The Need for Capability Policies

Position Paper

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

James Noble
Victoria University of Wellington
kjx@ecs.vuw.ac.nz

ABSTRACT

The object-capability model is one of the industry standards adopted for the implementation of security policies for web-based software. Object-capabilities in various forms are supported by programming languages such as E, Joe-E, Newspeak, Grace, and the newer versions of Javascript. Unfortunately, code written using capabilities tends to concentrate on the low-level *mechanism* rather than the high-level *policy*.

In this position paper, we argue that current specification methodologies cannot adequately capture all aspects of the *capability policies* required to support object-capability systems. We outline informally the features that such security policies should support, and we demonstrate (also informally) how we can reason that examples satisfy the capability policies.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Specification techniques; D.2.0 [**General**]: Protection mechanisms

General Terms

Object-Capability Security

Keywords

Security, Java, JavaScript, Grace

1. INTRODUCTION

Security is critically important to most programs written today — certainly to any program reachable via the Internet or that executes within a web browser or application server. Such programs typically have a number of *trusted* objects (the core of a web browser, or of an application server) that interact with *untrusted* objects (animation scripts displayed in a web page, or individual business services running on an application server). The key requirement of a secure system is to ensure that the trusted parts of the system can never be compromised by the untrusted parts: viewing a web page should never leak a user's address book or passwords, nor should an error in a business service terminate the application server.

Capabilities — unforgeable authentication tokens — have been used to provide security and task separation on multi-user machines since the 60s [11], *e.g.* PDP-1, operating systems *e.g.* CAL-TSS [22], and the CAP computer and operating system [48]. The key idea of capability-based security is that resources can only be accessed via capabilities: a program possessing a capability has the right to access the resource represented by that capability.

Object capabilities [30] apply the concept of capability to object-oriented programming languages. In an object capability system, an object is a capability for the services the object provides: any part of a program that has a reference to an object can always use all the services of that object. To restrict authority over an object, one can create an intermediate object which offers restricted services on the original object.

Object capabilities afford more fine-grained protection than privilege levels (as in Unix), static types, ad-hoc dynamic security managers (as in Java or JSand [1]), or state-machine-based event monitoring [5]. On the other hand, object capability systems are only secure as long as trusted capabilities (that is, trusted objects) are never leaked to untrusted code. Object capabilities have been adopted in several programming languages [32, 28, 46] and are increasingly used for the provision of security in web-programming in industry [33, 47, 42].

The key problem with object capability programming as practiced today is that — because capabilities are just objects — code manipulating capabilities is tangled together with code supporting the functional behaviour of the program. The actual security policies enforced by a program are *implicit*, scattered throughout the program's code. Any part of a program that uses an object may (by oversight, error, or fraud) hand that object to an untrusted part of the program, giving the untrusted code access to all the services provided by that object. This makes it difficult to determine what security properties are guaranteed by a given program, and as a result, programs are difficult to understand, check, and maintain.

In this position paper, we consider some approaches to this problem. Section 2 presents an example of object-capability programming, implemented in both statically and dynamically typed languages. Section 3 then develops the idea of capability policies that we hope to use to manage object-capability programs. Section 4 then informally explores how we may be able to show that programs adhere to capability policies: by specifying policies, reasoning about programs, potentially relying on specialised language constructs. Since this is a position paper, we end not with a conclusion, but by outlining the broad directions of our intended work in section 5.

FTfJP '13, Montpellier, France

```
1     public final class Mint {       }    //  the Mint capability
2
3     public final class Purse {
4         private final Mint mint;
5         private long balance;
6
7         public Purse(Mint mint, long balance) {  // Create new purse with money from mint.
8                  if (balance<0) { throw new IllegalArgumentException();    };
9                  this.mint = mint;  this.balance = balance;
10        }
11
12        public Purse(Purse pts) { //Create empty purse from same mint as Purse pts.
13                mint = pts.mint;  balance = 0;
14        }
15
16        public void deposit(Purse prs, long amount) { // Transfer money from prs.
17                if ( mint!=prs.mint || amount>prs.balance || amount+balance<0 )
18                      { throw new IllegalArgumentException();    };
19                prs.balance -= amount;  balance += amount;    }
20        }
21     }
```

