Static Use-Based Object Confinement

Christian Skalka
The Johns Hopkins University
ces@cs.jhu.edu

Scott Smith
The Johns Hopkins University
scott@cs.jhu.edu

Abstract

The confinement of object references is a significant security concern for modern programming languages. We define a language that serves as a uniform model for a variety of confined object reference systems. A *use-based* approach to confinement is adopted, which we argue is more expressive than previous communication-based approaches. We then develop a readable, expressive type system for static analysis of the language, along with a type safety result demonstrating that run-time checks can be eliminated. The language and type system thus serve as a reliable, declarative and efficient foundation for secure capability-based programming and object confinement.

1 Introduction

The confinement of object references is a significant security concern in languages such as Java. Aliasing and other features of OO languages can make this a difficult task; recent work [21, 4] has focused on the development of type systems for enforcing various containment policies in the presence of these features. In this extended abstract, we describe a new language and type system for the implementation of object confinement mechanisms that is more general than previous systems, and which is based on a different notion of security enforcement.

Object confimement is closely related to *capability*-based security, utilized in several operating systems such as EROS [16], and also in programming language (PL) architectures such as J-Kernel [6], E [5], and Secure Network Objects [20]. A capability can be defined as a reference to a data segment, along with a set of access rights to the segment [8]. An important property of capabilities is that they are *unforgeable*: it cannot be faked or reconstructed from partial information. In Java, object references are likewise unforgeable, a property enforced by the type sytem; thus, Java can also be considered a statically enforced capability system.

So-called *pure* capability systems rely on their high level design for safety, without any additional system-level mechanisms for enforcing security. Other systems *harden* the pure model by layering other mechanisms over pure capabilities, to provide stronger system-level enforcement of security; the `private` and `protected` modifiers in Java are an example of this. Types improve the hardening mechanisms of capability systems, by providing a declarative statement of security policies, as well as improving run-time efficiency through static, rather than dynamic, enforcement of security. Our language model and static type analysis focuses on capability hardening, with enough generality to be applicable to a variety of systems, and serves as a foundation for studying object protection in OO lanaguages.

2 Overview of the pop system

In this section, we informally describe some of the ideas and features of our language, called pop, and show how they improve upon previous systems. As will be demonstrated in Sect. 5, pop is sufficient to implement various OO language features, e.g. classes with methods and instance variables, but with stricter and more reliable security.

2.0.1 Use vs. communication-based security

Our approach to object confinement is related to previous work on containment mechanisms [2, 4, 21], but has a different basis. Specifically, these containment mechanisms rely on a *communication*-based approach to security; some form of barriers between objects, or domain boundaries, are specified, and security is concerned with communication of objects (or object references) across those boundaries. In our *use*-based approach, we also specify domain boundaries, but security is concerned with how objects are *used* within these boundaries. Practically speaking, this means that security checks on an object are performed when it is used (selected), rather than communicated.

The main advantage of the use-based approach is that security specifications may be more fine-grained; in a communication based approach we are restricted to a whole-object "what-goes-where" security model, while with a use-based approach we may be more precise in specifying what methods of an object may be used within various domains. Our use-based security model also allows "tunneling" of objects, supporting the multitude of protocols which rely on an intermediary that is not fully trusted.

$$
\begin{array}{llll}
m, x, \varsigma, \text{set}, \text{get} \in \mathit{ID},\ \iota \subseteq \mathit{ID} & & & \textit{identifiers} \\
l \in \mathit{Loc} & & & \textit{locations} \\
d \in \mathcal{D},\ D \subseteq \mathcal{D} & & & \textit{domains} \\
\varphi \in \mathcal{D} \to 2^{\mathit{ID}} & & & \textit{interfaces} \\
\varrho & ::= & m_i(x) = e_i \ ^{0<i\leq n} & \textit{method lists} \\
co & ::= & [\varrho] \cdot d \mid l & \textit{core objects} \\
o & ::= & co \cdot \varphi \mid \mathbf{weak}_\iota(o) & \textit{object definitions} \\
v & ::= & x \mid o & \textit{values} \\
e & ::= & v \mid e.m(e) \mid e \upharpoonright (d, \iota) \mid \text{let } x = v \text{ in } e \mid \text{ref}_\varphi e & \textit{expressions} \\
E & ::= & [\,] \mid E.m(e) \mid v.m(E) \mid E \upharpoonright (d, \iota) \mid \text{ref}_\varphi E \mid \mathbf{weak}_\iota(E) & \textit{evaluation contexts}
\end{array}
$$

Figure 1: Grammar for pop

2.0.2 Casting and weakening

Our language features a *casting* mechanism, that allows removal of access rights from particular views of an object, resulting in a greater attenuation of security when necessary. This casting discipline is statically enforced. It also features *weak* capabilities, a sort of deep-casting mechanism inspired by the same-named mechanism in EROS [16]. A weakened capability there is read-only, and any capabilities read from a weakened capability are automatically weakened. In our higher-level system, capabilities *are* objects, and any method access rights may be weakened.

