Keywords
object-capability, concurrency, message-passing, policy, types, Curry-Howard

ABSTRACT
Drossopoulou and Noble [6] argue persuasively for the need for a means to express policy in object-capability-based systems. We investigate a practical means to realize their aim via the Curry-Howard isomorphism [1] [13]. Specifically, we investigate representing policy as types in a behavioral type system for the RHO-calculus [15], a reflective higher-order variant of the π-calculus [18].

Submission to WPES

Policy as Types

L.G. Meredith
Biosimilarity, LLC
lgreg.meredith@biosimilarity.com

Mike Stay
Biosimilarity, LLC
mike.stay@biosimilarity.com

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

1. INTRODUCTION

The object-capability (ocap) security model grew out of the realization that good object-oriented programming practice leads to good security properties. Separation of duties leads to separation of authority; information hiding leads to integrity; message passing leads to authorization; and dependency injection to authority injection. An ocap-secure programming language enforces these patterns: the only way an object can modify any state but its own is by sending messages on the object references it possesses. Authority can then be denied simply by not providing the relevant object reference. Ocap languages do not provide ambient, undeniable authority such as the global variables in JavaScript, static mutable fields in Java and C#, or allowing access to arbitrary objects through pointer arithmetic as in C and C++.

The electronic society is moving steadily towards the object capability model. ECMAScript 5, the standard version of JavaScript adopted by all modern browsers, introduced a new security API that enables the web browser to be turned into an ocap-safe platform; Google's Caja project is an implementation [11]. Brendan Eich, CTO of Mozilla, wrote, "In SpiderMonkey + Gecko in Firefox, and probably in other browsers, we actually use OCap under the hood and have for years. In HTML5, the WindowProxy/Window distinction was finally specified, as an ad-hoc instance of OCap membranes. Any time we deviate from OCap, we regret it for both security bug and access-checking overhead reasons." [7]

We would like a way to declare the authority that an object ought to possess and then check whether the implementation matches the intent; that is, we would like a language for declaring security policy. Drossopoulou and Noble [6] convincingly argue that none of the current specification methods adequately capture all of the capability policies required to support ocap systems. The most intriguing aspect of the capability policies proposed in [6] is *deniability*. Deniability differs from usual style of specifications in the following two aspects: a) it describes policies which are *open*, i.e. apply to a module as well as all its extensions, and b) describes *necessary* conditions for some effect to take place. For example, a policy may require that in order for any piece of code to modify the balance of a bank account, it needs to have access to that account (access to the account is a necessary condition), and that this property is satisfied by all extensions of the account library (open). In contrast, classical Hoare Logic specifications [8] describe sufficient conditions and are closed. In this paper we concentrate on the use of the RHO-calculus [15] [1] for expressing deniability. To express necessary conditions, we invert the assertion and use bisimulation (e.g. any piece of code which does not have access to the account will be bisimilar to code in which the balance of the account is not modified). To express openness, we adapt the concept of the adjoint to the separation operator, i.e. any further code P which is attached to the current code will be bisimilar to P in parallel with code which does not modify the balance of the account.

In short, we show that the RHO-calculus is an ocap language and sketch a translation from a subset of JavaScript into the calculus; we also demonstrate that the corresponding Hennessy-Milner logic suffices to capture the key notion of deniability.

In the larger scheme, this identifies [6]'s notion of policy with the proposition-as-types paradigm, also known as The Curry-Howard isomorphism [1] [13].

[1] the *R*eflective *H*igher-*O*rder calculus, not to be confused with the ρ-calculus [5]

The Curry-Howard isomorphism relates formal logic to type theory; it says that propositions are to types as proofs are to programs. Hennessy-Milner logic lets us treat the type of a concurrent process as an assertion that it belongs to a set of processes, all of which satisfy some property. We can treat these properties as contracts governing the behavior of the process [14]; in other words, the language of behavioral types suffices as a security policy language.

1.0.1 Organization of the rest of the paper

We introduce the calculus. So equipped we illustrate translating a simple ocap example into the calculus. Next, we introduce the logic, and illustrate how we can reason about properties of the translated program and from there move to interpreting policy, focusing on deniability. Finally, we conclude with a discussion of some of the limitations of the approach and how we might approach these in future work.

2. THE CALCULUS

Before giving the formal presentation of the calculus and logic where we interpret ocap programs and policy, respectively, we begin with a couple of examples that illustrate a design pattern used over and over in this paper. The first is a mutable single-place cell for storing and retrieving state.

```
def Cell(slot, state) ⇒ {
  new (v) {
    v!(state)
    match {
      slot?get(ret) ⇒ {
        v?(s) ⇒ ret!(s)
        Cell(slot, s)
      }
      slot?set(s) ⇒ { Cell(slot, s) }
    }
  }
}
```

We read this as saying that a *Cell* is parametric in some (initial) *state* and a *slot* for accessing and mutating the cell's state. A *Cell* allocates a private channel v where it makes the initially supplied *state* available to its internal computations. If on the channel *slot* it receives a *get* message containing a channel *ret* indicating where to send the state of the cell, it accesses the private channel v and sends the value it received on to the *ret* channel; then it resumes behaving as a *Cell*. Alternatively, if it receives a *set* message containing some new state s, it simply continues as a *Cell* instantiated with accessor *slot* and state s.

