Swapsies on the Internet

First Steps towards Reasoning about Risk and Trust in an Open World

Sophia Drossopoulou, James Noble, Mark Miller

Abstract

Contemporary open systems use components developed by many different parties, linked together dynamically in unforeseen constellations. Code needs to live up to strict *security specifications:* it has to ensure the correct functioning of its objects when they collaborate with external objects of unknown provenance, and it has to protect its objects' integrity even when external objects are malicious.

In this paper we propose specification concepts to model risk and trust in such open systems. We use these concepts to specify Miller, Van Cutsem, and Tulloh's escrow exchange example, and discuss the meaning of such a specification. We argue informally that the code satisfies its specification.

1. Introduction

Playground Swapsies Imagine you are a child swapping football stickers in your school playground [34]. Perhaps you've got three Zinedine Zidanes, but you really want a Wiremu Reid? If you hand over the stickers you've got to trade to a gorilla from year ten, how do you know he won't run off with them before he gives you the stickers you need?

Football sticker swapsies illustrates the two complementary forces of *trust* and *risk*. When you show some of the stickers you've got to someone else, you risk that they might run of with those stickers, or rip them up. Awareness of trust means you can manage the amount of risk you are willing to tolerate. You're quite likely to lend your sticker book to your best friend, but you'd probably be careful to take only stickers you're definitively willing to risk losing when you go to meet the gorilla in year ten.

Internet Swapsies Playground swapsies is just one example of an interaction between *mutually untrusting* parties in an *open system* — other examples include so-called dark Internet markets (like Silk Road and Evolution) and even larger trading systems like eBay when participants choose not to rely on protection systems like PayPal. The systems are composed of different components (i.e. objects) from a range of different providers. There is no central trusted authority, no effective recourse to a universal clearing house, or a government or supranational agency — teachers don't interfere, and the kids don't want them to interfere anyway. This is for both political and technical reasons: a single trusted component becomes a single point of failure for the system if it is compromised, and typically requires a centralised architecture, such as shared databases or trusted transaction services. Rather, systems are built from the bottom up based only on object capabilities; the only access control mechanism is whether one object has a reference to another object, and thus the capability to call its methods [21]. Such code does not explicitly state its trust dependencies, nor the risks involved in using it.

Escrow Agents As a motivating example, we use the Escrow Exchange [24], a trusted third party that manages exchanges of different goods (eg money and shares) between trading parties. In particular:

- The trading parties do not trust each other,
- The parties do not rely on the escrow to determine the other party's trustworthiness; instead they themselves can dynamically identify whether they consider another party as trustworthy.
- The escrow has to make the exchange transactionally: it has to abort it, if any of the parties turns out to be untrustworthy, or if there are insufficient goods.
- The escrow has to eliminate the risks arising by untrustworthy parties.

The escrow example is important because it gauges how robustly trust and risk are managed. Supporting this kind of exchange isn't sufficient to build a whole system, but it is necessary: a system that cannot support escrow exchange will not be fit for most purposes. Many common approaches to security cannot deal with escrow exchanges: Information flow systems detect when information leaks to an untrusted party; confinement systems guarantee that state is not read or leaked to other components; sandboxes ensure that com-

ponents cannot affect the world outside the sandbox: thus, they all would *prevent* any exchanges with untrusted parties.

In contrast, our approach is to reason explicitly about code's security properties and guarantees, as trusted and untrusted components interact in an open world. We aim to describe how components coöperate to gradually establish trust, and to delineate the risks involved in that coöperation.

Our Contributions We address the following questions:

- How can we talk about trust in specifications of open systems?
- How can we quantify the risk posed by potentially untrustworthy objects?
- How can we prove that program code meets the specifications for risk and trust?

To address trust, we propose an **obeys** predicate, which describes whether an object may be trusted to satisfy some given specification. This predicate is *hypothetical*, in the sense that there is no "trust-bit" to observe in the run-time configuration. Rather, it guarantees that in all future configurations, the object will satisfy all requirements made by the specification. Trust is often *conditional*, in the sense that trust can sometimes established only under the condition that some other object is trusted.

To address risk, we propose further hypothetical predicates $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$, which describe whether an object may affect or may access a certain property (by executing some method). To address the open nature of the systems, we make the meaning of such assertions parametric with *any* code that may be linked to the current system — thus, in order for the software to guarantee its specification, it must be written in a very robust manner, so that no further, malicious code can steal its secrets or break its integrity.

Findings We have developed a specification of the escrow system, and we informally argue that the code adheres to it.

We were surprised to find that the specification for the Escrow is weaker than originally anticipated in two significant aspects: the escrow *cannot guarantee* that a reported successful transaction implies a) that the participants were trustworthy, nor that b) the participants are exposed to no risk by an untrustworthy participant (but we were able to characterize the risk to with participants are exposed). We were even more surprised to realize that it is *impossible to write* an escrow which would give guarantees a) and b). All the more striking, considering that the third co-author is one of the original developers of the escrow example.

Paper Organization Section 2 introduces the concepts participating in the Escrow Exchange. The naive code is not robust enough, and the specification, given in the traditional style, is not descriptive enough. Section 3 introduces concepts to describe trust and risk; we use them to give full specifications, and argue the code that satisfies these specifications. Section 4 sketches salient points in our specification language. Section 5 is related work. Section 6 concludes.

Disclaimers Throughout this paper, we make the simplifying assumptions that no two different arguments to methods are aliases, that the program is executed sequentially, that we can quantify over the entire heap, that object's do not breach their own encapsulation, and that any underlying network is error-free. This allows us to keep the specifications short, and to concentrate on the questions of risk and trust. Aliasing, concurrency, quantification, confinement and network errors can be dealt with using known techniques, but doing so would not shed any further light on the questions addressed in this paper.

2. Purse and Escrow

In this section we will work our way though a first version of an escrow exchange implementation (developed from Noble and Drossopoulou [26]) as a running example of the trust and risk involved in system specifications and implementations. We begin with a "traditional" approach based on Hoare logic, where trust is assumed and risk is implicit, and then show how explicit representations of trust and risk let us specify components more precisely, and reason about those specifications more accurately.

2.1 Purses

The escrow exchange example is based upon a system for modelling resources proposed in [23]. The "mints" and "purses" (or alternatively "banks" and "accounts"), can model anything fungible, including currencies or commodities: a mint models a type of currency or commodity, and a purse models an amount of that currency or commodity. We can model both money and goods by purses, although we distinguish them dynamically by using purses from different mints for money and goods respectively.

Figure 1 shows a traditional, Hoare logic style specification of the Mints and Purses — it consists of two policies (`Pol_deposit_1` and `Pol_deposit_2`) describing the behaviour of the method deposit, and one policy (`Pol_sprout`) describing the behaviour of `sprout`. A `Mint` object acts as a token representing a particular currency or type of goods. Mints have no public methods, but can e.g. be tested for identity to help verify transactions between purses.