Figure 1: The `Purse` example in Joe-E/Java, adapted from [28]

2. OBJECT-CAPABILITY EXAMPLE

To illustrate and concretise our claims and ideas, we will use as running example a system for electronic money as proposed in [32]. The example allows for mints with electronic money, purses held within mints, and transfers of funds between purses. The *currency of a mint* is the sum of the balances of all purses created by that mint. Purses trust the mint to which they belong, and programs using the money system trust their purses (and thus the mint). Crucially, separate users of the money system *do not* trust each other. The standard presentation of the mint example [32] defines six capability policies: we repeat them all here, although our discussions below will concentrate on the first three.

Pol_1 With two purses of the same mint, one can transfer money between them.

Pol_2 Only someone with the mint of a given currency can violate conservation of that currency.

Pol_3 The mint can only inflate its own currency.

Pol_4 No one can affect the balance of a purse they don't have.

Pol_5 Balances are always non-negative integers.

Pol_6 A reported successful deposit can be trusted as much as one trusts the purse one is depositing into.

An immediate consequence of these policies is that the mint capability gives its holder the ability to subvert the currency system by "printing money". This means that while purse capabilities may safely be passed around the system, the mint capability must be carefully protected.

Note, that there is also an implicit assumption that no purses are destroyed.[1]

[1]Namely, without this assumption, when a purse is destroyed then the currency of a mint may decrease, in opposition to **Pol_3**. The implication of this assumption is that there will be no explicit destruction of purses, but also no garbage collection of purses.

Several different implementations have been proposed for the mint. Fig.1 contains an implementation in Joe-E [28], a capability-oriented subset of Java, which restricts static variables and reflection. Fig.2 contains an implementation written in E [32], an object-based, capability-based language where the keywords `def` and `to` introduce objects and methods. In both implementations, the policies are only expressed implicitly.

In the Joe-E version, the policies are adhered to through the interplay of appropriate actions in the method bodies (*e.g.* the check in line 17), with the use of Java's *restrictive* language features (`private` members are visible to the same class only; `final` fields cannot be changed after initialisation; and `final` classes cannot be extended). The code concerned with the functional behaviour is tangled with the code implementing the policy (*e.g.* in `deposit`, line 19 is concerned with the functionality, while line 17 is concerned with **Pol_2**). The implementation of *one* policy is scattered throughout the code, and may use explicit runtime tests, as well as restrictive elements (*e.g.* **Pol_2** is implemented through a check in line 17, the `private` and `final` annotations, and the initialisations in lines 9 and 13). Note that an apparently innocuous change to this code — such as a public `getMint` accessor that returned a purse's `mint` — would be enough to leak the `mint` to untrusted code, destroying the security of the whole system.

In the E version, as the language does not offer many restrictive features compared with Java, the function `makeBrandPair` creates two associated objects, a sealer and an unsealer, such that the sealer can seal any entity in a box and the unsealer is the only object which can retrieve the contents of the box. Execution of `p1.deposit(p2,_)` will apply `p1`'s unsealer to unseal the `decr` closure of `p2`, sealed by `p2`'s sealer. Unsealing will be successful only if the unsealer of `p1` corresponds to the sealer of `p2` - *i.e.* if `p1` and `p2` belong to the same mint. The E implementation creates extra objects whose sole purpose is the implementation of the policy. For example, a mint with two purses requires a sealer, an unsealer, and two `decr` closures, while the execution of `deposit` creates the intermediately sealed version of the argument's `decr` closure. Again, a small change to the code — such as the purse returning the `unsealer` or passing it to another object — would leak that

```
def makeMint(name) :any {
    def [sealer, unsealer] := makeBrandPair(name)
    def mint {
        to makePurse(var balance :(int >= 0)) :any {
            def decr(amount :(0..balance)) :void {
                 balance -= amount
            }
            def purse {
                to sprout() :any { return mint.makePurse(0) }
                to getDecr() :any { return sealer.seal(decr) }
                to deposit(amount :int, src) :void {
                    unsealer.unseal(src.getDecr())(amount)
                    balance += amount
                 }
            }
            return purse
        }
    }
    return mint
}
```

Figure 2: The `Purse` example in E, taken from [32]

capability and undermine the security.