Despite the fact that this example bears a striking resemblance to code in any of a number of popular libraries, notably the `AKKA` library for the `Scala` language, it is only a mildly sugared form of the direct representation in our calculus. Notice also that like modern application code, it exhibits encapsulation and separation of implementation from API: a cell's internal access to `state` is kept in a private channel `v` while external access is through `slot`. Another nice thing about this design pattern is that it scales through composition: when translating ocap examples expressed in `ECMAScript` to RHO-calculus we will effectively treat the state of an object as a parallel composition of cells comprising its state.

The second gadget is an immutable map:

```
def Map(chan, key_1, state_1, ..., key_n, state_n) ⇒ {
  new (v_1, ... v_n) {
    v_1!(state_1)
    ...
    v_n!(state_n)
    chan?get(key_1, ret) ⇒ {
      v_1?(x) ⇒ ret!(x)
      Map(chan, key_1, state_1, ..., key_n, state_n)
    }
    ...
    chan?get(key_n, ret) ⇒ {
      v_n?(x) ⇒ ret!(x)
      Map(chan, key_1, state_1, ..., key_n, state_n)
    }
  }
}
```

The design pattern for maps is related to the one for cells, but is missing the capability to mutate the map. The code does not serve a request to set any of the values it contains. With these two patterns in hand it will be much easier to treat the examples. Moreover, as we will see in the sequel, even with these basic patterns we can already reason about policy. The shape of the reasoning about policy for these examples scales compositionally to reasoning about application-level policy.

2.0.2 The RHO-calculus

The RHO-calculus [16] is a variant of the asynchronous polyadic π-calculus. When names are polarized [19], the π-calculus is an ocap language. Pierce and Turner [20] defined the programming language Pict as sugar over the polarized π-calculus together with some supporting libraries; unfortunately, the authority provided by the libraries violated the principle of least authority. Košík refactored the libraries to create Tamed Pict, recovering an ocap language, then used it to write a defensively consistent operating system [12]. The RHO-calculus differs from the π-calculus in that names are quoted processes rather than generated by the ν operator. Both freshness and polarization are provided through the use of namespaces at the type level.

We let P, Q, R range over processes and x, y, z range over names, and $\vec{x}$ for sequences of names, $|\vec{x}|$ for the length of $\vec{x}$, and $\{\vec{y}/\vec{x}\}$ as partial maps from names to names that may be uniquely extended to maps from processes to processes [16].

$$
\begin{array}{rl}
M, N ::= & 0 \qquad\qquad\qquad\qquad \text{stopped process} \\
& \mid x?(y_1, \ldots, y_N) \Rightarrow P \qquad \text{input} \\
& \mid x!(Q_1, \ldots, Q_N) \qquad\quad \text{output} \\
& \mid M + N \qquad\qquad\quad \text{choice} \\
P, Q ::= & M \qquad\qquad\qquad\quad \text{include IO processes} \\
& \mid P \mid Q \qquad\qquad\quad \text{parallel} \\
& \mid *x \qquad\qquad\qquad\quad \text{dereference} \\
x, y ::= & @P \qquad\qquad\qquad\quad \text{reference}
\end{array}
$$

The examples from the previous section use mild (and entirely standard) syntactic sugar: `def` making recursive definitions a little more convenient than their higher-order encodings, `new` making fresh channel allocation a little more convenient and `match` for purely input-guarded choice. Additionally, the examples use {}-enclosed blocks together with line breaks, rather than | for parallel composition. Thus the expression $v!(state)$ is actually a thread running in parallel with the `match` expression in the *Cell* definition. The interested reader is directed to the appendix for more details.

2.1 Free and bound names

The syntax has been chosen so that a binding occurrence of a name is sandwiched between round braces, $(\,\cdot\,)$. Thus, the calculation of the free names of a process, P, denoted $\mathcal{FN}(P)$ is given recursively by

$$
\begin{aligned}
&\mathcal{FN}(0) \coloneqq \emptyset \\
&\mathcal{FN}(x?(y_1, \ldots, y_N) \Rightarrow P) \coloneqq \\
&\quad \{x\} \cup (\mathcal{FN}(P) \setminus \{y_1, \ldots y_N\}) \\
&\mathcal{FN}(x!(Q_1, \ldots, Q_N)) \coloneqq \{x\} \cup \bigcup \mathcal{FN}(Q_i) \\
&\mathcal{FN}(P \mid Q) \coloneqq \mathcal{FN}(P) \cup \mathcal{FN}(Q) \\
&\mathcal{FN}(*x)\, x \coloneqq \{x\}
\end{aligned}
$$

An occurrence of x in a process P is *bound* if it is not free. The set of names occurring in a process (bound or free) is denoted by $\mathcal{N}(P)$.

2.2 Structural congruence

The *structural congruence* of processes, noted $\equiv$, is the least congruence containing α-equivalence, $\equiv_\alpha$, that satisfies the following laws:

$$
\begin{aligned}
P \mid 0 \quad &\equiv P \equiv \quad 0 \mid P \\
P \mid Q \quad &\equiv \quad Q \mid P \\
(P \mid Q) \mid R \quad &\equiv \quad P \mid (Q \mid R)
\end{aligned}
$$

Name equivalence. As discussed in [15] the calculus uses an equality, $\equiv_N$, pronounced name-equivalence, recursively related to α-equivalence and structural equivalence, to judge when two names are equivalent, when deciding synchronization and substitution.