A `Purse` object represents a particular purse (account). Any purse `prs` knows its balance, and the the `Mint` it belongs to, represented by the terms `prs.balance` and `prs.mint` respectively. These may access fields of `prs` or call ghost methods. Money can be transferred between any two purses of the same mint through the method `deposit`. If the destination and source are purses (`dest`, `src`:`Purse`) from the same mint (predicate `SameMint`), and the source purse's balance covers the amount to be deposited, then the

```
specification Purse{

  policy Pol_deposit_1    //   1st case:
     dest,src:Purse ∧ SameMint(dest,src) ∧ 0 ≤amt≤src.balance
         { dest.deposit(src,amt) }
     res   ∧
     dest.balance=dest.balance_pre+amt  ∧ src.balance=src.balance_pre−amt  ∧
     ∀p:_pre Purse\{dest,src} p.balance=p.balance_pre

  policy Pol_deposit_2     // 2nd case:
     prs,src:Purse ∧ ¬( SameMint(dest,src) ∧ 0≤amt≤src.balance  )
         { dest.deposit(src,amt) }
     ¬res ∧ ∀p:_pre Purse. p.balance=p.balance_pre

  policy Pol_sprout
     p:Purse
         { p.sprout() }
     res:Purse ∧ SameMint(res,p) ∧ res.balance=0 ∧ ∀p:_pre Purse. p.balance=p.balance_pre
}

predicate SameMint(prs1,prs2) ≡ prs1:Purse ∧ prs2:Purse ∧ prs1.mint=prs2.mint
```

Figure 1. Specification of `Purse` — First Attempt

```
specification Escrow {

 policy Pol_deal_1    //   1st case:
    SameMint(buyerMoney,sellerMoney) ∧  SameMint(buyerGoods,sellerGoods) ∧ price, amt:ℕ  ∧
    buyerMoney.balance_pre ≥price ∧  sellerGoods.balance_pre ≥amt ∧
          { deal( sellerMoney, sellerGoods, buyerMoney, buyerGoods, price, amt) }
    res ∧
    buyerMoney.balance=buyerMoney.balance_pre−price ∧ sellerMoney.balance=sellerMoney.balance_pre+price ∧
    buyerGoods.balance=buyerGoods.balance_pre+amt ∧ sellerGoods.balance=sellerGoods.balance_pre−amt ∧
    ∀p:_pre Purse_pre \{sellerMoney,sellerGoods, buyerMoney,buyerGoods}: p.balance=p.balance._pre

 policy Pol_deal_2    //   2nd case:
     ¬( SameMint(buyerMoney,sellerMoney) ∧  SameMint(buyerGoods,sellerGoods) ∧
     buyerMoney.balance_pre ≥price ∧  sellerGoods.balance_pre ≥amt  )
          { deal( sellerMoney, sellerGoods, buyerMoney, buyerGoods, price, amt) }
     ¬res ∧
     ∀p:_pre Purse: p.balance=p.balance._pre
}
```

Figure 2. Specification of `Escrow` :: `deal` – First Attempt

amount is transferred to the destination purse without modifying any other purses, and the call returns `true` – c.f. policy `Pol_deposit_1`. If the two purses are not from the same mint, or the argument purse has insufficient funds, the transaction does not take place, all purses remain unaffected, and the call returns `false` – c.f. policy `Pol_deposit_2`.

A new purse can be created at any time by asking an existing purse to `sprout` — this returns a new, empty purse from the same mint – c.f. `Pol_sprout`. The new purse has a zero balance but can then be filled via `deposit`.

2.2 Specifying Swapsies

Figure 2 is our first attempt at specifying an escrow exchange deal in the traditional style. The `deal` method takes as arguments four `Purse` objects: `sellerMoney`, `sellerGoods`, `buyerMoney`, and `buyerGoods`, and two numbers: `price` and `amt`. The aim is to transfer `amt` goods from the `sellerGoods` purse into the `buyerGoods` purse, and to transfer `price` from the `buyerMoney` purse into the `sellerMoney` purse.

This exchange needs to be transactional: Either `amt` goods are exchanged for the `price`, or the whole exchange must be terminated. The exchange must be robust against error and fraud — whether a potentially innocent mistake, such as a buyer accidentally supplying a `buyersMoney` purse with insufficient funds to pay the `price`, or a seller that supplies some kind of impostor `sellersGoods` purse that appears to operate correctly but that never actually transfers goods away from the seller.

The specification consists of two policies: `Pol_deposit_1` promises that if the purses come from the same mints, and have sufficient funds (lines 4-5), then the result will be `true` (line 7), the transfer of the monies and the goods will take place (lines 8-9), and all other purses will remain unaffected (line 10). `Pol_deposit_2` promises, that if the purses do not come from the same mints, or have insufficient funds, then the result will be `false` and all purses will be unaffected.

2.3 Naïve Implementation

Figure 4 shows a very naïve attempt to implement the `Escrow`: the `deal` method just transfers the money from buyer to seller, and then transfer the goods from seller to buyer. Indeed, if it is called in good faith (under the assumptions that everything is trustworthy, and there are enough money and goods in the respective purses to complete the transfer) then `deal` satisfies `Pol_deal_1`.

```
1 method deal(                  // returns boolean
2        sellerMoney, sellerGoods,
3        buyerMoney, buyerGoods,
4        price, amt)
5   {
6    sellerMoney.deposit(price,buyerMoney)
7    return (buyerGoods.deposit(amt,sellerGoods))
8   }
```

Figure 3. Naïve Deal

```
1  method dealV1(
2         sellerMoney, sellerGoods,
3         buyerMoney, buyerGoods,
4         price, amt)   // price and amount
5  {
6    // make temporary money Purse
7    escrowMoney = sellerMoney.sprout
8    // make temporary goods Purse
9    escrowGoods = buyerGoods.sprout
10
11   res = escrowMoney.deposit(price,buyerMoney)
12   if (!res) then
13        // insufficient money in buyerMoney
14        // or different money mints
15        { return false }
16
17   // sufficient money, same mints
18   // price transferred to escrowMoney
19   res = escrowGoods.deposit(amt, sellerGoods)
20   if (!res) then
21        // insufficient goods in sellerGoods
22        // or different goods mints
23        {  // undo the goods transaction
24           buyerMoney.deposit(price,escrowMoney)
25           return false }
26
27   // price in escrowMoney, amt in escrowGoods
28   // now complete the transaction
29   buyerMoney.deposit(price,escrowMoney)
30   sellerGoods.deposit(amt,escrowGoods)
31 }
```

Figure 4. First attempt at escrow deal

Unfortunately, `deal` does not satisfy `Pol_deal_2`: If the buyer has enough money but the seller does not have the goods to sell, then `deposit` on line 6 will successfully pay the buyer's money to the seller, but the second `deposit` on line 7 will fail to give the goods to the `buyer`.

2.4 Swapping via an Escrow Purse

The accepted solution to these problems is to use a pair of private *escrow* purses, one for on each side of the transaction (money and goods). Rather that swapping money and goods between buyer's and seller's purses in one go, the buyer's money and seller's goods are moved first into escrow purses, and then from the escrow purses into the final destinations. In this way, we only complete the second half of the transaction when we are sure enough money and goods are securely in the escrow purses. If the transaction needs to be abandoned halfway through, we can return the buyer's money from the escrow purse without any reference to the seller.