2.0.3 Static protection domains

The pop language is an object-based calculus, where object methods are defined by lists of method definitions in the usual manner. For example, substituting the notation ... for syntactic details, the definition of a file object with read and write methods would appear as follows:

$$[\text{read}() = \ldots, \text{write}(x) = \ldots] \cdot \ldots \cdot \ldots$$

Additionally, every object definition statically asserts membership in a specific *protection domain* d, so that expanding on the above we could have:

$$[\text{read}() = \ldots, \text{write}(x) = \ldots] \cdot d \cdot \ldots$$

While the system requires that all objects are annotated with a domain, the *meaning* of these domains is flexible, and open to interpretation. Our system, considered in a pure form, is a core analysis that may be specialized for particular applications. For example, domains may be as interpreted as code owners, or they may be interpreted as denoting regions of static scope—e.g. package or object scope.

Along with domain labels, the language provides a method for specifying a security policy, dictating how domains may interact, via *user interface* definitions φ. Each object is annotated with a user interface, so that letting φ be an appropriately defined user interface and again expanding on the above, we could have:

$$[\text{read}() = \ldots, \text{write}(x) = \ldots] \cdot d \cdot \varphi$$

We describe user interfaces more precisely below, and illustrate and discuss relevant examples in Sect. 5.

2.0.4 Object interfaces

Other secure capability-based language systems have been developed [5, 6, 20] that include notions of access-rights interfaces, in the form of object types. Our system provides a more fine-grained mechanism: for any given object, its user-interface definition φ may be defined so that different domains are given more or less restrictive views of the same object, and these views are statically enforced. Note that the use-based, rather than communication-based, approach to security is an advantage here, since the latter allows us to more precisely modulate *how* an object may be used by different domains, via object method interfaces.

For example, imagining that the file objects defined above should be read-write within their local domain, but read only outside of it, an appropriate definition of φ for these objects would be as follows:

$$\varphi \triangleq \{d \mapsto \{\text{read}, \text{write}\}, \partial \mapsto \{\text{read}\}\}$$

The distinguished domain label ∂ matches any domain, allowing *default* interface mappings and a degree of "open-endedness" in program design.

The user interface is a mapping from domains to access rights—that is, to sets of methods in the associated object that each domain is authorized to use. This looks something like an ACL-based security model; however, ACLs are defined to map *principals* to privileges. Domains, on the other hand, are fixed boundaries in the code which may have nothing to do with principals, The practical usefulness of a mechanism with this sort of flexibility has been described in [3], in application to mobile programs.

$$
\begin{array}{rcll}
d, ([\varrho]\cdot d' \cdot \varphi).m_i(v), \sigma & \rightarrow & d', (e_i[([\varrho]\cdot d' \cdot \varphi')/\varsigma])[v/x], \sigma & (\textit{send}) \\
\multicolumn{4}{l}{\quad \text{where } \varrho = (m_j(x) = e_j \;{}^{0<j\leq n}),\ 0 < i \leq n,\ m_i \in \varphi(d)} \\
\multicolumn{4}{l}{\quad \text{and } \varphi' = \{d' \mapsto \{m_1, \ldots, m_n\}\}} \\
d, (co \cdot \varphi) \mid (d', \iota), \sigma & \rightarrow & d, (co \cdot \varphi \oplus d' \mapsto \iota), \sigma \qquad \iota \subseteq \varphi(d') & (\textit{cast}) \\
d, \mathbf{weak}_\iota(o) \mid (d', \iota'), \sigma & \rightarrow & d, \mathbf{weak}_\iota(o \mid (d', \iota')), \sigma & (\textit{castweak}) \\
d, \mathbf{weak}_\iota(o).m(v), \sigma & \rightarrow & d', \mathbf{weak}_\iota(e), \sigma' & (\textit{weaken}) \\
\multicolumn{4}{l}{\quad \text{if } m \notin \iota \text{ and } d, o.m(v), \sigma \rightarrow d', e, \sigma'} \\
d, \mathrm{ref}_\varphi v, \sigma & \rightarrow & d, l \cdot \varphi, \sigma \oplus l \mapsto v \qquad l \notin \mathrm{dom}(\sigma) & (\textit{newcell}) \\
d, (l \cdot \varphi).\mathrm{set}(v), \sigma & \rightarrow & d, v, \sigma \oplus l \mapsto v \qquad \text{if set} \in \varphi(d) & (\textit{set}) \\
d, (l \cdot \varphi).\mathrm{get}(), \sigma & \rightarrow & d, \sigma(l), \sigma \qquad \text{if get} \in \varphi(d) & (\textit{get}) \\
d, \mathrm{let}\, x = v \,\mathrm{in}\, e, \sigma & \rightarrow & d, e[v/x], \sigma & (\textit{let}) \\
d, E[e], \sigma & \rightarrow & d', E[e'], \sigma' \qquad \text{if } d, e, \sigma \rightarrow d', e', \sigma' & (\textit{context})
\end{array}
$$

Figure 2: Operational semantics for pop

3 The language pop: syntax and semantics

We now formally define the syntax and operational semantics of pop, an object-based language with state and capability-based security features. The grammar for pop is defined in Fig. 1. It includes a countably infinite set of identifiers $\mathcal{D}$ which we refer to as *protection domains*. The definition also includes the notation $m_i(x) = e_i \;{}^{0<i\leq n}$ as an abbreviation for $m_1(x) = e_1, \ldots, m_n(x) = e_n$; henceforth we will use this same vector abbreviation method for all language forms. Read-write cells are defined as primitives, with a cell constructor $\mathrm{ref}_\varphi v$ that generates a read-write cell containing v, with user interface φ. The object weakening mechanism $\mathbf{weak}_\iota(o)$ described in the previous section is also provided, as is a casting mechanism $o \mid (d, \iota)$, which updates the interface φ associated with o to map d to ι. The operational semantics will ensure that only *downcasts* are allowed.