2.3 Semantic substitution

The engine of computation in this calculus is the interaction between synchronization and a form of substitution, called semantic substitution. Semantic substitution differs from ordinary syntactic substitution in its application to a dropped name. For more details see [16]

$$
(*x)\{\widehat{@Q/@P}\} \quad \coloneqq \quad \begin{cases} Q & x \equiv_N @P \\ *x & otherwise \end{cases}
$$

Finally equipped with these standard features we can present the dynamics of the calculus.

2.4 Operational Semantics

The reduction rules for RHO-calculus are

$$
\frac{x_0 \equiv_N x_1, |\vec{y}| = |\vec{Q}|}{x_0?(\vec{y}) \Rightarrow P \mid x_1!(\vec{Q}) \to P\{@\vec{Q}/\vec{y}\}} \quad (\textsc{Comm})
$$

In addition, we have the following context rules:

$$
\frac{P \to P'}{P \mid Q \to P' \mid Q} \quad (\textsc{Par})
$$

$$
\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q} \quad (\textsc{Equiv})
$$

The context rules are entirely standard and we do not say much about them here. The communication rule makes it possible for agents to synchronize at name-equivalent guards and communicate processes packaged as names. Here is a simple example of the use of synchronization, reduction and quasiquote: $x(z) \Rightarrow w!(y!(*z)) \mid x!(P) \to w!(y!(P))$. The input-guarded process, $x(z) \Rightarrow w!(y!(*z))$, waits to receive the code of P from the output-guarded data, $x!(P)$, and then reduces to a new process the body of which is rewritten to include P, $w!(y!(P))$.

Bisimulation. One of the central features of process calculi, referred to in the literature as bisimulation, is an effective notion of substitutability, i.e. when one process can be substituted for another, and with it a range of powerful proof techniques. For this paper we focus principally on the use of logic to interpret policy and so move a discussion of bisimulation to an appendix. The main theorem of the next section relates the logic to bisimulation. The reader unfamiliar with this important aspect of programming language semantics is strongly encouraged to look at [22].

3. INTERPRETING CAPABILITIES

Rather than giving a translation of the whole of `ECMAScript` into RHO-calculus, we focus on one example from [6] designed to illustrate key aspects of object-capabilities and requirements for a policy language.

```
var makeMint = () => {
  var m = WeakMap();
  var makePurse = () => mint(0);
  var mint = balance => {
    var purse = def({
      getBalance: () => balance,
      makePurse: makePurse,
      deposit:
       (amount, srcP) => Q(srcP).then(src => {
         Nat(balance + amount);
         m.get(src)(Nat(amount));
         balance += amount;
       })
    });
    var decr = amount => {
      balance = Nat(balance - amount);
    };
    m.set(purse, decr);
    return purse;
  };
  return mint;
};
```

Interpreting this code illustrates how surprisingly intuitive and natural it is to represent and reason about object-capabilities in RHO-calculus. Essentially, a capability—*i.e.* access to some behavior, power or authority—is represented by a channel. When an agent in the RHO-calculus has access to a channel it has exactly that: access to some behavior, power or authority.

Armed with *Cell* and *Map* from the first section, we can translate each of the language elements from the bank example into RHO-calculus. Note that the interpretation takes a channel k on which to signal completion of the statement; synchronous JavaScript necessarily needs to be transformed into continuation-passing style as part of the interpretation in RHO-calculus. The asynchronous `Q` library used in the definition of the `deposit` method, on the other hand, is unneeded because RHO-calculus is inherently asynchronous.

$$[\![\texttt{var } x; P]\!](x,k) \coloneqq Cell(x, \texttt{undefined}) \mid [\![P]\!](x,k)$$

$$[\![\texttt{var } x = v; P]\!](x,k) \coloneqq \texttt{new}(k_1)\{Cell(x, [\![v]\!](x,k_1)) \mid k_1?() \Rightarrow [\![P]\!](x,k)\}$$

$$[\![P; Q]\!](x,k) \coloneqq \texttt{new}(k_1)\{[\![P]\!](x,k_1) \mid k_1?() => [\![Q]\!](x,k)\}$$

$$[\![\texttt{i+= a}; P]\!](x,k) \coloneqq \texttt{new}(k_1)\{\ \texttt{i}!get(r)\ \ r?(v) \Rightarrow \texttt{\{i}!set(v + [\![\texttt{a}]\!](x,k)) \mid k_1!()\texttt{\}}\ \ [\![P]\!](x,k_1)\ \}$$

$$[\![\texttt{def}(\{key_1 : state_1, \ldots, key_n : state_n\})]\!](x,k) \coloneqq Map(x, key_1, state_1, \ldots, key_n, state_n) \mid k!()$$

We do not give an explicit translation of `WeakMap` because it is quite large, but it is essentially the same as *Map* plus some machinery for performing updates.

