{MayAccess}^{Ind,Now}(X^*,o)) \wedge \text{MayAccess}^{Ind,Eve}(X^*,o')]\}$

By contraposition,
$M,\sigma \vDash \neg\text{MayAccess}^{Ind,Now}(o,o')$
$\wedge$
$\forall X^* \in \text{Obj}.\neg[(\text{MayAccess}^{Ind,Now}(o,X^*) \vee \text{MayAccess}^{Ind,Now}(X^*,o)) \wedge \text{MayAccess}^{Ind,Eve}(X^*,o')]$
$\implies$
$\text{MayAccess}^{Ind,Eve}(o,o')$

The contraposition result of EPC says that o′ is eventually path-isolated from o in all possible states arising from σ when o is does not have a path to o′ in σ *and* for all object X* that has eventual paths to o, it is not true in σ that either o is connected to X* or X* is connected to o by some path.

Using NPC1 and EPC, we derive NPC2 below which is a relation between state protection in eventual outcomes, and present path configurations. EPC says that to protect o′ from o in all states arising from a state σ, we only need to ensure isolation of o′ in σ, provided o′ exists in σ:

---

*Necessary Path Condition 2 (NPC2)

$M,\sigma \vDash \text{MayAffect}^{Eve}(o,o')$
$\implies$
$\text{MayAccess}^{Ind,Now}(o,o')$
$\vee$
$\{\exists X^* \in \text{Obj}.[(\text{MayAccess}^{Ind,Now}(o,X^*) \vee \text{MayAccess}^{Ind,Now}(X^*,o)) \wedge \text{MayAccess}^{Ind,Eve}(X^*,o')]\}$

By contraposition,
$M,\sigma \vDash \neg\text{MayAccess}^{Ind,Now}(o,o')$
$\wedge$
$\forall X^* \in \text{Obj}.\neg[(\text{MayAccess}^{Ind,Now}(o,X^*) \vee \text{MayAccess}^{Ind,Now}(X^*,o)) \wedge \text{MayAccess}^{Ind,Eve}(X^*,o')]$
$\implies$
$\neg\text{MayAffect}^{Eve}(o,o')$

The interpretation of NPC2 is identical to EPC, with protection of state being made an explicit consequent from the conditions of EPC. So far these results serve as a good base to enforce $\neg\text{MayAffect}^{Eve}(o,o')$ but we require *stronger* necessary conditions for $\neg\text{MayAffect}^{Eve}(o,o')$, because enforcing protection of an object's state with just these conditions require the impractical configuration that o is path-isolated from o′ on the reference graph (no paths). Cooperation between objects imply that there needs to be some path established between the objects for interaction to take place. Therefore, we can expect neither $\neg\text{MayAccess}^{Ind,Now}$(o,o′) nor $\neg\text{MayAccess}^{Ind,Eve}$(o,o′) to hold in a system that allows cooperation between o and o′.

So far we have only defined the relation between the existence of paths and an object's state. This is not the full picture, as we have not said anything about capabilities, or whether the paths are traverseable. We now examine whether it is possible to build a relation from MayAffect^{Eve} to $\text{MayAccess}^{Dir,Eve}$.

---

*Necessary Execution Condition 1 (NEC1)

$M,\sigma \vDash \text{MayAffect}^{Eve}(o,o') \implies \exists X^* \in \text{Object}.\ \text{MayAccess}^{Dir,Eve}(X^*,o')$

By contraposition and quantifier equivalence,
$M,\sigma \vDash [\forall X^* \in \text{Object}.\ \neg\text{MayAccess}^{Dir,Eve}(X^*,o')] \implies \neg\text{MayAffect}^{Eve}(o,o')$

NEC1 is a stronger version of NPC1 and says that in order for an object o to modify the state of o′, there must exist an object X* that has a direct path to o′, and that X* *can* traverse such a path. This is derived trivially from our PFA assumption in the beginning. Consequently, in our contraposition result, we can deny *all* objects in our system capability of o′ to enforce the protection of o′ from o. NEC1 does not yield a very useful result, as this is just another way of implementing protection of o′ from o through denying all paths to o′ in the system reference graph. To see why denying all objects the capability of o′ is equivalent to

denying all paths to o', see the results from GPC.

In NEC1, if X* refers to the same object as o, then we have a straightforward configuration where object o has the capability of o' and can therefore affect o'. However, very crucially, o having the capability of o' is *not* a sufficient condition—denying o the capability of o' does *not* deny o the ability to affect o', since X* can refer to an object that is *not* o. NEC2 below is an expansion of NEC1 to illustrate this crucial result in the OCap model.

---

*NECESSARY EXECUTION CONDITION 2 (NEC2)
$M,\sigma \vDash \text{MayAffect}^{Eve}(o,o') \implies$
$[\text{MayAccess}^{Dir,Eve}(o,o') \lor$
$\exists X^* \in \text{Object}, X^* \neq o.\ \text{MayAccess}^{Dir,Eve}(X^*,o')]$

By contraposition,
$M,\sigma \vDash [\neg\text{MayAccess}^{Dir,Eve}(o,o') \land$
$\forall X^* \in \text{Object}, X^* \neq o.\ \neg\text{MayAccess}^{Dir,Eve}(X^*,o')]$
$\implies \neg\text{MayAffect}^{Eve}(o,o')$

We now examine the second half of the necessary condition in NEC2:
$[\exists X^* \in \text{Object}, X^* \neq o.\ \text{MayAccess}^{Dir,Eve}(X^*,o')]$
which means that there exists object X* that is *not* object o, that must have the capability of o'. We now make use of our result in NPC1, which states that a path must exist from o to o' in order for $\text{MayAffect}^{Eve}(o,o')$ to hold. Since X* is not o, and X* has a direct path to o', we have to connect o to X* in order to connect o to o'. With this, we can now construct NEC3 from NEC2 and NPC1.