We require that for any φ and d, the method names $\varphi(d)$ are a subset of the method names in the associated object. Note that object method definitions may contain the distinguished identifier ς which denotes *self*, and which is bound by the scope of the object; objects always have full access to themselves via ς. We require that self never appear "bare"—that is, the variable ς must always appear in the context of a method selection $\varsigma.m(e)$. This restriction ensures that ς cannot escape its own scope, unintentionally providing a "back-door" to the object. *Rights amplification* via ς is still possible, but this is a *feature* of capability-based security, not a flaw of the model.

The small-step operational semantics for pop is defined in Fig. 2 as the relation $\rightarrow$ on *configurations* d, e, σ, where *stores* σ are partial mapping from locations l to values v. The reflexive, transitive closure of $\rightarrow$ is denoted $\rightarrow^\star$. If $d_1, e, \varnothing \rightarrow^\star d', e', \sigma'$ with d_1 the top-level domain for all all programs, then if e' is a value we say e *evaluates to* e', and if e' is not a value and d', e', σ' cannot be reduced, then e is said to *go wrong*. If $x \in \mathrm{dom}(f)$, the notation $f \oplus x \mapsto v$ denotes the function which maps x to v and otherwise is equivalent to f. If $x \notin \mathrm{dom}(f)$, $f \oplus x \mapsto v$ denotes the function which extends f, mapping x to v. All interfaces φ are *quasi*-constant, that is, constant except on a known finite set D (since they're statically user-defined with default values), and we write $\varphi \oplus \partial \mapsto \iota$ to denote φ' which has default value ι, written $\varphi'(\partial)$, and where $\varphi'(d) = \varphi(d)$ for all $d \in D$.

In the *send* rule, full access to self is ensured via the object that is substituted for ς; note that this is the selected object o, updated with an interface φ' that confers full access rights on the domain of o. The syntactic restriction that ς never appear bare ensures that this strengthened object never escapes its own scope.

Of particular note in the semantics is the myriad of runtime security checks associated with various language features; our static analysis will make these uneccessary, by compile-time enforcement of security.

4 Types for pop: the transformational approach

To obtain a sound type system for the pop language, we use the *transformational* approach; we define a semantics-preserving transformation of pop into a *target* language, that comes pre-equipped with a sound let-polymorphic type system. This technique has several advantages: since the transformation is computable and easy to prove correct, a sound *indirect* type system for pop can be obtained as the composition of the transformation and type judge-

$$
\begin{aligned}
\{d_1 \mapsto \iota_1, \widehat{\cdots, d_n \mapsto} \iota_n, \partial \mapsto \iota\} &= \{\varnothing\}\{\partial = \iota\}\{d_1 = \iota_1\}\ldots\{d_n = \iota_n\} \\
[\![x]\!]_d &= x \\
[\![\varsigma]\!]_d &= (\varsigma\{\}) \\
[\![[m_i(x) = e_i \ {}^{0<i\leq n}] \cdot d' \cdot \varphi]\!]_d &= \ \text{let}\ o_\varsigma = \mathsf{fix}\,\varsigma.\lambda\_.\{\text{obj} = \{m_i = \lambda x.[\![e_i]\!]_{d'} \ {}^{0<i\leq n}\}, \\
&\qquad\qquad\qquad\qquad\quad \text{ifc} = \{d' = \{m_1, \ldots, m_n\}\}, \\
&\qquad\qquad\qquad\qquad\quad \text{strong} = \bar{\varnothing}\} \ \ \text{in} \\
&\quad\ \{\text{obj} = (o_\varsigma\{\}).\text{obj}, \text{ifc} = \hat{\varphi}, \text{strong} = \bar{\varnothing}\} \\
[\![e_1.m(e_2)]\!]_d &= \ \text{let}\ c_1 = [\![e_1]\!]_d \ \text{in} \\
&\quad\ c_1.\text{strong} \ni m; \\
&\quad\ (c_1.\text{ifc}.d \vee c_1.\text{ifc}.\partial) \ni m; \\
&\quad\ \text{let}\ c_2 = (c_1.\text{obj}.m)([\![e_2]\!]_d) \ \text{in} \\
&\quad\ \text{let}\ w = c_1.\text{strong} \wedge c_2.\text{strong} \ \text{in} \\
&\quad\ \{\text{obj} = c_2.\text{obj}, \text{ifc} = c_2.\text{ifc}, \text{strong} = w\} \\
[\![e \mid (d', \{m_1, \ldots, m_n\})]\!]_d &= \ \text{let}\ c = [\![e]\!]_d \ \text{in} \\
&\quad\ c.\text{ifc}.d' \ni m_1; \ldots; c.\text{ifc}.d' \ni m_n; \\
&\quad\ \text{let}\ i = (c.\text{ifc})\{d' = \{m_1, \ldots, m_n\}\} \ \text{in} \\
&\quad\ \{\text{obj} = c.\text{obj}, \text{ifc} = i, \text{strong} = c.\text{strong}\} \\
[\![\mathbf{weak}_\iota(e)]\!]_d &= \ \text{let}\ c = [\![e]\!]_d \ \text{in} \\
&\quad\ \{\text{obj} = c.\text{obj}, \text{ifc} = c.\text{ifc}, \text{strong} = c.\text{strong} \ominus \iota\} \\
[\![\text{ref}_\varphi e]\!]_d &= \ \text{let}\ x = \text{ref}\ [\![e]\!]_d \ \text{in} \\
&\quad\ \text{let}\ o = \{\text{get} = \lambda y.!x, \text{set} = \lambda y.x := y\} \ \text{in} \\
&\quad\ \{\text{obj} = o, \text{ifc} = \hat{\varphi}, \text{strong} = \bar{\varnothing}\} \\
[\![\text{let}\ x = e_1 \ \text{in}\ e_2]\!]_d &= \text{let}\ x = [\![e_1]\!]_d \ \text{in}\ [\![e_2]\!]_d
\end{aligned}
$$