The key point is that the translation is compositional, so we can build up the translation of the whole from the translation of the parts. Moreover, we can reason equationally in the translated code. For example, to translate the code fragment below we calculate directly

$$\begin{array}{l}
[\![\texttt{var } balance = initAmt;\ balance \mathrel{+}= amount; P]\!](x,k) \\
= \ (\mathrm{defn}) \\
\{ \\
\quad Cell(balance, [\![initAmt]\!](x,k)) \\
\quad k!() \\
\quad \mathrm{new}\ (rslt, k_1)\ \{ \\
\quad\quad balance!get(rslt) \\
\quad\quad rslt?(b) \Rightarrow\ \{ \\
\quad\quad\quad balance!set(b + [\![amount]\!](x,k)) \\
\quad\quad\quad k_1!() \\
\quad\quad \} \\
\quad\quad k_1?() \Rightarrow [\![P]\!](x,k) \\
\quad \} \\
\} \\
= \ (\mathrm{extrusion\ laws\ for\ new}) \\
\mathrm{new}\ (rslt, k_1)\ \{ \\
\quad Cell(balance, [\![initAmt]\!](x,k)) \\
\quad k!() \\
\quad balance!get(rslt) \\
\quad rslt?(b) \Rightarrow \{ \\
\quad\quad balance!set(b + [\![amount]\!](x,k)) \\
\quad\quad k_1!() \\
\quad \} \\
\quad k_1?() \Rightarrow [\![P]\!](x,k) \\
\}
\end{array}$$

4. LOGIC

Namespace logic resides in the subfamily of Hennessy-Milner logics known as spatial logics, discovered by Caires and Cardelli [3]. Thus, in addition to the action modalities, we also find at the logical level formulae for *separation* corresponding to the structural content of the parallel operator at the level of the calculus. Likewise, we have quantification over names. There are important differences between the logic presented here and Caires' logic. The interested reader is referred to [15].

4.1 Examples

A principal advantage to using logical formulae of this kind is that they denote classes or programs. Unlike the usual notion of type, where a type's inhabitants are instances of data structures, in namespace logic a formula (i.e. a *type*) ϕ is inhabited by programs—namely all the programs that satisfy the formulae. Namespace logic takes this a step further. Because it is built on a reflective programming language (the RHO-calculus) it also has formulae, $\ulcorner\phi\urcorner$, for describing classes of names. Said another way, the inhabitants of types of the form $\ulcorner\phi\urcorner$ are names and hence these types have the right to be called namespaces. The next two examples show how useful namespaces are in the security setting.

4.1.1 Controlling access to namespaces

Suppose that $\ulcorner\phi\urcorner$ describes some namespace, *i.e.* some collection of names. We can insist that a process restrict its next input to names in that namespace by insisting that it witness the formula

$$\langle\ulcorner\phi\urcorner?b\rangle\, true\, \&\neg\langle\ulcorner\neg\phi\urcorner?b\rangle true$$

which simply says the the process is currently able to take input from a name in the namespace $\ulcorner\phi\urcorner$ and is not capable of input on any name not in that namespace. In a similar manner, we can limit a server to serving only inputs in $\ulcorner\phi\urcorner$ throughout the lifetime of its behavior [2]

$$\mathsf{rec}\ X\ .\ \langle\ulcorner\phi\urcorner?b\rangle X\&\neg\langle\ulcorner\neg\phi\urcorner?b\rangle X$$

This formula is reminiscent of the functionality of a firewall, except that it is a *static* check. A process witnessing this formula will behave as though it were behind a firewall admitting only access to the ports in $\ulcorner\phi\urcorner$ without the need for the additional overhead of the watchdog machinery.

We will use refinements this basic technique of walling off a behavior behind a namespace when reasoning about the ocap examples and especially in the context of deniability.

[2]Of course, this formula also says the server never goes down, either—or at least is always willing to take such input.

$$\begin{array}{lr}
\phi,\psi ::= true & \text{verity}\\
\quad \mid 0 & \text{nullity}\\
\quad \mid \neg\phi & \text{negation}\\
\quad \mid \phi\&\psi & \text{conjunction}\\
\quad \mid \phi \mid \psi & \text{separation}\\
\quad \mid \urcorner b\ulcorner & \text{disclosure}\\
\quad \mid a!(\phi) & \text{dissemination}\\
\quad \mid \langle a?b\rangle\phi & \text{reception}\\
\quad \mid \mathsf{rec}\ X\ .\ \phi & \text{greatest fix point}\\
\quad \mid \forall n:\psi\ .\ \phi & \text{quantification}\\
a ::= \ulcorner\phi\urcorner & \text{demarcation}\\
\quad \mid b & ...\\
b ::= \ulcorner P\urcorner & \text{nomination}\\
\quad \mid n & ...
\end{array}$$

We let $PForm$ denote the set of formulae generated by the ϕ-production, $QForm$ denote the set of formulae generated by the a-production and $\mathcal{V}$ denote the set of propositional variables used in the rec production. Additionally, we let $PForm_{-Par}$ denote the separation-free fragment of formulae.

Inspired by Caires' presentation of spatial logic [3], we give the semantics in terms of sets of processes (and names). We need the notion of a valuation $v : \mathcal{V} \to \wp(Proc)$, and use the notation $v\{\mathcal{S}/X\}$ to mean

$$v\{\mathcal{S}/X\}(Y) \quad = \quad \begin{cases} S & Y = X \\ v(Y) & otherwise \end{cases}$$

The meaning of formulae is given in terms of two mutually recursive functions,

$$\begin{array}{c}
[\![-]\!](-) : PForm \times [\mathcal{V} \to \wp(Proc)] \to \wp(Proc)\\
((-))(-) : QForm \times [\mathcal{V} \to \wp(Proc)] \to \wp(@Proc)
\end{array}$$

taking a formula of the appropriate type and a valuation, and returning a set of processes or a set of names, respectively.