The `deal` method in Figure 4 shows such an implementation. This code is based on [24] and [26]. First, two escrow purses (`escrowMoney` and `escrowGoods`) are sprouted from the inputs — lines 6–9. The escrow purses are newly created within the method, and so cannot be manipulated by the buyer or seller.

Second, we attempt to escrow the buyer's money by transferring it from the `buyerMoney` purse into the new `escrowMoney` purse — line 11. According to the specification (Fig. 1), if this request returns true, then the money will have been transferred and both purses must be from the same mint. If the request fails we abort the transaction.

Third, we attempt to escrow the seller's goods – line 19, again by depositing them into the escrow purse. If we are unsuccessful, we again abort the transaction, after we have returned that money to the buyer – lines 24 and 25.

At this point (line 27) the `deal` method should have sole access to sufficient money and goods in the escrow purses. The method completes the transaction by transferring the escrowed money and goods into the respective destination purses – lines 29 and 30. Thanks to the escrow purses, these transfers should not fail so this code should meet the Figure 2's specification. If only the truth were that simple.

2.5 The failure of `dealV1`

The `dealV1` method in Figure 4 does not satisfy the `Escrow` specification in Figure 2 — in fact, in an open system, it *cannot*. The critical problems are assumptions about trust: both the code and the specification implicitly trust the purse objects with which they interact. Considering both the `Purse` and `Escrow` specifications: what happens if a purse or escrow is asked to interact with an untrustworthy purse? How much risk is involved: just the potentially untrustworthy

purse? That purse plus any other purse it knows about, or interacts with (e.g. both are passed into the same method)? Any purse (or indeed any object) anywhere in the system?

Classical specifications like Figures 1 and 2 have no notion of the risks involved when an object that does not meet its specification. All bets are off: the world ends.

Just because we can't write specifications doesn't mean that we can't write programs. Unfortunately, the code in Figure 4 is in no better shape than the specification. Imagine if `sellerMoney` was a malicious, untrustworthy object. At line 7, the `sprout` call could itself return a malicious object, which would then be stored in `escrowMoney`. Then at line 11, during execution of `escrowMoney.deposit(price, buyerMoney)` the malicious `escrowMoney` purse could steal all the money out of `buyerMoney` purse, and still return `false`. As a result, the seller would lose all their money, and receive no goods! Even if the seller was more cautious, and themselves sprouted a special temporary purse with a balance of exactly `price` to pass in as `sellerMoney`, they would still lose all this money without any recompense.

Perhaps there is something else we could do — a `trusted` method on every object, say, that returns `true` if the object is trusted, and `false` otherwise? The problem, of course, is that an object that is untrustworthy is, well, untrustworthy: we cannot expect a `trusted` method ever to return `false`. This leads to our definition of trust: trust is *hypothetical*, and in relation to some specification.

3. Specifying Trust and Risk Explicitly

The key claim of this paper is that we need specifications that let us talk about trust and risk explicitly. In this section, we begin by informally introducing three novel specification language constructs: **obeys** to model trust, and $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$ to model risk. We then revisit the specifications from the previous section using these constructs, showing how they can be used to specify the purse and escrow examples, and we argue informally that a revised `escrow` method can in fact meet revised specifications.

3.1 Modelling Trust: obeys

To model trust, we introduce a special predicate, "**obeys**", of the form o **obeys** $Spec$ which we interpret to mean that an object o can be trusted to adhere to the specification $Spec$.

Thus, specifications consist of sets of policies, and are predicates over objects. Any object which adheres to the specification may be safely assumed to satisfy all the policies in the specification.

Because we generally can't be sure that an object — especially one supplied from elsewhere in an open system — can actually be trusted to obey a particular specification, our reasoning and specifications tend to be *hypothetical*: analysing the same piece of code several times under different trust assumptions — i.e. assuming that particular objects may or may not be trusted to obey particular specifications.

Thus, *if* object `o` can be trusted to obey specification `Spec`, and `Spec` had a policy describing the behaviour of some method `m`, then we may expect the method call `o.m`$(...)$ to behave according to that policy - otherwise, all bets are off. This also leads to chains of hypothetical reasoning, as every method request on an object adds the assumption that the receiver obeys the specification we hope it does.

3.2 Modelling Risk: $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$

To model risk, we introduce predicates $\mathcal{M}ay\mathcal{A}ccess$ and $\mathcal{M}ay\mathcal{A}ffect$, which express whether an object may read or may affect another object or property. We will write "$\mathcal{M}ay\mathcal{A}ffect$`(o,p)`" to mean that it is possible that some method invocation on o would affect the object or property p. Similarly, we will write "$\mathcal{M}ay\mathcal{A}ccess$`(o,p)`" to mean that it is possible that the code in object o could potentially gain a capability to access to p — that is, a reference to p. In practice, $\mathcal{M}ay\mathcal{A}ccess$`(o,p)` means that p is in the transitive closure of the points-to relation on the heap starting from o, including both public and private references.

3.3 Valid Purses - the Policies of `Purse`

We will now revisit the specifications for `Purse` and `Escrow` and give their policies using the new features introduced in the previous section. Once again, we begin by considering the specification of purses, before going on to the specification and then implementation of the escrow itself. Figure 5 revisits the purse specification policies from Figure 1, making the risk and trust explicit.

The specification in Figure 5 consists of five policies. It is parametric over a class `Prs`; thus, any class adheres to the specification, if it satisfies all five policies. In that sense, the specification is a predicate over classes.

The cases `Pol_deposit_1` and `Pol_deposit_2` taken together distinguish between a successful and an unsuccessful deposit, signalled by returning `true` or `false` respectively. In the first case, i.e. `Pol_deposit_1` where the result is `true`, argument `src` must have been a valid purse ("'src **obeys** ValidPurse") from the same mint as the receiver, and `src` must have sufficient balance. In the second case, i.e. `Pol_deposit_2` where the result is `false`, either `src` was not a valid purse, or not the same mint as the receiver, or had insufficient funds. Moreover, in the first case, the transaction will happen, but all other purses will be unmodified, whereas in the second case no purses will be modified.

The key difference between the `ValidPurse` specification and the earlier `Purse` specification is that `ValidPurse` uses **obeys** clauses to reason about trust explicitly. For the reasons described above, `ValidPurse` cannot make absolute statements about trust, but can support relative, hypothetical statements. Consider the request

```
res=dest.deposit(src,amt).
```

```
specification ValidPurse(Prs){

  policy Pol_deposit_1     //   1st case:
      dest:Prs ∧ amt∈ℚ
         { dest.deposit(src,amt) }
      res → (
              // TRUST
              src obeys_pre ValidPurse ∧ SameMint(dest,src)_pre
              // FUNCTIONAL SPECIFICATION
              ∧ 0≤amt≤src.balance_pre ∧
              dest.balance=dest.balance_pre+amt ∧ src.balance=src.balance_pre−amt  ∧
              //RISK
              ∀ p.(p obeys ValidPurse ∧ p≠{dest,src} → p.balance=p.balance_pre)  )

  policy Pol_deposit_2     //   2nd case:
      dest:Prs ∧ amt∈ℚ
         { dest.deposit(src,amt) }
       ¬res → (
              // TRUST and FUNCTIONAL SPECIFICATION
              ¬( src obeys_pre ValidPurse ∧ SameMint(dest,src)_pre ∧ 0≤amt≤src.balance_pre) ∧
              // RISK
              ∀ p. (p obeys_pre ValidPurse → p.balance=p.balance_pre)   )

  policy Pol_sprout
      dest:Prs
         { dest.sprout() }
      // TRUST
      res obeys ValidPurse ∧  SameMint(dest,res)_pre ∧
      // FUNCTIONAL SPECIFICATION
      res.balance=0 ∧
      // RISK
      ∀p.(p obeys_pre ValidPurse →  p.balance=p.balance_pre)

  policy Pol_mint_constant
       p:Prs
         { any_code }
       p.mint=p.mint_pre

  policy Pol_protect_balance
      // RISK
     o:Object ∧ p:Prs ∧ MayAffect(o,p.balance) → MayAccess(o,p)
}

predicate SameMint(prs1,prs2) ≡  prs1.mint=prs2.mint
```