Figure 3: The pop-to-pml term transformation

ments in the target language, eliminating the overhead of a type soundess proof entirely. The technique also eases development of a *direct* pop type system— that is, where pop expressions are treated directly, rather than through transformation. This is because safety in a direct system can be demonstrated via a simple proof of correspondance between the direct and indirect type systems, rather than through the usual (and complicated) route of subject reduction. This technique has been used to good effect in previous static treatments of languages supporting stack-inspection security [12], information flow security [11], and elsewhere [15].

While we focus on the logical type system in this presentation, we will briefly describe how the transformational approach has benefits for type *inference*, allowing an algorithm to be developed using existing, efficient methods.

4.1 The pop-to-pml transformation

The target language of the transformation is pml [18, 19], a calculus of extensible records based on Rémy's Projective ML [13], and equipped with references, sets and set operations. The language pml allows definition of records with default values $\{v\}$, where for every label a we have $\{v\}.a = v$. Records r may be modified with the syntax $r\{a = v\}$, such that $(r\{a = v\}).a = v$ and $(r\{a = v\}).a' = r.a'$ for all other a'.

The language also allows definition of finite sets B of atomic identifiers b chosen from a countably infinite set $\mathcal{L}_b$, and cosets $\bar{B}$. This latter feature presents some practical implementation issues, but in this presentation we take it at mathematical face value— that is, as the countably infinite set $\mathcal{L}_b - B$. The language also contains set operations $\ni$, $\vee$, $\wedge$ and $\ominus$, which are membership check, union, intersection and difference operations, respectively.

The pop-to-pml term transformation is given in Fig. 3. For brevity in the transformation we define the following syntactic sugar:

$$
\begin{gathered}
\{m_1 = e_1, \ldots, m_n = e_n\} \\
\triangleq \\
\{\varnothing\}\{m_1 = e_1\} \cdots \{m_n = e_n\}
\end{gathered}
$$

$$
\begin{array}{rcll}
\tau & ::= & \alpha, \beta, \ldots \mid \tau \to \tau \mid \{\tau\} \mid a : \tau \,;\, \tau \mid \partial\tau \mid b\tau, \tau \mid \varnothing \mid \omega \mid \tau \text{ ref} \mid c & \textit{types} \\
c & ::= & + \mid - & \textit{constructors}
\end{array}
$$

Figure 4: pml type grammar

$$
\frac{\alpha \in \mathcal{V}_k}{\alpha : k} \qquad \frac{\tau : \mathit{Type}}{\tau \text{ ref} : \mathit{Type}} \qquad \frac{\tau, \tau' : \mathit{Type}}{\tau \to \tau' : \mathit{Type}} \qquad \frac{\tau : \mathit{Row}_{\varnothing}, \mathit{Set}_{\varnothing}}{\{\tau\} : \mathit{Type}} \qquad \varnothing : \mathit{Set}_B \qquad \omega : \mathit{Set}_B \qquad c : \mathit{Con}
$$

$$
\frac{\tau : \mathit{Con} \qquad b \notin B \qquad \tau' : \mathit{Set}_{B \cup \{b\}}}{(b\tau, \tau') : \mathit{Set}_B} \qquad \frac{\tau : \mathit{Type}}{\partial\tau : \mathit{Row}_A} \qquad \frac{\tau : \mathit{Type} \qquad a \notin A \qquad \tau' : \mathit{Row}_{A \cup \{a\}}}{(a : \tau \,;\, \tau') : \mathit{Row}_A}
$$

Figure 5: Kinding rules for pml types

$$
\begin{array}{rcll}
\mathsf{fix}\,\varsigma.\lambda\_.e & \triangleq & \mathsf{fix}\,\varsigma.\lambda x.e & x \text{ not free in } e \\
e_1; e_2 & \triangleq & \text{let } x = e_1 \text{ in } e_2 & x \text{ not free in } e_2
\end{array}
$$

The translation is effected by transforming pop objects into rows with obj fields containing method transformations, ifc fields containing interface transformations, and strong fields containing sets denoting methods on which the object is *not* weak. Interface definitions φ are encoded as records $\hat{\varphi}$ with fields indexed by domain names; elevated rows are used to ensure that interface checks are total in the image of the transformation.