$[\![true]\!](v) \coloneqq Proc$
$[\![0]\!](v) \coloneqq \{P : P \equiv 0\}$
$[\![\neg\phi]\!](v) \coloneqq Proc / [\![\phi]\!](v)$
$[\![\phi \& \psi]\!](v) \coloneqq [\![\phi]\!](v) \cap [\![\psi]\!](v)$
$[\![\phi \mid \psi]\!](v) \coloneqq$
$\quad \{P : \exists P_0, P_1 . P \equiv P_0 \mid P_1,\ P_0 \in [\![\phi]\!](v),\ P_1 \in [\![\psi]\!](v)\}$
$[\![\ulcorner b \urcorner]\!](v) \coloneqq \{P : \exists x . P \equiv *x,\ x \in (\!(b)\!)(v)\}$
$[\![a!(\phi)]\!](v)$
$\coloneqq \{P : \exists P' . P \equiv x!(P'),\ x \in (\!(a)\!)(v),\ P' \in [\![\phi]\!](v)\}$
$[\![\langle a?b \rangle \phi]\!](v) \coloneqq$
$\quad \{P : \exists P' . P \equiv x?(y) \Rightarrow P', x \in (\!(a)\!)(v),$
$\quad\quad \forall c . \exists z . P'\{z/y\} \in [\![\phi\{c/b\}]\!](v)\}$
$[\![\mathsf{rec}\ X\ .\ \phi]\!](v) \coloneqq \cup\{\mathcal{S} \subseteq Proc : \mathcal{S} \subseteq [\![\phi]\!](v\{\mathcal{S}/X\})\}$
$[\![\forall n : \psi\ .\ \phi]\!](v) \coloneqq \cap_{x \in ((@\psi))(v)} [\![\phi\{x/n\}]\!](v)$
$((\ulcorner \phi \urcorner))(v) \coloneqq \{x : x \equiv_N @P, P \in [\![\phi]\!](v)\}$
$((\ulcorner P \urcorner))(v) \coloneqq \{x : x \equiv_N @P\}$

We say P witnesses ϕ (resp., x witnesses $\ulcorner \phi \urcorner$), written $P \models \phi$ (resp., $x \models \ulcorner \phi \urcorner$) just when $\forall v . P \in [\![\phi]\!](v)$ (resp., $\forall v . x \in [\![\ulcorner \phi \urcorner]\!](v)$).

THEOREM 4.1.1 (EQUIVALENCE). $P \dot{\approx} Q \Leftrightarrow \forall \phi \in PForm_{-Par} . P \models \phi \Leftrightarrow Q \models \phi$.

The proof employs an adaptation of the standard strategy. A consequence of this theorem is that there is no algorithm guaranteeing that a check for the witness relation will terminate. However there is a large class of interesting formulae for which the witness check does terminate, see [3]. *Nota bene*: including separation makes the logic strictly more observant than bismulation.

4.1.2 Syntactic sugar

In the examples below, we freely employ the usual DeMorgan-based syntactic sugar. For example, $\phi \Rightarrow \psi \coloneqq \neg(\phi \& \neg\psi)$ and $\phi \vee \psi \coloneqq \neg(\neg\phi \& \neg\psi)$. Also, when quantification ranges over all of $@Proc$, as in $\forall n : \ulcorner true \urcorner\ .\ \phi$, we omit the typing for the quantification variable, writing $\forall n\ .\ \phi$.

5. INTERPRETING POLICY

Drossopoulou and Noble [6] argue for several different types of policy specifications. In this paper we focus on deniability policies, since the others have ready translations in this setting. Deniability is the one type of policy specification where the translation is not at all straightforward. For the reader familiar with logics of concurrency the core idea is to use rely-guarantee combined with the adjunct to spatial separation.

A visual inspection of the code for *Cell* reveals that if a process P's names are fresh with respect to *slot*, then P cannot directly affect any $Cell(slot, state)$. Unfortunately, visual inspection is not reliable in settings only mildly more complex, as in the bank example above; Yee reports that a 90-man-hour review of around a hundred lines of heavily commented Python code by security specialists was not sufficient to discover three inserted bugs [23, Section 7]. However, we can use our logic to state and verify the fact that P cannot affect any `Cell`. Adapting the firewall formula, let $soleAccess(slot) \coloneqq \mathsf{rec}\ X . \langle slot?b \rangle X \& \neg\langle \neg slot?b \rangle X$, and $noAccess(slot) \coloneqq \mathsf{rec}\ X\ .\ \neg\langle slot?b \rangle X$; then we have

$$Cell(slot, state) \models soleAccess(slot)$$

which implies

$$\mathtt{new}(slot)\{Cell(slot, state)\} \dot{\approx} 0.$$

If we have that $P \models noAccess(slot)$, then

$$\forall\ state . \mathtt{new}(slot)\{P \mid Cell(slot, state)\} \dot{\approx} P.$$

For, by hypothesis, *slot* cannot be free in P, thus

$$\mathtt{new}(slot)\{P \mid Cell(slot, state)\}$$
$$\equiv$$
$$P \mid \mathtt{new}(slot)\{Cell(slot, state)\}$$

which means

$$P \mid \mathtt{new}(slot)\{Cell(slot, state)\} \dot{\approx} P \mid 0 \equiv P.$$

More generally, note that objects involve not just one cell, but many. By restricting those cells to slots that live in namespace $Slot \coloneqq \ulcorner \phi \urcorner$ for some ϕ we can scale our formula to separate objects from a given environment.