Figure 5. Specification of `ValidPurse`

If the destination purse accepts the deposit, then we would like to deduce that it has been able to retrieve the funds from the source purse. So, we would like to assert the absolute statement that

`res → src obeys ValidPurse`

But the `ValidPurse` specification only applies when the receiver `dest` is trustworthy: we can get only as far as the relative conclusion

`res ∧ dest obeys ValidPurse → src obeys ValidPurse`

meaning that, if the `deposit` method returns true, then we can trust `src` if we were willing to trust `dest`.

`Pol_sprout`, the third policy, is basically the same as the earlier version in Figure 1, except that the postcondition now is weaker, as it only promises that the result is a trusted purse, without guaranteeing which class it belongs to.

The fourth policy, `Pol_mint_constant`, guarantees that execution of *any* code, will not change the mint of a valid purse, even if that code belongs to objects of a class different than `Prs`. This is another key ingredient of our specification language: the possibility to require that any unknown code has to guarantee some properties.

Finally, the fifth policy, `Pol_protect_balance`, delimits the risk involved with the purses. This policy guarantees that a valid purse `p`'s balance can only be changed ("`MayAffect(o,p.balance)`") by some object `o` that may access that purse ("`MayAccess(o,p)`").

In [12] a statically typed implementation of purses and mints [13] is proven to meet such a specification. This can also be done in the dynamically typed setting.

3.4 Establishing Mutual Trust

The key to successful swapsies — or any other trading — is establishing just enough mutual trust for just long enough for the two parties to be able to complete the transaction. The previous section argued that a call like:

```
res1=dest.deposit(src,amt)
```

lets us conclude that

```
res1 ∧ dest obeys ValidPurse → src obeys ValidPurse
```

So, if an amount is deposited successfully into a trustworthy destination `ValidPurse`, that purse vouches that the `src` is itself trustworthy. This is essentially what it means to accept a trustworthy deposit, and accepting trustworthy deposits is key to being a trustworthy purse — to quote Miller et al. [23]: "*A reported successful deposit can be trusted as much as one trusts the purse one is depositing into*". This trust is just one way: from the destination to the source purse.

Noble and Drossopoulou [26] offer a key insight: we can establish mutual trust between two purses by attempting a second deposit in the reverse direction:

```
res2=src.deposit(dest,amt)
```

which gives

```
res2 ∧ src obeys ValidPurse → dest obeys ValidPurse
```

Reasoning conditionally, on a path where `res1 ∧ res2` are true, we'll have established mutual trust:

```
dest obeys ValidPurse ⟷ src obeys ValidPurse
```

As with much of our reasoning, this is both conditional and hypothetical: at a particular code point, when two `deposit` requests have succeeded (or rather, that they have both *reported* success) then we can conclude that either both are trust worthy, or both are untrustworthy: we have only *hypothetical* knowledge of the **obeys** predicate.

This is a variation on the classic problem of "trusting trust". We rely on the destination purse to validate source purse, and vice versa. But if neither purse acts in good faith, if neither purse is trustworthy, there is no sure way we can tell the difference between two trustworthy purses mutually trusting each other, and two untrustworthy purses both lying.

3.5 Escrow with Explicit Mutual Trust

Two way deposit calls are sufficient to establish mutual trust, but come with risks. For example, as part of validating that a buyer's money purse mutually trusts the seller's money purse, we must pass the buyer's purse as argument in a `deposit` call to the seller's money purse, e.g.

```
sellerMoney.deposit(buyerMoney,0)
```

If the seller's purse is not in fact trustworthy, then it can take this opportunity to steal all the money in the buyer's purse before the transaction officially starts, even if the `amt` that is supposed to be deposited is `0`.

We can minimise this risk by careful use of escrow purses. Rather than mutually validating buyers and sellers directly, we can create an escrow purse on the destination side of the transaction (the seller's money and the buyer's goods) and then mutually validate the buyer's and sellers actual purses against the escrow — resulting in a chain of mutual trust between the destination purse and the escrow purse, and the escrow purse and the source purse. This allows us to hypothesise that the source and destination purses are mutually trusting before we start on the transaction proper.

```
1  method dealV2( // returns Boolean
2         sellerMoney, sellerGoods,
3         buyerMoney, buyerGoods,
4         price, amt
5  ) {
6    //setup and validate Money purses
7    escrowMoney = sellerMoney.sprout
8    res=escrowMoney.deposit(0,sellerMoney)
9    if (!res) then {return false}
10   res = buyerMoney.deposit(0,escrowMoney)
11   if (!res) then {return false}
12   res = escrowMoney.deposit(0,buyerMoney)
13   if (!res) then {return false}
14
15   //setup and validate Goods purses
16   escrowGoods = buyerGoods.sprout
17   res=escrowGoods.deposit(0,buyerGoods)
18   if (!res) then {return false}
19   res = sellerGoods.deposit(0,escrowGoods)
20   if (!res) then {return false}
21   res = escrowGoods.deposit(0,sellerGoods)
22   if (!res) then {return false}
23
24   res = escrowMoney.deposit(price,buyerMoney)
25   if (!res) then {return false}
26   res = escrowGoods.deposit(amt,sellerGoods)
27   if (!res) then {
28     buyerMoney.deposit(price,escrowMoney)
29     return false}
30
31   sellerMoney.deposit(price,escrowMoney)
32   buyerGoods.deposit(amt,escrowGoods)
33
34   return true
35 }
```

Figure 7. Revised `Escrow` method

The resulting escrow method is in Figure 7. Line 7 creates a `escrowMoney` purse and then lines 8–13 hypothetically establish mutual trust between the `escrowMoney`, `sellerMoney`, and `buyerMoney` purses. We don't need the `sellerMoney` purse to validate the `escrowMoney` purse explicitly (`sellerMoney.deposit(escrowMoney,0)`) because the `sprout` method specification says sprouted purses can trusted as much as their parent purses (`res` **obeys** `ValidPurse`). If any of these `deposit` request fail, we abort. Lines 16–22 do exactly the same, but for goods purses rather than money purses. Finally, lines 24–34 carry out the escrow exchange itself, in exactly the same manner as lines 11–34 of the first escrow specification in Figure 2.