Both in the Joe-E and the E version, we have tangling of policy with functionality, as well as scattering of the policy implementations.

3. CAPABILITY POLICIES

We define a *capability policy* as a specification that determines how capabilities are intended to be used within a program: which objects are trusted, which are untrusted, and precisely which capabilities can be accessed by which object. A key feature of capability systems is the *principle of least authority* — a program object should only be able to access the capabilities (i.e. the other objects) that it needs in order to function correctly: even a trusted object should not have access to all the capabilities (objects) in the system [40, 34, 48]. A range of object capability policies are discernible from the literature [30, 32, 31].These policies generally have the following characteristics:

- They are *program centred*: they talk about properties of programs rather than protocols.
- They are *fine-grained*: they can talk about *individual objects*, while *coarse-grained* policies only talk about large components such as `System` or `DOM`.
- They are *open*. *Open* requirements must be satisfied for any use of the code *extended in any possible manner*, while *closed* requirements need only be satisfied for any use of code itself.
- They have *rely* as well as *deny* elements. Rely elements promise that execution on a state satisfying a given pre-condition will reach another state which satisfies some post-condition [19]. Deny elements promise that if an execution reaches a certain state, or changes state in a certain way, or accesses some program entity, then the code must satisfy some given properties. In other words, rely policies are about sufficient conditions, while deny policies are about necessary conditions.

None of the terms above are standard; we coined them to help us delineate our intended research. The mint's policies are capability policies, because:

- They talk about actual programs;
- They talk about *individual* purses and mints;
- The policy in [32] is open; any expansion of the code (through dynamic loading, subclassing, mashups *etc.*) should satisfy the requirements.
- **Pol_1** is a rely requirement: executing `deposit` in a state where the two purses belong to the same mint leads to a state where the money has been transferred.
- **Pol_2** is a deny requirement; it says that a currency may be changed by some code only if the code contained a function call executed by the mint owning the currency.
- **Pol_3** is another deny requirement; it says that if the currency should change, then it increases.
- **Pol_4** is also a deny requirement, preventing objects that cannot access a purse from modifying its value.
- **Pol_5** is very similar to **Pol_3**, requiring purses' balances to be positive.
- **Pol_6** can be formulated as a combination of a conditional rely requirement (if the purse is trusted then the deposit is trusted) and a deny requirement (that a deposit operation cannot have a larger effect than the footprint of the purse into which it is deposited — and, presumably, the purse from which the deposited funds are withdrawn, although that is not explicitly stated in the policy).

Open policies are central for Javascript security, which requires that in any mashup, untrusted code cannot access the trusted security-critical resources of the execution environment (*e.g.* the DOM), nor interfere with the execution of any other component [27, 21]. These works usually implement coarse-grained, fixed-in-advance policies. SecureJS [45] leverages local scoping (a restrictive feature) to prove fine-grained confinement (*e.g.* `decr` is confined), but cannot express the high-level policies (*e.g.* currency cannot be affected). JSand [1] uses Secure ECMAScript and proxies (other restrictive features) to isolate the DOM, and then ensures access to that DOM proxy are mediated by dynamically checking security policies expressed as JavaScript predicates.

Deny policies are related to *deny-guarantee* specifications [13] which can forbid given locations from being modified by the current, or by the other threads. Deny policies typically apply throughout program execution, rather than during specific functions, and may talk about any properties of the program (*e.g.* the currency), rather than specific locations.

Deny policies are also related to *correspondence assertions* [49, 17], which require that a principal reaching a certain point in a protocol must be preceded by some other principal reaching a corresponding point. Recently, correspondence assertions have been adapted to talk about program state, and thus can prove that the code adheres to security, authentication, and privacy policies [6]: functions are annotated by *refinement types* which require that the function is only called if its arguments satisfy the type's conditions.

Deny policies go further than correspondence assertions in the following significant ways:

- They support *implicit* properties, *i.e.* properties which depend on state reachable from more than one object, perhaps quantifying over the complete heap, or even on the history of execution. In our example, the currency is the sum of the balance of all purses from the same mint, and therefore is an implicit property.

- They are *pervasive*, *i.e.* they are not attached to one function, and may be affected by several different methods. For example, the currency may be affected by the creation of purses and the payments.