For now, we refine the basic idea to illustrate how to model deniability. The shape of the logical statement above is: if $Q \models \phi$ then when $P \models \psi$ we have that $\mathtt{new}(x_1, \ldots, x_N)\{P \mid Q\} \dot{\approx} S$ for some properties ϕ and ψ and some behavior, S. In light of theorem 4.1.1 we can, in principle, transform this into a statement of the form, "If $Q \models \phi$, then when $P \models \psi$ we have that $\mathtt{new}(x_1, \ldots, x_N)P \mid Q \models \omega$ for some properties ϕ, ψ, and ω." This is the shape of our encoding of deniability policies. The key insight here is to mediate between closed-world and open-world forms of deniability. In this open-world formulation we do not quantify over all potential contexts and attackers, but work just with those that can be relied upon to use the API, even in attacks.

The first element of this encoding we wish to draw attention to is that this provides just the emphasis on necessary rather than sufficient conditions. Moreover, notice that the form of the statement remains true to

the intent of deniability even if the logic is *more* discriminating than bisimulation—which is the case if we include the separation operator (as noted in [3]). In the presence of a logic with separation this shape turns out to be a form of rely-guarantee identified by Honda in [9] and, as Honda observes, can be internalized in the logic as the adjunct to separation. Thus, we include a form of rely-guarantee formulae, $\phi \rhd_{\vec{x}} \psi$, where $\vec{x} = \{x_1, \ldots, x_N\}$, with semantics $[\![\phi \rhd_{\vec{x}} \psi]\!](v) = \{P : \forall Q.Q \in [\![\phi]\!](v) \Rightarrow \texttt{new}(x_1, \ldots, x_N)\{P \mid Q\} \in [\![\psi]\!](v)\}$, as the logical form that mediates between closed-world and open-world formulations of deniability.

To apply this to the bank example amounts to recognizing that the `purse` is essentially a generalized *Cell*. There are two means to access the internal state: `deposit` and `decr`. The example, however, enjoys an additional level of indirection in that in the `ECMAScript` the corresponding functions these are scoped just to that purse and affect no other. Attempting to model this naively will quickly run into problems, e.g. as a parallel composition of purses will contend over serving balance update messages. This is where the higher order nature of the RHO-calculus makes it more suitable than the π-calculus for modeling these kinds of programs. In our translation a reference to `purse` $[\![\texttt{var}\ purse = \ldots]\!]$ is a *Cell* that holds the *code* of a process for a (generalized) *Cell* serving the `deposit` and `decr` messages. Updates through a purse first access and execute the behavior to serve balance update messages and then put the modified *behavior* (with the new balance) back in the *Cell* for the purse. Thus, to express the property that no one can affect the balance of a purse they don't have [6] we only have to reason about access through this one channel, *purse*, providing access to the outer *Cell*. The higher-order nature of the calculus does the rest of the state management for us. But, we have already accomplished that for *Cell*s, generically, in our previous reasoning.

6. CONCLUSIONS AND FUTURE WORK

The bulk of our work in modeling deniability policies as rely-guarantee formulae in a logic with separation was in setting up the calculi and logic. Once these are absorbed, the actual model is short and to the point. We take this as evidence that the approach is natural and commensurate with the expressiveness demands of deniability style policies.

In this connection it is important to note that a lot is already known about the complexity characteristics of spatial-behavioral logics. Interpreting ocap policy in this setting, therefore, has the further advantage that we are able to classify policy statements that are potentially too hard to check automatically, or even check at all. To wit, [4] note that the addition of the adjunct to separation is not a conservative extension!

In developing this approach we uncovered several situations where value-dependent policies were of interest or importance. This suggests that some sort of dependent types may be necessary for addressing more realistic examples.

Finally, source code for implementations of an interpreter for the RHO-calculus and a model-checker for namespace logic are available upon request and a more industrial scale version of the core concepts are used heavily in the `SpecialK` library available on `github` [17]. This library is already in use in industrial projects such as the Protunity Business Matching Network [10].

Acknowledgments.. The authors wish to thank Fred Fisher, Mark Miller and James Noble for thoughtful and stimulating conversation about policy, object capabilities and types. The first two authors' work on this topic is generously supported by Agent Technologies, Inc.

7. REFERENCES

[1] Samson Abramsky, *Proofs as processes*, Theor. Comput. Sci. **135** (1992), no. 1, 5–9.

[2] Allen L. Brown, Cosimo Laneve, and L. Gregory Meredith, *Piduce: A process calculus with native xml datatypes*, In Proc. of EPEW05/WS-F0M5, volume 3670 of Lect, Springer, 2005, pp. 18–34.

[3] Luís Caires, *Behavioral and spatial observations in a logic for the pi-calculus.*, FoSSaCS, 2004, pp. 72–89.

[4] Luís Caires and Etienne Lozes, *Elimination of quantifiers and undecidability in spatial logics for concurrency*, 2004.

[5] Horatiu Cirstea, Germain Faure, and Claude Kirchner, *A rho-calculus of explicit constraint application*, Proceedings of the 5th workshop on rewriting logic and applications, vol. 117 of Electronic Notes in Theoretical Computer Science, 2004.