3.6 Specifying the Mutual Trust Escrow

Figure 6 shows a specification for the revised escrow deal method from Figure 7. Whereas our original specification in

```
1  specification ValidEscrow(Escr) {
2   policy Pol_deal_1    //   1st case:
3      e:Escr ∧ price,amt∈ ℚ ∧ price,amt>0
4         { e.deal( sellerMoney, sellerGoods, buyerMoney, buyerGoods, price, amt) }
5      res ∧ BadPartPurses=∅  → (
6        // FUNCTIONAL SPECIFICATION
7        buyerMoney.balance=buyerMoney.balance_pre−price ∧sellerMoney.balance=sellerMoney.balance_pre+price∧
8        buyerGoods.balance=buyerGoods.balance_pre+amt ∧ sellerGoods.balance=sellerGoods.balance_pre−amt ∧
9        // RISK
10       ∀p:_preOtherPurses. p.balance=p.balance._pre )
11
12  policy Pol_deal_2    //   2nd case:
13     e:Escr ∧ price,amt∈ ℚ ∧ price,amt>0
14        { e.deal( sellerMoney, sellerGoods, buyerMoney, buyerGoods, price, amt) }
15     ¬res ∧ BadPartPurses=∅  → (
16        // FUNCTIONAL SPECIFICATION
17        ¬( SameMint(buyerMoney,sellerMoney) ∧  SameMint(buyerGoods,sellerGoods) ∧
18          buyerMoney.balance_pre ≥ price ∧ sellerGoods.balance_pre ≥ amt ) ∧
19       // RISK
20       ∀p:_preGoodPurses. p.balance=p.balance._pre   )
21
22  policy Pol_deal_3    //   3rd case:
23     e:Escr ∧ price,amt∈ ℚ ∧ price,amt>0
24        { e.deal( sellerMoney, sellerGoods, buyerMoney, buyerGoods, price, amt) }
25     ¬res ∧ BadPartPurses≠∅  →
26        //RISK
27        ∀p:_preGoodPurses. ( p.balance=p.balance._pre ∨ ∃ bp∈BadPartPurses_pre.MayAccess(bp,p)_pre )
28
29  policy Pol_deal_4    //   4th case:
30     e:Escr ∧ price,amt∈ ℚ ∧ price,amt>0
31        { e.deal( sellerMoney, sellerGoods, buyerMoney, buyerGoods, price, amt) }
32     res ∧ BadPartPurses≠ ∅    →  (
33       // TRUST
34       buyerMoney obeys PurseSpec ⟷ sellerMoney obeys PurseSpec  ∧
35       buyerGoods obeys PurseSpec ⟷ sellerGoods obeys PurseSpec  ∧
36       //RISK
37       ∀p:_preOtherPurses. ( p.balance=p.balance._pre ∨ ∃ bp∈ BadPartPurses_pre.MayAccess(bp,p)_pre )   )
38
39 where // auxiliary definitions
40   GoodPurses = { p | p obeys _preValidPurse }
41   PartPurses = { sellerMoney, sellerGoods,  buyerMoney, buyerGoods }
42   BadPartPurses = PartPurses \ GoodPurses
43   OtherPurses = GoodPurses \ PartPurses
44 }
```

Figure 6. `ValidEscrow` specification

Figure 2 consisted of two cases based on the value of the result, our revised `ValidEscrow` specification distinguishes four cases, based on the value of the result, *as well as* the trustworthiness of the participants.

We use the auxiliary definitions from lines 40-44: The set `PartPurses` describes the four "participant purses" passed as arguments. `BadPartPurses` contains the untrustworthy participant purses. `GoodPurses` is all trustworthy purses in the system that do confirm to the `ValidPurse` specification, and `OtherPurses` is the trustworthy purses that do *not* participate in this particular deal.

We now discuss the four cases of the policy

1st case: The result is `true` and all participant purses are trustworthy. Then, the goods and money purses were from the same mints respectively, and there was sufficient money in the buyer's purse and sufficient goods in the sellers purse. In this case, everything is fine, so we can play swapsies: `price` will have been transferred from the buyer's to the seller's money purse, and `amt` will have been transferred from the seller's to the buyer's goods purse. No risk arises: no other purses' balance will change (whether passed in to the method or not).

2nd case: The result is `false` and all participant purses are trustworthy. Then one or more of the functional correctness conditions are not satisfied: purses' mints did not match appropriately, or input purses did not have sufficient balance. And no risk to any purses.

3rd case: The result is `false` and some participant purse is untrustworthy. In this case, no trustworthy purses' balances have been changed — unless they were already

accessible by an untrustworthy purse passed in to the method.

4^{th} case: The result is `true` and some participant purse is untrustworthy. In this case, at least two matching participant purses are untrustworthy. That is, the pair of matching purses have coöperated to suborn the escrow *and we cannot tell*. Therefore, either both money purses are untrustworthy,

`buyerMoney` **obeys** `PurseSpec` $\longleftrightarrow$
`sellerMoney` **obeys** `PurseSpec`

or both goods purses are untrustworthy:

`buyerGoods` **obeys** `PurseSpec` $\longleftrightarrow$
`sellerGoods` **obeys** `PurseSpec`

or all four are bad.

The risk is that an uninvolved trustworthy purse's balance cannot be changed unless it was previously accessible from a bad purse.

The first and second case correspond to the traditional `Escrow` specification in Figure 2, because traditional specifications assume all objects are trustworthy.

Discussion The 3^{rd} and 4^{th} case represent more of a risk than we would like: ideally (as in the 2^{nd} case) we'd hope nothing should have changed. But an escrow method cannot undo a system that is already suborned — if one of the participant purses is already benefiting from a security breach, passing that purse in to this method gives it an opportunity to exercise that breach. On the other hand, the risk is contained: in the absence of preëxisting breaches, this method cannot make things worse.

The 4^{th} case does not prevent trustworthy participant purses from being modified, to cater e.g., for the possibility that the two money purses are trustworthy, while the two goods purses are not, in which case the money transaction will take place as expected, while all bets are off about the goods transaction. We can give the stronger guarantee for the 3^{rd} case, because by the time the escrow starts making non-0 transactions (line 24 onwards in Fig. 7) it has established that the purses in each pair are either both trustworthy or both not trustworthy.

Most importantly, and perhaps surprisingly. the return value of the method, `res`, does *not* indicate whether the participants were trustworthy or not. Namely, a `true` result may be obtained in the 1^{st} case (all purses trustworthy) as well as the 4^{th} (some purses are untrustworthy). The return value indicates *only* whether the escrow attempted to complete the transaction (returning `true`) or abort (returning `false`). This came indeed as a surprise to the original developers of the `deal` method. As with much of our reasoning around trust, this leads to yet more conditional reasoning, which must be interpreted hypothetically.

Nevertheless, the return value does communicate a valuable guarantee to an honest participant, whose money and goods purses are both trustworthy: If `deal` returns `true`, then the swap has taken place. Furthermore if it returns `false`, the swap has not taken place and with *no* more risk to the honest purses than those that existed before the call.