Of technical interest is the use of lambda abstractions with recursive binding mechanisms in pml— of the form $\mathsf{fix}\, z.\lambda x.e$, where z binds to $\mathsf{fix}\, z.\lambda x.e$ in e— to encode the self variable ς in the transformation. Also of technical note is the manner in which weakenings are encoded. In a pop weakened object $\mathbf{weak}_\iota(o)$, the set ι denotes the methods which are inaccessible via weakening. In the encoding these sets are turned "inside out", with the strong field in objects denoting the fields which *are* accessible. We define the translation in this manner to allow a simple, uniform treatment of set subtyping in pop; the following section elaborates on this.

The correctness of the transformation is established by the simple proof of the following theorem:

Theorem 4.1 (Transformation correctness) *If e evaluates to v then $[\![e]\!]_{d_1}$ evaluates to $[\![v]\!]_{d_1}$. If e diverges then so does $[\![e]\!]_{d_1}$. If e goes wrong then $[\![e]\!]_{d_1}$ goes wrong.*

4.2 Types for pop

A sound polymorphic type system for pml is obtained in a straightforward manner as an instantiation of $\mathrm{HM}(X)$ [9, 17], a constraint-based polymorphic type framework. Type judgements in $\mathrm{HM}(X)$ are of the form $C, \Gamma \vdash e : \sigma$, where C is a type constraint set, Γ is a typing environment, and σ is a polymorphic type scheme. The instantiation consists of a type language including row types [13] and a specialized language of *set* types, defined in Fig. 4. To ensure that only meaningful types can be built, we immediately equip this type language with *kinding* rules, defined in Fig. 5, and hereafter consider only well-kinded types. Note in particular that these kinding rules disallow duplication of record field and set element labels.

Set types behave in a manner similar to row types, but have a succinct form more appropriate for application to sets. In fact, set types have a direct interpretation as a particular form of row types [19], which is omitted here for brevity. The field constructors $+$ and $-$ denote whether a set element is present or absent, respectively. The set types $\varnothing$ and ω behave similarly to the uniform row constructor $\partial\tau$; the type $\varnothing$ (resp. ω) specifies that all other elements not explicitly mentioned in the set type are absent (resp. present). For example, the set $\{b_1, b_2\}$ has type $\{b_1+, b_2+, \varnothing\}$, while $\{\overline{b_1, b_2}\}$ has type $\{b_1-, b_2-, \omega\}$. The use of element and set variables γ and β allows for fine-grained polymorphism over set types.

Syntactic type safety for pml is easily established in the $\mathrm{HM}(X)$ framework [17]. By virtue of this property and Theorem 4.1, a sound, indirect static analysis for pop is immediately obtained by composition of the pop–to–pml transformation and pml type judgments:

Theorem 4.2 (Indirect type safety) *If e is a closed pop expression and $C, \Gamma \vdash [\![e]\!]_{d_1} : \sigma$ is valid, then e does not go wrong.*

While this indirect type system is a sound static analysis for pop, it is desirable to define a direct static analysis for pop. The term transformation required for the indirect analysis is an unwanted complication for compilation, the indirect type system is not a clear declaration of program properties for the programmer, and type error reporting would be extremely troublesome. Thus, we define a direct type system for pop, the development of which significantly benefits from the transformational approach. In

$$\tau ::= \alpha, \beta, \ldots \mid \{\tau\} \mid [\tau]^{\tau}_{\tau} \mid m : \tau \to \tau\ ;\ \tau \mid d : \tau\ ;\ \tau \mid m, \tau \mid \epsilon$$

Figure 6: Direct pop type grammar

$$\frac{\alpha \in \mathcal{V}_k}{\alpha : k} \qquad \epsilon : \mathit{Meth}_M, \mathit{Set}_M, \mathit{Ifc}_D \qquad \frac{m \notin M \qquad \tau : \mathit{Set}_{M \cup \{m\}}}{m, \tau : \mathit{Set}_M} \qquad \frac{\tau_1 : \mathit{Set}_\varnothing \qquad d \notin D \qquad \tau : \mathit{Ifc}_{D \cup \{d\}}}{d : \{\tau_1\}\ ;\ \tau_2 : \mathit{Ifc}_D}$$

$$\frac{\tau_1 : \mathit{Type} \qquad \tau_2 : \mathit{Type} \qquad m \notin M \qquad \tau : \mathit{Meth}_{M \cup \{m\}}}{m : \tau_1 \to \tau_2\ ;\ \tau : \mathit{Meth}_M} \qquad \frac{\tau_1 : \mathit{Meth}_\varnothing \qquad \tau_2 : \mathit{Ifc}_\varnothing \qquad \tau_3 : \mathit{Set}_\varnothing}{[\tau_1]^{\{\tau_2\}}_{\{\tau_3\}} : \mathit{Type}}$$

Figure 7: Direct pop type kinding rules

particular, type safety for the direct system may be demonstrated by a simple appeal to safety in the indirect system, rather than *ab initio*.