[6] Sophia Drossopoulou and James Noble, *The need for capability policies*, Proceedings of the 15th Workshop on Formal Techniques for Java-like Programs (New York, NY, USA), FTfJP '13, ACM, 2013, pp. 6:1–6:7.

[7] Brendan Eich, `https://mail.mozilla.org/pipermail/es-discuss/2013-March/029080.html`, 2013.

[8] C. A. R. Hoare, *An axiomatic basis for computer programming*, Commun. ACM **12** (1969), no. 10, 576–580.

[9] Kohei Honda and Nobuko Yoshida, *A unified theory of program logics: an approach based on the π-calculus*, Proceedings of the 2008 international conference on Visions of Computer Science: BCS International Academic

Conference (Swinton, UK, UK), VoCS'08, British Computer Society, 2008, pp. 259–274.

[10] Wadood Ibrahim, Jassen Klassen, and Lucius G. Meredith, *Protunity business matching network*, `http://www.protunity.com`, Launched April 2013.

[11] Google Inc., *Html service: Caja sanitization*, `https://developers.google.com/apps-script/guides/html-service-caja`, 2013.

[12] Matej Košík, *A Contribution to Techniques for Building Dependable Operating Systems*, Ph.D. thesis, Slovak University of Technology in Bratislava, May 2011.

[13] Jean-Louis Krivine, *The curry-howard correspondence in set theory*, Proceedings of the Fifteenth Annual IEEE Symp. on Logic in Computer Science, LICS 2000 (Martin Abadi, ed.), IEEE Computer Society Press, June 2000.

[14] L. G. Meredith and Steve Bjorg, *Contracts and types*, Commun. ACM **46** (2003), no. 10, 41–47.

[15] L. Gregory Meredith and Matthias Radestock, *Namespace logic: A logic for a reflective higher-order calculus.*, in *TGC* [16], pp. 353–369.

[16] ———, *A reflective higher-order calculus.*, Electr. Notes Theor. Comput. Sci. **141** (2005), no. 5, 49–67.

[17] Lucius G. Meredith, *Specialk scala library*, `https://github.com/leithaus/SpecialK`, Launched April 2013.

[18] Robin Milner, *The polyadic π-calculus: A tutorial*, Logic and Algebra of Specification **Springer-Verlag** (1993).

[19] Martin Odersky, *Polarized name passing.*, FSTTCS (P. S. Thiagarajan, ed.), Lecture Notes in Computer Science, vol. 1026, Springer, 1995, pp. 324–337.

[20] Benjamin C. Pierce and David N. Turner, *Pict: A programming language based on the pi-calculus*, Proof, Language and Interaction: Essays in Honour of Robin Milner (Gordon Plotkin, Colin Stirling, and Mads Tofte, eds.), MIT Press, 2000, pp. 455–494.

[21] David Sangiorgi and David Walker, *The π-calculus: A theory of mobile processes*, Cambridge University Press, 2001.

[22] Davide Sangiorgi, *The bisimulation proof method: Enhancements and open problems*, Formal Methods for Open Object-Based Distributed Systems (Roberto Gorrieri and Heike Wehrheim, eds.), Lecture Notes in Computer Science, vol. 4037, Springer Berlin Heidelberg, 2006, pp. 18–19.

[23] Ka-Ping Yee, *Report on the pvote security review*, Tech. Report UCB/EECS-2007-136, EECS Department, University of California, Berkeley, Nov 2007.

8. APPENDIX: ρ-CALCULUS SUGAR

It is known that replication (and hence recursion) can be implemented in a higher-order process algebra [21]. This encoding provides a good example of calculation with the RHO-calculus.

$$D(x) \coloneqq x?(y) \Rightarrow (x!(*y) \mid *y)$$
$$\int_x P \coloneqq x!(D(x) \mid P) \mid D(x)$$

$$\begin{aligned}
&\int_x P \\
&= \\
&x!((x?(y) \Rightarrow (x!(*y) \mid *y)) \mid P) \mid x?(y) \Rightarrow (x!(*y) \mid *y) \\
&\rightarrow \\
&(x!(*y) \mid *y)\{@(x?(y) \Rightarrow (*y \mid x!(*y))) \mid P/y\} \\
&= \\
&x!(*@(x?(y) \Rightarrow (x!(*y) \mid *y)) \mid P) \\
&\quad \mid (x?(y) \Rightarrow (x!(*y) \mid *y)) \mid P \\
&\rightarrow \\
&\cdots \\
&\rightarrow^* \\
&P \mid P \mid \ldots
\end{aligned}$$

Of course, this encoding, as an implementation, runs away, unfolding $\int_x P$ eagerly. A lazier and more implementable replication operator, restricted to input-guarded processes, may be obtained as follows.

$$\int^x u?(v) \Rightarrow P \coloneqq x!(u?(v) \Rightarrow (D(x) \mid P)) \mid D(x)$$

It is worth noting that the combination of reference and dereference operators is essential to get computational completeness.