This `ValidEscrow` specification also gives a guarantee to other purse objects, who do *not* necessarily take part in the deal. Namely, if the participants had no prior access to these purses, then even if those participants were dishonest, the purses' balance can never be affected.

4. Formal Definitions

In this section, we sketch the most salient features of the formal underpinnings of our system, and leave the full exposition to further work.

Underlying Programming and Specification Language We assume a small object oriented language, $\mathcal{FOCAL}$, which supports classes, fields and methods. $\mathcal{FOCAL}$ is memory-safe: it does not allow addresses to be forged, or non-existent methods or fields to be called, read or written. $\mathcal{FOCAL}$ is dynamically typed: it does not check that the arguments to a method call or a field write are of the appropriate type either statically or dynamically: in this sense, $\mathcal{FOCAL}$ is inspired by JavaScript, E, and Dart's unchecked mode.

$\mathcal{FOCAL}$ supports modules, M, which are mappings from class identifiers to class definitions. The module linking operator $*$ combines these definitions, provided that the modules have separate domains, and performs no other checks. This reflects the open world setting, where code of different provenance interoperates with one another without a central authority checking their compatibility.

$\mathcal{FOCAL}$ dynamically enforces `private` fields and methods. Accessing or calling private fields or methods is only allowed from method bodies of the same class; if not, the exception `error` is thrown. Incorporating private fields directly allows us to express some notion of secrets shared across objects, without more complex features such as nested lexical scopes.

The operational semantics of $\mathcal{FOCAL}$ has the shape

$$\mathtt{M},\ \kappa,\ e\ \leadsto\ \kappa',\ r,$$

where `M` is a module containing all class declarations used, κ, κ' are runtime configurations, *e* is an expression, and *r* is a result, i.e. an address, or the exception `error`.

Paths start with the receiver **this**, or the formal parameter **x**, followed by a sequence of field identifiers (`f`). We define $\lfloor p \rfloor_\kappa$, the lookup of a path p in a context κ in the expected way, where we read the receiver or argument from the frame, and follow the values of the fields in the heap.

We assume an underlying specification language of state dependent resp. state independent predicates Q-s, resp. P-s, and state dependent resp. state independent functions, f-s, resp. g-s. For example, `OrderedList` resp. $\leq$ are state dependent resp. state independent predicates, while `length`,

resp. + are state dependent , resp. state independent functions. We require the underlying specification language to support both one, and two state assertions.

Definition 1 (Two-state assertion, basic)**.**

$$\begin{array}{lcl} P & ::= & Q(arg_t^*) \mid R(arg^*)_t \mid P \wedge P \mid P \vee P \mid \\ & & \texttt{true} \mid \texttt{false} \\ arg & ::= & f(arg_t^*) \mid g(arg^*)_t \mid p \mid val \\ t & ::= & \mathit{pre} \mid \mathit{post} \mid \epsilon \end{array}$$

We expect validity of one resp. two state assertions to have the shape $M, \kappa \vdash P$, resp, $M, \kappa, \kappa' \vdash P$. E.g., if $\lfloor \texttt{x.balance} \rfloor_\kappa = 4$, and $\lfloor \texttt{x.balance} \rfloor_{\kappa'} = 14$, we expect $\texttt{M}, \kappa \models \texttt{x.balance} < 10$, and $\texttt{M}, \kappa, \kappa' \models \texttt{x.balance}_{post} = \texttt{x.balance}_{pre} + 10$.

Hypothetical Access and Affect We expand the specification language with the special predicates *MayAffect*, and *MayAccess* which we use to model risk.

Definition 2 (*MayAffect* and *MayAccess*)**.**

$$\begin{array}{lcl} R & ::= & \textit{...as in def. 1 ...} \mid \\ & & \mathit{MayAffect}(p, p') \mid \\ & & \mathit{MayAccess}(p, p') \mid \end{array}$$

We define the validity of these assertions as follows:

- $\texttt{M}, \kappa, \kappa' \models \mathit{MayAffect}(\texttt{p}, \texttt{p}')_t$ iff
  $\exists$public method $\texttt{m}$, paths, $\bar{\texttt{p}}$.
  $M, \kappa'', \texttt{p.m}(\bar{\texttt{p}}) \rightsquigarrow \_, \kappa'''$ and $\lfloor \texttt{p}' \rfloor_{\kappa''} \neq \lfloor \texttt{p}' \rfloor_{\kappa'''}$.
- $\texttt{M}, \kappa, \kappa' \models \mathit{MayAccess}(\texttt{p}, \texttt{p}')_t$ iff
  $\exists$fields $\texttt{f}_1 ... \texttt{f}_n$. $\lfloor \texttt{p.f}_1 ... \texttt{f}_n \rfloor_{\kappa''} = \lfloor \texttt{p}' \rfloor_{\kappa''}$.

In both cases above, if *t*=pre, then $\kappa'' = \kappa$, else $\kappa'' = \kappa'$.

For example, assume a module M_1, which defined classes `Mint` and `Purse`, and where $M_1(\texttt{Mint})$ has no fields or methods, while $M_1(\texttt{Purse})$ has public fields `mint` and `balance` where balance is boxed, and methods `deposit` and `sprout` which directly transferred the moneys, and created the new `Purse`. Assume also another module M_2, where $M_2(\texttt{Mint})$ has a field `accounts` which is a mapping from `Purses` to serial numbers, and another public field `balances` which is a mapping from the serial numbers to `float`-s, which represent the balance. Then, in any κ_1, where $p :_\kappa \texttt{Purse}$, and $o :_\kappa \texttt{Object}$:

$M_1, \kappa_1 \models \mathit{MayAffect}(\texttt{p}, \texttt{p.balance})$,
$M_2, \kappa_1 \not\models \mathit{MayAffect}(\texttt{p}, \texttt{p.balance})$,
$M_1, \kappa_1 \models \mathit{MayAffect}(\texttt{o}, \texttt{p.balance}) \rightarrow \mathit{MayAccess}(\texttt{o}, \texttt{p})$
$M_2, \kappa_1 \not\models \mathit{MayAffect}(\texttt{o}, \texttt{p.balance}) \rightarrow \mathit{MayAccess}(\texttt{o}, \texttt{p})$

Arising Runtime Configurations To give meaning to our policies, it is essential to examine only those runtime configurations (*i.e.* context and code pairs) which may arise through the execution of the given modules.

We therefore define $\mathcal{A}rising(\texttt{M})$ as the set of runtime configurations which may be reached during execution of some initial configuration (κ_0,$expr_0$). We call initial configurations those with minimal heap, expression $expr_0$. The set $\mathcal{R}each(\texttt{M}, \kappa, \texttt{e})$ collects all configurations at the start of any method call during execution of $\kappa, \texttt{e}$ - as in visible states.

Definition 3.

$$\begin{array}{lcl} \mathcal{A}rising & : & Program \longrightarrow \mathcal{P}(Context \times Expr) \\ \mathcal{A}rising(\texttt{M}) & = & \bigcup_{(\kappa,\texttt{e}) \in \mathcal{I}nit(\texttt{M})} \mathcal{R}each(\texttt{M}, \kappa, \texttt{e}) \end{array}$$

Policies and Specifications We define adherence to policy, of the form $M \models Policy$, and we require that *any* module M' linked to M will exhibit behaviour which respects the requirements of P. For the latter, we only inspect the behaviour exhibited at configurations arising from $M * M'$, rather than any well-formed configuration.