The direct type language for pop is defined in Fig. 6. We again ensure the construction of only meaningful types via kinding rules, defined in Fig. 7, hereafter considering only well-kinded pop types. The most novel feature of the pop type language is the form of object types $[\tau]^{\{\tau_1\}}_{\{\tau_2\}}$, where τ_2 is the type of any weakening set imposed on the object, and τ_1 is the type of its interface. Types of sets are essentially the sets themselves, modulo polymorphic features; we abbreviate a type of the form $\tau; \epsilon$ or τ, ϵ as τ.

The close correlation between the direct and indirect type system begins with the type language: types for pop have a straightforward interpretation as pml types, defined in Fig. 8. This interpretation is extended to constraints and typing environments in the obvious manner. In this interpretation, we turn weakening sets "inside-out", in keeping with the manner in which weakening sets are turned inside-out in the pop-to-pml term transformation. The benefit of this approach is with regard to subtyping: weakening sets can be safely strengthened, and user interfaces safely weakened, in a uniform manner via subtyping coercions.

The direct type judgement system for pop, the rules for which are *derived* from pml type judgements for transformed terms, is defined in Fig. 9. Note that subtyping in the direct type system is defined in terms of the type interpretation, where $C_1 \Vdash C_2$ means that every solution of C_1 is also a solution of C_2. The following definition simplifies the statement of the SEND rule:

Definition 4.1 *$C \Vdash m \notin \tau_w$ holds iff $(\!| C |\!) \nVdash \exists\phi.((\!| \tau_w |\!)_+ \leq (\!| m, \phi |\!)_+)$ holds, where $\phi \notin \mathrm{fv}(C, \tau_w)$.*

The easily proven, tight correlation between the indirect and direct pop type systems is clearly demonstrated via the following lemma:

Lemma 4.1 *$d, C, \Gamma \vdash e : \tau$ is valid iff $(\!| C |\!), (\!| \Gamma |\!) \vdash [\![e]\!]_d : (\!| \tau |\!)$ is.*

And in fact, along with Theorem 4.1, this correlation is sufficient to establish direct type safety for pop:

Theorem 4.3 (Direct type safety) *If e is a closed pop expression and $d, C, \Gamma \vdash e : \tau$ is valid, then e does not go wrong.*

This result demonstrates the advantages of the transformational method, which has allowed us to define a direct, expressive static analysis for pop with a minimum of proof effort.

An important consequence of this result is the implication that certain *optimizations* may be effected in the pop operational semantics, as a result of the type analysis. In particular, since the result shows that any well-typed program will be operationally safe, or *secure*, the various run-time security checks— i.e. those associated with the *send*, *cast*, *weaken*, *set* and *get* rules— may be eliminated entirely.

4.2.1 Type inference

The transformational method allows a similarly simplified approach to the development of type inference. The $\mathrm{HM}(X)$ framework comes equipped with a type inference algorithm modulo a constraint normalization procedure (constraint normalization is the same as constraint satisfaction, e.g. *unification* is a normalization procedure for equality constraints). Furthermore, efficient constraint normalization procedures have been previously developed for row types [10, 14], and even though set types are novel, their interpretation as row types [19] allows a uniform implementation. This yields a type inference algorithm for pml in the $\mathrm{HM}(X)$ framework. An indirect inference analysis for pop may then be immediately obtained as the composition of the pop-to-pml transformation and pml type inference.

Furthermore, a direct type inference algorithm can be derived from the indirect algorithm, just as direct type judgements can be derived from indirect judgements. Only

$$
\begin{aligned}
(\!| [\tau_1]^{\{\tau_2\}}_{\{\tau_3\}} |\!) &= \{\mathrm{obj} : \{(\!| \tau_1 |\!)\}\,;\ \mathrm{ifc} : \{(\!| \tau_2 |\!)\}\,;\ \mathrm{strong} : \{(\!| \tau_3 |\!)_-\}\,;\ \partial\{\varnothing\}\} \\
(\!| m : \tau_1 \to \tau_2\,;\ \tau |\!) &= m : (\!| \tau_1 |\!) \to (\!| \tau_2 |\!)\,;\ (\!| \tau |\!) \\
(\!| d : \{\tau_1\}\,;\ \tau_2 |\!) &= d : \{(\!| \tau_1 |\!)_+\}\,;\ (\!| \tau_2 |\!) \\
(\!| \epsilon |\!) &= \partial\{\varnothing\} \\
(\!| \beta |\!), (\!| \beta |\!)_+, (\!| \beta |\!)_- &= \beta \\
(\!| m, \tau |\!)_+ &= m+, (\!| \tau |\!)_+ \\
(\!| \epsilon |\!)_+ &= \varnothing \\
(\!| m, \tau |\!)_- &= m-, (\!| \tau |\!)_- \\
(\!| \epsilon |\!)_- &= \omega
\end{aligned}
$$

Figure 8: The pop-to-pml type transformation

the syntactic cases need be adopted, since efficient constraint normalization procedures for row types may be reused in this context— recall that the direct pop type language has a simple interpretation in the pml type language.