Armed with this we can consider a conservative extension of the calculus presented above.

$$\begin{aligned}
&P, Q ::= \ldots \\
&\quad \mid (\mathtt{def}\, X(y_1, \ldots, y_N) \Rightarrow P)(Q_1, \ldots, Q_N) \\
&\quad \mid X(Q_1, \ldots, Q_N) \\
&\quad \mid \mathtt{new}(x_1, \ldots, x_N) P
\end{aligned}$$

The notation for recursive definitions is slightly quirky in that it simultaneously defines and invokes the definition to some arguments – in order to make the re-

cursive form be a process expression, as opposed to a separate category of declaration. Comparing this with the examples above we have simply omitted the application of the recursive definitions to initial arguments in the interest of brevity.

$$(\texttt{def}X(y_1,\ldots,y_N) \Rightarrow P)(Q_1,\ldots,Q_N) \coloneqq \int^{n(X,P)} (n(X,P) \Rightarrow P') \mid n(X,P)!(Q_1,\ldots,Q_N)$$

where $P' \coloneqq P\{n(X,P)!(Q_1,\ldots,Q_N)/X(Q_1,\ldots,Q_N)\}$, i.e., the body of P in which every occurrence of

$X(Q_1,\ldots,Q_N)$

is replaced with $n(X,P)!(Q_1,\ldots,Q_N)$, and $n(X,P)$ generates a name from X guaranteed fresh in P.

Likewise, there are several translation of the π-calculus `new` operator into the RHO-calculus. Rather than take up space in this paper we simply refer the reader to [16], or better yet, invite them to come up with a definition to test their understanding of the calculus.

Additionally, we call out input-guarded only summations using the notation

$$\begin{array}{l}\texttt{match } \{\\ \quad x_1?(y_{1,1},\ldots,y_{1,N}) \Rightarrow P_1\\ \quad \ldots\\ \quad x_M?(y_{M,1},\ldots,y_{M,N'}) \Rightarrow P_M\\ \}\end{array}$$

instead of

$x_1?(y_{1,1},\ldots,y_{1,N}) \Rightarrow P_1 + \ldots + x_M?(y_{M,1},\ldots,y_{M,N'}) \Rightarrow P_M$

In a `match` context we avail ourselves of a pattern-matching syntax, such as $x?get(ret) \Rightarrow ret!(v)$ and $x!get(k)$. Again, encodings are quite standard, e.g. see [2].

Finally, we use $\{\ldots\}$ for grouping expressions and in more complex examples, omit $\mid$ in favor of line breaks for legibility. Thus, an expression for a classic race-condition $x?(y_1,\ldots,y_N) \Rightarrow P \mid x!(Q_1,\ldots,Q_N) \mid x?(z_1,\ldots,z_N) \Rightarrow R$ might be written

$$\begin{array}{l}\{\\ \quad x?(y_1,...,y_N) \Rightarrow P\\ \quad x!(Q_1,...,Q_N)\\ \quad x?(z_1,...,z_N) \Rightarrow R\\ \}\end{array}$$

9. APPENDIX: BISIMULATION

Having taken the notion of restriction out of the language, we carefully place it back into the notion of observation, and hence into the notion of program equality, *i.e.* bisimulation. That is, we parameterize the notion of barbed bisimulation by a set of names over which we are allowed to set the barbs. The motivation for this choice is really comparison with other calculi. The set of names of the RHO-calculus is *global*. It is impossible, in the grammar of processes, to guard terms from being placed into contexts that can potentially observe communication, so we provide a place for reasoning about such limitations on the scope of observation in the theory of bisimulation.

From an ocap perspective, having the ability to enumerate names is usually a sign that a language is not even memory-safe, let alone capability-safe. However, in Section 4.1.1 below, we show that we can use types to create namespaces and then statically prove which namespaces a process has access to.

DEFINITION 9.0.2. *An* observation relation, $\downarrow_{\mathcal{N}}$, *over a set of names,* $\mathcal{N}$*, is the smallest relation satisfying the rules below.*

$$\frac{y \in \mathcal{N},\ x \equiv_N y}{x!(*v)\downarrow_{\mathcal{N}} x} \qquad (\text{OUT-BARB})$$

$$\frac{P \downarrow_{\mathcal{N}} x \text{ or } Q \downarrow_{\mathcal{N}} x}{P \mid Q \downarrow_{\mathcal{N}} x} \qquad (\text{PAR-BARB})$$

We write $P \Downarrow_{\mathcal{N}} x$ *if there is* Q *such that* $P \Rightarrow Q$ *and* $Q \downarrow_{\mathcal{N}} x$.

Notice that $x?(y) \Rightarrow P$ has no barb. Indeed, in RHO-calculus as well as other asynchronous calculi, an observer has no direct means to detect if a sent message has been received or not.

DEFINITION 9.0.3. *An* $\mathcal{N}$-barbed bisimulation *over a set of names,* $\mathcal{N}$*, is a symmetric binary relation* $\mathcal{S}_{\mathcal{N}}$ *between agents such that* $P\ \mathcal{S}_{\mathcal{N}} Q$ *implies:*

1. *If* $P \to P'$ *then* $Q \Rightarrow Q'$ *and* $P'\ \mathcal{S}_{\mathcal{N}} Q'$.
2. *If* $P \downarrow_{\mathcal{N}} x$, *then* $Q \Downarrow_{\mathcal{N}} x$.

P *is* $\mathcal{N}$*-barbed bisimilar to* Q, *written* $P \dot{\approx}_{\mathcal{N}} Q$, *if* $P\ \mathcal{S}_{\mathcal{N}} Q$ *for some* $\mathcal{N}$*-barbed bisimulation* $\mathcal{S}_{\mathcal{N}}$.