Policies have one of the three following forms: invariants of the form P, which require that some property holds at all visible states of a program; or $P\ \texttt{code}\ P'$ where P must be a one-state assertion, and which require that execution of `code` in any state which satisfies P will lead to a state which satisfied P'; or $P\ \texttt{any\_code}\ P'$ which, like two state invariants require that execution of *any* code in a state which satisfies P will lead to a state which satisfies P'.

Definition 4 (Policies)**.**

$$Policy \quad ::= \quad P \mid P\ \{\texttt{code}\}\ P \mid P\ \{\texttt{any\_code}\}\ P$$

Adherence to policy::

- $M \models P$ iff
  $\forall M'. \forall (\kappa, \_) \in \mathcal{A}rising(M * M').$
  $M, \kappa \models P$
- $M \models P\ \{\texttt{code}\}\ P'$ iff
  $\forall M'. \forall (\kappa, \_) \in \mathcal{A}rising(M * M').$
  $(\ M, \kappa \models P \ \wedge \ M, \kappa, \texttt{code} \rightsquigarrow \texttt{res}, \kappa'$
  $\longrightarrow$
  $M, \kappa, \kappa' \models P' \ )$
- $M \models P\ \{\texttt{any\_code}\}\ P'$ iff
  $\forall M'. \forall (\kappa, \texttt{code}) \in \mathcal{A}rising(M * M').$
  $(\ M, \kappa \models P \ \wedge \ M, \kappa, \texttt{code} \rightsquigarrow \texttt{res}, \kappa'$
  $\longrightarrow$
  $M, \kappa, \kappa' \models P' \ )$

We therefor see that $M \models Policy$ not only ensures that execution of M will guarantee $Policy$, but also, that the code of M has been written in such a robust manner that any other code linked with M cannot break $Policy$.

Thus, even though:

$M_1, \kappa_1 \models \mathit{MayAffect}(\texttt{o}, \texttt{p.balance}) \rightarrow \mathit{MayAccess}(\texttt{o}, \texttt{p})$.

the module M_1 does not adhere to that policy, i.e.:

$M_1 \not\models \mathit{MayAffect}(\texttt{o}, \texttt{p.balance}) \rightarrow \mathit{MayAccess}(\texttt{o}, \texttt{p})$.

This is so, because we can always link a further module M', which has code which, at some point, accesses the boxed `balance` and stores it in some further object, and then, later on, modifies the `balance` of `p` without necessarily having access to `p` itself. On the other hand, in an M_3 where `balance` is private, we would have:

$M_3, \kappa_1 \models \mathit{MayAffect}(\texttt{o}, \texttt{p.balance}) \rightarrow \mathit{MayAccess}(\texttt{o}, \texttt{p})$.

Policy Specifications and the obeys assertion A policy specification is parameterised by the class that should adhere to it, and consists of a set of policies.

Definition 5 (Specifications and Adherence)**.**

• $PolSpec$ ::= spec $SpcId$ < $ParId$ > { $Policy^*$ }

Adherence to policy specifications, and **obeys**.

If Spec≡ spec SpcId < ParId > {$\texttt{Pol}_1$, ... $\texttt{Pol}_n$}, then

• $M, \kappa \models$ SpcId < ClssId > iff
$\forall i \in \{1..n\}.\ M \models \texttt{Pol}_i[\texttt{ClssId}/\texttt{ParId}]$

• $M \models$ p **obeys** SpcId iff
$\forall i \in \{1,..n\}.\ Pol_i \equiv$ z : ParId $\wedge$ Rest $\rightarrow$
$\forall \kappa', \texttt{code}, \texttt{code}'.(\kappa', \texttt{code}') \in \mathcal{R}eaching(M, \kappa, \texttt{code}).$
$M, \kappa' \models \texttt{Rest}[\texttt{p}/\texttt{z}]$

In other words, p **obeys** Spec in κ, if it conforms to all Spec's policies in all configurations reachable from κ.

5. Related Work

Object Capabilities and Sandboxes. Object capabilities were first introduced [21] nine years ago, and many recent studies manage or verify safety or correctness of object capability programs. Google's Caja [22] applies sandboxes, proxies, and wrappers to limit components' access to *ambient* authority. Sandboxing has been validated formally: Maffeis et al. [19] develop a model of JavaScript, demonstrate that it obeys two principles of object capability systems and show how untrusted applications can be prevented from interfering with the rest of the system.

JavaScript analyses. More practically, Karim et al. apply static analysis on Mozilla's JavaScript Jetpack extension framework [17], including pointer analyses. Bhargavan et al. [5] extend language-based sandboxing techniques to support "defensive" components that can execute successfully in otherwise untrusted environments. Politz et al. [28] use a JavaScript typechecker to check properties such as *"multiple widgets on the same page cannot communicate."* Lerner et al. extend this system to ensure browser extensions observe *"private mode"* browsing conventions, such as that *"no private browsing history retained"* [18]. Dimoulas et al. [10] generalise the language and typechecker based approach to enforce explicit policies, that describe which components may access, or may influence the use of, particular capabilities. Alternatively, Taly et al. [33] model JavaScript APIs in Datalog, and then carry out a Datalog search for an "attacker" from the set of all valid API calls. The problem posed by the Escrow example is that it establishes a two-way dependency between trusted and untrusted systems — precisely the kind of dependencies these techniques prevent.

Relational models of trust. Artz and Gil [4] survey various types of trust in computer science generally, although trust has also been studied in specific settings, ranging from peer-to-peer systems [2] and cloud computing [15] to mobile ad-hoc networks [9], the internet of things [14], online dating [27], and as a component of a wider socio-technical system [8, 35]. Considering trust (and risk) in systems design, Cahill et al.'s overview of the SECURE project [6] gives a good introduction to both theoretical and practical issues of risk and trust, including a qualitative analysis of an e-purse example. This project builds on Carbone's trust model [7] which offers a core semantic model of trust based on intervals to capture both trust and uncertainty in that trust. Earlier Abdul-Rahman proposed using separate relations for trust and recommendation in distributed systems [1], more recently Huang and Nicol preset a first-order formalisation that makes the same distinction [16]. Solhaug and Stølen [32] consider how risk and trust are related to uncertainties over actual outcomes versus knowledge of outcomes. Compared with our work, these approaches produce models of trust relationships between high-level system components (typically treating risk as uncertainty in trust) but do not link those relations to the system's code.

Logical models of trust. Various different logics have been used to measure trust in different kinds of systems. Murray and Lowe [25] model object capability programs in CSP, and use a model checker to ensure program executions do not leak information. Carbone et al. [31] use linear temporal logic to specific trust relationships in service oriented architectures. Ries et al. [30] evaluate trust under uncertainty by evaluating Boolean expressions in terms of real values for average rating, certainty, and initial expectation. Aldini [3] describes a temporal logic for trust that supports model checking to verify some trust properties. Primiero and Taddeo [29] have developed a modal type theory that treats trust as a second-order relation over base relations between counterparties. Merro and Sibilio [20] developed a trust model for a process calculus based on labelled transition systems. Compared with our proposal, these approaches use process calculi or other abstract logical models of systems, rather than engaging directly with the system's code.