5 Using pop

By choosing different naming schemes, a variety of security paradigms can be effectively and reliably expressed in pop. One such scheme enforces a strengthened meaning of the `private` and `protected` modifiers in class definitions, a focus of other communication-based capability type analyses [4, 21]. As demonstrated in [21], a `private` field can leak by being returned by reference from a `public` method. Here we show how this problem can be addressed in a use-based model. Assume the following Java-like pseudocode package p, containing class definitions c_1, c_2, and possibly others, where c_2 specifies a method m that leaks a `private` instance variable:

```
package p begin

  class c1 {                class c2 {
    public :                  public :
      f(x) = x                  m(x) = b
    private :                   a = new c1
      g(x) = x                private :            ...
    protected :                 b = new c1
      h(x) = x                protected :
  }                             c = new c1
                            }
end
```

We can implement this definition as follows. Interpreting domains as class names in pop, let p denote the set of all class names $c_1, \ldots, c_n$ in package p, and let $p \mapsto \iota$ be syntactic sugar for $c_1 \mapsto \iota_1, \ldots, c_n \mapsto \iota_n$. Then, the appropriate interface for objects in the encoding of class c_1 is as follows:

$$\varphi_1 \triangleq \{p \mapsto \{f, h\}, \partial \mapsto \{f\}\}$$

(Recall that all objects automatically have full access to themselves, so full access for c_1 need not be explicitly stated). The class c_1 can then be encoded as an object *factory*, an object with only one publicly available method that returns new objects in the class, and some arbitrary label d:

$$
\begin{aligned}
o_1 &\triangleq [f(x) = x,\ g(x) = x,\ h(x) = x] \cdot c_1 \cdot \varphi_1 \\
\mathrm{fctry}_{c_1} &\triangleq [\mathrm{new}(x) = o_1] \cdot d \cdot \{\partial \mapsto \{\mathrm{new}\}\}
\end{aligned}
$$

To encode c_2, we again begin with the obvious interface definition for objects in the encoding of class c_2:

$$\varphi_2 \triangleq \{p \mapsto \{m, a, c\}, \partial \mapsto \{m, a\}\}$$

However, we must now encode *instance variables*, in addition to methods. In general, this is accomplished by encoding instance variables a containing objects as methods $a()$ that return references to objects. Then, any selection of a is encoded as $a()$.get(), and any update with v is encoded $a()$.set(v). By properly constraining the interfaces on these references, a "Java-level" of modifier enforcement can be achieved; but casting the interfaces of stored objects *extends* the security, by making objects *unusable* outside the intended domain. Let $e \mid (\{d_1, \ldots, d_n\}, \iota)$ be sugar for $e \mid (d_1, \iota) \mid \cdots \mid (d_n, \iota)$. Using fctry_{c_1}, we may create a `public` version of an object equivalent to o_1, without any additional constraints on its confinement, as follows:

$$o_a \triangleq \mathrm{fctry}_{c_1}.\mathrm{new}()$$

Letting $p' = p - \{c_2\}$, we may create a version of an object equivalent to o that is `private` with respect to the encoding of class c_2, using casts as follows:

$$o_b \triangleq (\mathrm{fctry}_{c_1}.\mathrm{new}()) \mid (\partial, \varnothing) \mid (p', \varnothing)$$

Default

$$\vdash \{\partial \mapsto \iota\} : (\partial : \iota)$$

Interface

$$\frac{\vdash \varphi : \tau}{\vdash \varphi \oplus d \mapsto \iota : (d : \iota\, ;\ \tau)}$$

Var

$$\frac{\Gamma(x) = \sigma \qquad (\!| C |\!) \Vdash (\!| \sigma |\!)}{d, C, \Gamma \vdash x : \sigma}$$

Sub

$$\frac{d, C, \Gamma \vdash e : \tau \qquad (\!| C |\!) \Vdash (\!| \tau |\!) \leq (\!| \tau' |\!)}{d, C, \Gamma \vdash e : \tau'}$$

Let

$$\frac{d, C, \Gamma \vdash v : \sigma \qquad d, C, (\Gamma; x : \sigma) \vdash e : \tau}{d, C, \Gamma \vdash \mathrm{let}\, x = v \,\mathrm{in}\, e : \tau}$$

$\forall$ Intro

$$\frac{d, C \wedge D, \Gamma \vdash e : \tau \qquad \bar{\alpha} \cap \mathrm{fv}(C, \Gamma) = \varnothing}{d, C \wedge \exists \bar{\alpha}.D, \Gamma \vdash e : \forall \bar{\alpha}[D].\tau}$$