---

*NECESSARY EXECUTION CONDITION 3 (NEC3)
$M,\sigma \vDash \text{MayAffect}^{Eve}(o,o') \implies$
$\text{MayAccess}^{Dir,Eve}(o,o') \lor$
$[\exists X^* \in \text{Object}, X^* \neq o.\ \text{MayAccess}^{Ind,Eve}(o,X^*) \land$
$\text{MayAccess}^{Dir,Eve}(X^*,o')]$

By contraposition,
$M,\sigma \vDash [\exists X^* \in \text{Object}, X^* \neq o.\ \text{MayAccess}^{Ind,Eve}(o,X^*)$
$\land \text{MayAccess}^{Dir,Eve}(X^*,o')] \implies$
$\text{MayAccess}^{Dir,Eve}(o,o') \lor$

ShuPeng: Questions for myself... Can I can form necessary AND sufficient conditions for MayAffect?
The power of OCap patterns hence lies in providing concrete examples of a system of cooperation that allows the existence of paths between objects for cooperation while still dictating the degree of control of one object can have over another. The logics we have developed so far illuminate the power of attenuating objects X* that can enable protection. Indeed attenuating objects feature prominently in the literature of OCap patterns which we shall see in the next section.

3.4 Pattern 1: The JavaScript DOM Tree

We use a JavaScript DOM Tree OCap pattern largely inspired by the example in Devriese et al.[1] where they use a Kripke worlds framework to reason about the pattern. We define the following variables throughout our pattern:

- o,o' ∈Object
- Node,ReNode ⊆Object
- n,n' ∈Node
- rn,rn' ∈ReNode

---

*NODE VULNERABILITY
$\forall o, n.\ \text{MayAccess}^{Dir,Eve}(o,n)$
$\implies$
$\text{MayAffect}^{Eve}(o,n) \land$
$\forall n'.\ \text{MayAccess}^{Dir,Eve}(o,n')$

The vulnerability of a node lies in the fact that it contains a public method `setProperty(key,value)` that will modify an internal mapping data structure. A node also has a public field `parent` that will return the capability of its parent node. Consequently, this allows an object which has the capability of any one node in the tree to navigate up to the root node (Document), and consequently navigate to all other nodes in the tree.

---

*POLICY 1: NECESSARY CONDITION
$\forall n,o,\ RN \subseteq \text{ReNode}.$
$\text{Dom}(RN,n) \land \text{MayAffect}^{Eve}(o,n)$
$\implies$
$\exists rn \in RN.\ \text{MayAccess}^{Ind,Eve}(o,rn)$
$\land\ \text{MayAccess}^{Dir,Eve}(rn,n)$

This policy states that if an Object o may affect the state of a Node n, and that n is dominated(protected) by a set of ReNodes RN, then it implies that a path exists from o to some rn in RN, and that rn has strong access to n. From Policy 1, using contraposition, we derive the relation:

$\forall$n,o, RN$\subseteq$ReNode.
[$\forall$rn$\in$RN. $\neg$MayAccess^{Ind,Eve}(o,rn) $\vee$
$\forall$rn$\in$RN. $\neg$MayAccess^{Dir,Eve}(rn,n)]
$\implies$ $\neg$MayAffect^{Eve}(o,n) $\vee$ $\neg$Dom(RN,n)

Note that without prescribing any rules to ReNode, we can deny object o the ability to affect node n by enforcing that no path exists from o to any rn in the set RN that dominates n, which by implication is another way of saying isolation from o to n, since:

$\forall$n,o. Dom(RN,n) $\wedge$
$\forall$rn$\in$RN. $\neg$MayAccess^{Ind,Eve}(o,rn) $\implies$
$\neg$MayAccess^{Ind,Eve}(o,n).

Another implication is that we can also deny object o the ability to affect node n by enforcing $\forall$rn$\in$RN. $\neg$MayAccess^{Dir,Eve}(rn,n). This is however, *not* enforceable. This is because by definition a ReNode rn holds a field containing the capability of the node n it is meant to protect when the ReNode is constructed, and field that holds the capability of the node is private and constant, that cannot be subsequently changed or removed. Consequently, there must exist at least one ReNode rn in the dominating RN set that holds the capability of n:

$\exists$rn $\in$RN. Dom(RN,n) $\wedge$MayAccess^{Dir,Eve}(rn,n)
$\implies$
$\neg$($\forall$rn$\in$RN. $\neg$MayAccess^{Dir,Eve}(rn,n))

Policy 2: Domination of Nodes

3.5 Pattern 2: Caretaker

3.6 Pattern 3: Membrane

Policy 1:

References

[1] Dominique Devriese, Lars Birkedal, and Frank Piessens. Reasoning about object capabilities with logical relations and effect parametricity. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, pages 147–162. IEEE, 2016.

[2] Sophia Drossopoulou, James Noble, Mark S Miller, and Toby Murray. Reasoning about risk and trust in an open word. 2015.

[3] Mark S Miller. Robust composition: Towards a unified approach to access control and concurrency control. 2006.

[4] Mark S Miller and Jonathan S Shapiro. Paradigm regained: Abstraction mechanisms for access control. In *Annual Asian Computing Science Conference*, pages 224–242. Springer, 2003.