- They are *persistent*, *i.e.* they allow the comparison of properties of the state at different times in execution. For example, **Pol_3** compares the currency between any two times in execution.

Deny policies could be transformed into equivalent refinement types; however, the transformation would not be trivial, and the resulting policies would not be open (because the refinement types cannot prevent the addition of functions which break the requirements), and less abstract (how would refinement types express that the currency can only grow?).

4. REASONING ABOUT CAPABILITY POLICIES

Our ultimate goal is to make capabilities and capability policies explicit in specifications as well as in programs, and then to formally prove that programs adhere to policies. In this section we outline our current ideas on the specification of capability policies, and on the verification that code does, indeed, adhere to such policies. We also consider specialised language constructs to support this reasoning.

The ideas in this section are starting points only. More work is needed, and planned.

4.1 Specifying Policies

To specify capability policies, we must be able to specify both rely and deny policies. For rely properties, we can draw from specification languages for functional properties, *e.g.* JML [23], or separation-logic-based [41, 36], enhanced so as to also talk about indirect properties.

Unfortunately, the pervasive nature of deny properties means that they cannot be treated through preconditions on methods (as *e.g.* in [6]). Instead, we will need to draw upon ideas from various modal and temporal logics [16, 4], but talking about program entities, rather than events. For example, in an expanded notation, $\Box\ \forall p1, p2 : Purse, amt : Nat.\ (\ p1.deposit(p2, amt) \longrightarrow \exists m : Mint. \blacklozenge p1 = m.Purse() \wedge \blacklozenge p2 = m.Purse()\ )$, says that transfer of moneys is successful only if the purses had been previously created by the same mint.[2]

The persistent nature of deny properties necessitates refining the relations between different instants in time, *e.g.* a change in the balance of a purse is preceded by a transfer, which in its turn, again, is preceded by the mint creating the two purses. In more detail: if the balance of a purse of $p1$ decreases by amt over its immediately previous value, then the immediately preceding step executed $p1.deposit(p2, mt)$, and at some times prior to that step, the purses $p1$ and $p2$ were created by the same mint. The annotation val_{prev} is meant to indicate a value *immediately before* the event in question, and the annotation θ_{prec} is meant to indicate an event *immediately preceding* the event in question. Thus, we express this policy as follows:

$$
\begin{array}{l}
\Box\ \forall p1 : Purse, amt : Nat :\\
\quad p1.balance == (\{p1.balance\}_{prev} - amt) \quad \longrightarrow\\
\qquad (\exists p2 : Purse.\{p1.deposit(p2, amt)\}_{prec} \wedge\\
\qquad\quad \exists m : Mint. \blacklozenge(p1 = m.Purse()) \ \wedge\ \blacklozenge(p2 = m.Purse())\ ).
\end{array}
$$

Another facet of deny properties is the different modes of causality. For example, does **Pol_2** mean that a change in the currency implies that the mint object was accessible, or, more strongly, that the mint executed a method? Classical approaches to ownership, for example, support the latter approach[12].

Object invariants [29, 35, 44] are relevant, *e.g.* an object's sealer and unsealer must come from the same mint. Monotonic properties are relevant too, *e.g.* **Pol_3** says that the currency can only grow. Such properties are akin to history invariants [25]. Accommodating for object and history invariants poses the challenge of deciding at which point they may be broken/must be restored [3, 24]; known approaches follow different, but fixed rules [14], we shall investigate whether the rules could be part of user-defined policies.

To address the crucial issue of capabilities leaking from trusted to untrusted code, we can apply techniques drawn from ownership types [8, 18, 7, 15]. Ownership types restrict heap topology to manage access between objects, and have generally been used to support encapsulation and concurrency. By applying ownership to capabilities, we will be able to support many deny policies directly: **Pol_4** and **Pol_5**, for example, or the policy of a client object of the currency system: that if an object owns its purse, no other objects should be able to access that purse.

We also expect to be able to employ effects systems [26] to restrict interactions between sets of object, *e.g.* $\forall m, m' : Mint.m \neq m' \rightarrow m.Purse() \# m'.Purse()$ says that `Purses` created by different `Mints` will not affect each other. These systems can expand our earlier work on effects and isolation [9, 15].