Formal Verification of Object Capability Programs. Drossopoulou and Noble [11, 26] have analysed Miller's Mint and Purse example [21] by expressing it in Joe,and in Grace [26], and discussed the six capability policies as proposed in [21]. In [12], Drossopoulou and Noble proposed a complex specification language, and used it to fully specify the six policies from [21]; however, their formalisation showed that they allowed several possible interpretations. They also uncovered the need for another four policies and formalised them as well. Most recently, Drossopoulou and Noble [13] have shown how different implementations of the underlying Mint and Purse systems coexist with different policies. In contrast, this work proposes $\mathcal{F}OCaL$, which is untyped and modelled on JavaScript rather than Java; a much simpler specification language; the **obeys** predicate to model trust; clearer definitions of accessibility predicates to model risk; a full specification of the Escrow,; and sketches of the reasoning steps using the proposed predicates.

6. Conclusions and Further Work

In this paper, we addressed the question of specification of trust and rusk, and reasoning about such specifications. To answer these questions, we proposed:

- *Hypothetical* predicates to describe who may access or modify an object or a property,
- *Obeys* predicates to describe whether an object can be trusted to satisfy some specification,
- *Open Assertions* whose validity must be guaranteed execution of *any* code,
- *Open Policies* whose validity holds in the presence of any code linked to the current code.
- *Conditional* reasoning steps.

In further work, we will develop all details of the formal model and prove soundness of the rules for reasoning. We will investigate approaches like uniqueness, ownership, and separation logic to frame our specifications, particularly to help reasoning about risk, and extend our approach to deal with concurrency and distribution.

Acknowledgments

We are grateful to Toby Murray for crucial feedback. .

References

[1] Aifarez Abdul-Rahman and Stephen Halles. A distributed trust model. In *New Security Paradigms Workshop*, 1988. Langdale, Cumbria.

[2] Karl Aberer and Zoran Despotovic. Managing trust in a peer-2-peer information system. In *CKIM*, 2001.

[3] Alessandro Aldini. A calculus for trust and reputation systems. In *IFIPTM*, 2014.

[4] Donovan Artz and Yolanda Gil. A survey of trust in computer science and the semantic web. *Journal of Web Semantics*, 2007.

[5] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, and Sergio Maffeis. Language-based defenses against untrusted browser origins. In *USENIX Security*, 2013.

[6] Cahill et al. . Using trust for secure collaboration in uncertain environments. *Pervasive Computing*, July 2003.

[7] M. Carbone, M. Nielsen, and V. Sassone. A formal model for trust in dynamic networks. In *SEFM*, 2003.

[8] Jin-Hee Cho and Kevin s. Shan. Building trust-based sustainable networks. *IEEE TECHNOLOGY AND SOCIETY MAGAZINE*, "Summer", 2013.

[9] Jin-Hee Cho, Ananthram Swami, and Ing-Ray Chen. A survey on trust management for mobile ad hoc networks. *IEEE COMMUNICATIONS SURVEYS & TUTORIALS*, 13(4), 2011.

[10] Christos Dimoulas, Scott Moore, Aslan Askarov, and Stephen Chong. Declarative policies for capability control. In *Computer Security Foundations Symposium*, 2014.

[11] Sophia Drossopoulou and James Noble. The need for capability policies. In *FTfJP*, 2013.

[12] Sophia Drossopoulou and James Noble. Towards capability policy specification and verification, May 2014. http://ecs.-victoria.ac.nz/Main/TechnicalReportSeries.

[13] Sophia Drossopoulou and James Noble. How to break the bank: Semantics of capability policies. In *iFM*, 2014.

[14] Lize Gu, Jingpei Wang, and Bin Sun. Trust management mechsnism for internet of things. *China Communications*, February 2014.

[15] Sheikh Mahbub Habib and Max Mühlhäuser Sebastian Ries and. Towards a trust management system for cloud computing. In *TrustCom*, 2011.

[16] Jingwei Huang and David M. Nicol. A formal-semantics-based calculus of trust. *IEEE INTERNET COMPUTING*, 2010.

[17] Rezwana Karim, Mohan Dhawan, Vinod Ganapathy, and Chung-Chieh Shan. An Analysis of the Mozilla Jetpack Extension FrameworK. In *ECOOP*, Springer, 2012.

[18] Benjamin S. Lerner, Liam Elberty, Neal Poole, and Shriram Krishnamurthi. Verifying web browser extensions' compliance with private-browsing mode. In *European Symposium on Research in Computer Security (ESORICS)*, September 2013.

[19] S. Maffeis, J.C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *Proc of IEEE Security and Privacy*, 2010.

[20] Massimo Merro and Eleonora Sibilio. A calculus of trustworthy ad hoc networks. *Formal Aspects of Computing*, page 25, 2013.

[21] Mark Samuel Miller. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. PhD thesis, Baltimore, Maryland, 2006.

[22] Mark Samuel Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike Stay. Safe active content in sanitized JavaScript. code.google.com/p/google-caja/.

[23] Mark Samuel Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments: From object to capabilities. In *Financial Cryptography*. Springer, 2000.

[24] Mark Samuel Miller, Tom Van Cutsem, and Bill Tulloh. Distributed electronic rights in JavaScript. In *ESOP*, 2013.

[25] Toby Murray and Gavin Lowe. Analysing the information flow properties of object-capability patterns. In *FAST*, LNCS, 2010.

[26] James Noble and Sophia Drossopoulou. Rationally reconstructing the escrow example. In *FTfJP*, 2014.

[27] Gregory Norcie, Emiliano De Cristofaro, and Victoria Bellotti. Bootstrapping trust in online dating: Social verification of online dating profiles. In *Financial Cryptography and Data Security*, 2013.

[28] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. Adsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security*, 2011.

[29] Giuseppe Primiero and Mariarosaria Taddeo. A modal type theory for formalizing trusted communications. *Journal of Applied Logic*, 10, 2012.

[30] Sebastian Ries, Sheikh Mahbub Habib, Max Mühlhäuser Sebastian Ries and, and Vijay Varadharajan. Certainlogic: A logic for modeling trust and uncertainty. In *TRUST*, 2011. LNCS 6740.

[31] Roberto Carbone et al. Towards formal validation of trust and security in the internet of services. In *Future Internet Assembly*, 2001. LNCS 6656.

[32] Solhaug and Stølen. Uncertainty, subjectivity, trust and risk: How it all fits together. In *STM*, 2011.

[33] Ankur Taly, Ulfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. Automated Analysis of Security-Critical JavaScript APIs. In *SOSP*, 2011.

[34] The Swapsies. Got Got Need. In *5: A February Records Anniversary Compilation*. February Records, 2015.

[35] Marc Walterbusch, Benedikt Martens, and Frank Teuteberg. Exploring trust in cloud computing: A multi- method approach. In *ECIS*, page 145, 2013.