We expect to define the semantics of the specification language by means of satisfiability of assertions in the context of a given stack and heap [36]. For deny policies, we will have to expand the approach, define satisfiability over the history of executions. [4].

---

[2] As in [4], the operator $\Box\ \theta$ expresses that formula θ holds in all subsequent states, while the operator $\blacklozenge\ \theta$ expresses that θ held at some earlier stage. Therefore, in more detail, the requirement $\Box\ \forall p1, p2 : Purse, amt : Nat.\ (\ p1.deposit(p2, amt) \longrightarrow \exists m : Mint. \blacklozenge p1 = m.Purse() \wedge \blacklozenge p2 = m.Purse()\ )$, says that for any two purses $p1$ and $p2$, if the term $p1.deposit(p2, amt)$ executes successfully, then, for some Mint m, at some earlier time, $p1$ was created by calling the method *Purse* on m, and at some other earlier time, $p2$ was created by calling the method *Purse* on the same mint, m.

4.2 Reasoning About Capability Policies

The main challenge in reasoning about programs' adherence to capability policies is reasoning about deny policies, and the combination of rely and deny steps. We have no full logics yet, but have initial ideas, which we discuss in terms of the code examples.

We first consider the code written in Joe-E/Java (Fig.1). In this example, the currency of a mint is the sum of balances of the purses whose `mint` field points to that mint.

The treatment of **Pol_1** requires nothing more than standard Hoare Logic: if `prs1` and `prs2` share `mint`s, then a call of

```
prs1.deposit(prs2,amt)
```

transfers `amt` from `prs2` to `prs1` (for appropriate amounts and balances).

In contrast, **Pol_2** requires a novel kind of reasoning. Because `mint` is final (and implicitly assuming `Purse`s are not destroyed) the set of purses within a mint can only be affected through the creation of a new `Purse`. By inspection of lines 8 and 19, the method `deposit` and constructor `Purse(Purse)` preserve the currency in the mint. Moreover, since `balance` is private, any modifications to `balance` must be done through the methods of class `Purse`. Therefore, the only way to affect the currency is through the constructor `Purse(_,_)`, which takes the mint as a parameter.

We now briefly look at the E code in Fig.2. and its adherence to **Pol_2**: The function `makeMint` creates a mint and a sealer/unsealer pair. Here the currency of the mint is the sum of the balance of all purses that have been sealed by the mint's sealer, and consequently can be unsealed by the mint's unsealer. The `makePurse` function creates objects which have access to the sealer/unsealer. Other than the mint itself, and its purses, no other object has access to that pair. By inspection of lines 5,6 and 9-13, we see that the only operation which affects the currency is `makePurse(balance)`, which can only be executed by the mint object.

The arguments used above do not fit the Hoare Logic nor the type-inference format. Nevertheless, they reflect the way one informally reasons about code. They argue in terms of the footprint of a property, and of the set of method calls which might affect that footprint. They consider the uses of restrictive language features (*e.g.* `final`) in the program to reduce that set. They also use rely reasoning (*e.g.* calls to `deposit` or `Purse(Purse)` preserve the currency in the mint).

A formal logic to support reasoning about capability policies will need to combine both rely and deny steps. It will have the usual Hoare Logic rules, as well as inference rules for the calculation of footprints of properties, the effect of restrictive features, for the passing of object capabilities, for lexically scoped languages. To prove soundness of our logic [39] we will need to expand the approach to deal with the deny arguments, perhaps applying ideas from provenance [37], and considerate reasoning [43].

4.3 Language Features

We are also considering the extent to which particular language constructs can support reasoning about policies — both for extant features and potential novel features. We have already seen how reasoning about object-capability programming in the class-based Java style (in Fig.1) differs in some important respects from a lexically-scoped E style (in Fig.2): we would like to extend this analysis to understand particular constructs in more detail.

We have begun collecting programming language idioms often used in capability programs (*e.g.* sealers, revocation, membranes [20, 47, 30], *e.t.c.*), and identify idioms which have the same effect (*e.g.,* the use of field `mint` in Fig.1 has the same effect as that of `sealer`/`unsealer` in Fig.2). We will lift idioms to more succinct, abstract language features.

Consider the use of the field `mint` in the code in Fig.1: its purpose is to ensure that no transactions involve `Purse`s from different `Mint`s. This is enforced through the `private` annotation (line 4), initialisation (lines 9 and 13), and check (line 17). The idiom would be directly expressible more directly in a variation of ownership types [8] which allowed for dynamic checks for owners [18, 38]. Making the `Mint` the owner of the `Purse`s and replacing the field declaration, the initialisation, and the check mentioned above through one type argument to `Mint` would reduce the code by 30%. Crucially, it would also prevent a purse from ever leaking its mint capability to an untrusted object.

An extension of the money example is that `Purse`s should belong to `Person`s, and that `Person`s should not have access to `Purse`s belonging to other `Person`s. Thus, a person `p1` wanting to pay person `p2`, could create a `Purse`, pay some amount into it, and then make it belonged to `p2`, thus ensuring that the purse can be safely passed around and not be tampered with. This can be modelled by multiple ownership, here with a `Mint` and a `Person` owner [7, 15] although we need to discriminate to allow for different treatment of, and different roles for, the different owners: the `Person` owner guarantees encapsulation, is checked statically, and is mutable, while the `Mint` owner makes no encapsulation guarantees, is checked dynamically, and is immutable. Encapsulation is often implicitly present in programs written in dynamic languages: In Fig.2, all `purses` created by the same `mint` share, and do not leak further, the same `sealer`/`unsealer` pair. The code could be made more succinct and more abstract — not to mention more secure — with dynamically checked owners[18].

More generally, we are interested in the role of restrictive features in supporting deny policies. We expect to expand these features and allow dynamically enforced versions of the restrictions: dynamic application and revocation of the restrictions, dynamic linking of ownership domains[2], dynamic merger or dissolution of ownership boxes, *etc.*Such features have their counterpart in the dynamic treatment of capabilities,*e.g.* revocation, membranes, and proxies [20, 47, 10]. These kind of dynamically enforced properties inspired by static type systems seems to offer interesting opportunities for cross-pollination between static and dynamic languages.

5. CONCLUSION AND FUTURE WORK

In this position paper, we have advocated that capability policies are a necessary adjunct to reasoning about programs using object-capability security. Object capabilities make it possible to write secure programs even in dynamically typed object-oriented languages, but whether dynamically or statically typed, such programs security properties will be implicit in their source code. Capability policies have the potential to allow programmers to specify their expectations about programs' security properties, and hopefully will let us check (statically or dynamically), and then argue formally, that a particular program does in fact obey its desired security policies.

Acknowledgments

We are grateful to Mark Miller, Sylvan Klebsch, Robert O'Callaghan, Neal Glew, Sergio Maffeis, Lawrence Tratt, Marco Servetto, Gavin Bierman, Chris Hawlblitzel, Manuel Faehndrich, and the anonymous reviewers for discussions on this material; their challenges helped us clarify our ideas. This work is supported by the Royal Society of New Zealand Marsden Fund.

6. REFERENCES

[1] Pieter Agten, Steven Van Acker, Yoran Brondsema, Phu H. Phung, Lieven Desmet, and Frank Piessens. Jsand: Complete client-side sandboxing of third-party JavaScript without browser modifications. In *ACSAC*, 2012.

[2] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, Springer, 2004.

[3] M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, LNCS, 2004.

[4] David Basin, , Felix Klaedtke, and Samuel Müller. Monitoring security policies with metric first-order temporal logic. In *SACMAT*, 2010.

[5] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *PLDI*, 2005.

[6] Jesper Bengtson, Kathiekeyan Bhargavan, Cedric Fournet, Andrew Gordon, and S.Maffeis. Refinement Types for Secure Implementations. *ACM ToPLAS*, 2011.

[7] Nicholas Cameron, Sophia Drossopoulou, James Noble, and Matthew Smith. Multiple Oownership. In *OOPSLA*, ACM, 2007.

[8] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*. ACM, 1998.

[9] Dave Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Types and Effects. In *OOPSLA*, 2002.

[10] Tom Van Cutsem and Mark S. Miller. Trustworthy proxies: Virtualizing objects with invariants. In *ECOOP*, 2013.

[11] Jack B. Dennis and Earl C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Comm. ACM*, 9(3), 1966.

[12] Werner M. Dietl and Peter Müller. Object Ownership in Program Verification. *Aliasing in Object-Oriented Programming*, 2012.

[13] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *ESOP*. Springer, 2009.

[14] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, LNCS. Springer, 2008.

[15] Sophia Drossopoulou, David Clarke, and James Noble. Roles for Owners - Work in Progress. In *IWACO 2011*, ACM DL, July 2011.

[16] Deepak Garg, Lujo Bauer, Kevin D. Bowers, Frank Pfenning, and Michael K. Reiter. A linear knowledge of authorization and knowledge. In *ESoRICS*, LNCS. Springer, 2006.

[17] Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. In *MFPS*. Elsevier, ENTCS, 2001.

[18] Donald Gordon and James Noble. Dynamic Ownership in a Dynamic Language. In *Dynamic Languages Symposium*. ACM, 2007.

[19] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[20] Yves Jaradin, Fred Piessens, and Peter Van Roy. Capability confinement by membranes, 2005. TR Université Catholique De Louvain.

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

[22] Butler W. Lampson and Howard E. Sturgis. Reflection on an Operating System Design. *Communications of the ACM*, 19(5), 1976.

[23] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Iowa State Univ. `www.jmlspecs.org`, February 2007.

[24] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP*, Springer, 2004.

[25] K. Rustan M. Leino and Wolfram Schulte. Using history invariants to verify observers. In *ESOP*, 2007.

[26] Y. Lu and J. Potter. Protecting Representation with Effect Encapsulation. In *POPL*, pages 359–371, 2006.

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

[28] Adrian Mettler, David Wagner, and Tyler Close. Joe-E a Security-Oriented Subset of Java. In *NDSS*, 2010.

[29] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

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

[31] Mark Samuel Miller. Secure Distributed Programming with Object-capabilities in JavaScript. Talk at Vrije Universiteit Brussel, mobicrant-talks.eventbrite.com, October 2011.

[32] Mark Samuel Miller, Chip Morningstar, and Bill Frantz. Capability-based Financial Instruments: From Object to Capabilities. In *Financial Cryptography*. Springer, 2000.

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

[34] Roger Needham. Protection systems and protection implementations. In *Joint Computer Conference*, pages 571–578, 1972.

[35] Matthew Parkinson. Class invariants: the end of the road? In *IWACO*, 2007.

[36] Matthew Parkinson and Alexander J. Summers. The Relationship between Separation Logic and Implicit Dynamic Frames. In *ESOP*, 2011.

[37] Roly Perera, Umut Acar, James Cheney, and Paul Blain Levy. Functional programs that explain their work. In *ICFP*. ACM, 2012.

[38] Alex Potanin, Monique Damitio, and James Noble. Are your incoming aliases really necessary? Counting the cost of object ownership. In *ICSE*, 2013.

[39] Azalea Raad and Sophia Drossopoulou. A Sip of the Chalice. In *FTFJP*, July 2011.

[40] Jerome H. Saltzer. Protection and the control of information sharing in Multics. *CACM*, 17(7):p.389ff, 1974.

[41] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit Dynamic Frames. *ToPLAS*, 2012.

[42] Marc Stiegler. The lazy programmer's guide to security. HP Labs, www.object-oriented-security.org.

[43] Alexander J. Summers and Sophia Drossopoulou. Considerate Reasoning and the Composite Pattern. In *VMCAI*, 2010.

[44] Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. The need for Flexible Object Invariants. In *IWACO*, ACM DL, July 2009.

[45] Ankur Taly, Ulfar Erlingsson, John C. Mitchell, Mark S. Miller, and Jasvir Nagra. Automated Analysis of

Security-Critical JavaScript APIs. In *IEEE Symposium on Security and Privacy (SP)*, 2011.

[46] The NewpeakTeam. Several Newspeak Documents. newspeaklanguage.org/, September 2012.

[47] Tom van Cutsem. Membranes in Javascript. prog.vub.ac.be/ tvcutsem/invokedynamic/js-membranes.

[48] M. V. Wilkes and R. M. Needham. The Cambridge CAP computer and its operating system, 1979.

[49] T. Wood and S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, 1993.