$\forall$ Elim

$$\frac{d, C, \Gamma \vdash e : \forall \bar{\alpha}[D].\tau' \qquad (\!| C |\!) \Vdash (\!| [\bar{\tau}/\bar{\alpha}]D |\!)}{d, C, \Gamma \vdash e : [\bar{\tau}/\bar{\alpha}]\tau'}$$

Ref

$$\frac{d, C, \Gamma \vdash e : \tau \qquad \vdash \varphi : \tau_\varphi}{d, C, \Gamma \vdash \mathrm{ref}_\varphi e : [\mathrm{set} : \tau \to \tau, \mathrm{get} : \tau' \ \to \tau]^{\tau_\varphi}_{\{\epsilon\}}}$$

Obj

$$\frac{\vdash \varphi : \tau_\varphi \qquad \tau'_\varphi = d' : \{m_i\ ^{0<i\leq n}\} \qquad (d', C, \Gamma; x : \tau_j; \varsigma : [m_i : \tau_i \to \tau'_i\ ^{0<i\leq n}]^{\{\tau'_\varphi\}}_{\{\epsilon\}} \vdash e_j : \tau'_j)\ ^{0<j\leq n}}{d, C, \Gamma \vdash [m_i(x) = e_i\ ^{0<i\leq n}] \cdot d' \cdot \varphi : [m_i : \tau_i \to \tau'_i\ ^{0<i\leq n}]^{\{\tau_\varphi\}}_{\{\epsilon\}}}$$

Send

$$\frac{d, C, \Gamma \vdash e_1 : [m : \tau' \to \tau^{\tau_\varphi}_{\{\tau_w\}}\ ;\ \tau_1]^{\{d:\{m,\ \tau_2\}\ ;\ \tau_3\}}_{\{\tau_w\}} \qquad d, C, \Gamma \vdash e_2 : \tau' \qquad C \Vdash m \notin \tau_w}{d, C, \Gamma \vdash e_1.m(e_2) : \tau^{\tau_\varphi}_{\{\tau_w\}}}$$

Cast

$$\frac{d, C, \Gamma \vdash e : [\tau]^{\{d':\{m_i\ ^{0<i\leq n}\ ;\ \tau_1\}\ ;\ \tau_2\}}_{\tau_w} \qquad \iota = \{m_i\ ^{0<i\leq n}\}}{d, C, \Gamma \vdash e{\restriction}(d', \iota) : [\tau]^{\{d':\iota\ ;\ \tau_2\}}_{\tau_w}}$$

Weak

$$\frac{d, C, \Gamma \vdash e : \tau^{\tau_\varphi}_{\{m_i\ ^{0<i\leq n}, \tau'\}} \qquad \iota = \{m_i\ ^{0<i\leq n}\}}{d, C, \Gamma \vdash \mathbf{weak}_\iota(e) : \tau^{\tau_\varphi}_{\{m_i\ ^{0<i\leq n}, \tau'\}}}$$

Figure 9: Direct type system for pop

We may create a version of an object equivalent to o that is `protected` with respect to the encoding of package p, as follows:

$$o_c \triangleq (\mathrm{fctry}_{c_1}.\mathrm{new}())\upharpoonright(\partial, \varnothing)$$

Let o_2 be defined as follows:

$$\begin{array}{lll} o_2 & \triangleq & \mathrm{let}\, r_a = \mathrm{ref}_{\{\partial \mapsto \{\mathrm{set},\mathrm{get}\}\}}\, o_a \,\mathrm{in} \\ & & \mathrm{let}\, r_b = \mathrm{ref}_{\{c_1 \mapsto \{\mathrm{set},\mathrm{get}\}\}}\, o_b \,\mathrm{in} \\ & & \mathrm{let}\, r_c = \mathrm{ref}_{\{c_1 \mapsto \{\mathrm{set},\mathrm{get}\}, p \mapsto \{\mathrm{set},\mathrm{get}\}\}}\, o_c \,\mathrm{in} \\ & & \quad [\ m(x) = \varsigma.b().\mathrm{get}(), \\ & & \quad\ \ a(x) = r_a, \\ & & \quad\ \ b(x) = r_b, \\ & & \quad\ \ c(x) = r_c\ ] \cdot c_2 \cdot \varphi_2 \end{array}$$

Then fctry_{c_2} is encoded, similarly to fctry_{c_1}, as:

$$\mathrm{fctry}_{c_2} \triangleq [\mathrm{new}(x) = o_2] \cdot d \cdot \{\partial \mapsto \{\mathrm{new}\}\}$$

Given this encoding, if an object stored in b is leaked by a non-local use of m, it is unuseable. This is the case because, even though a non-local use of m will return b, in the encoding this return value explicitly states it cannot be used outside the confines of c_2; as a result of the definition of φ_1 and casting, the avatar o_b of b in the encoding has an interface equivalent to:

$$\{c_2 \mapsto \{f, h\}, p' \mapsto \varnothing, \partial \mapsto \varnothing\}$$

While the communication-based approach accomplishes a similar strengthening of modifier security, the benefits of greater flexibility may be enjoyed via the use-based approach. For example, a `protected` reference can be safely passed outside of a package and then back in, as long as a use of it is not attempted outside the package. Also for example are the fine-grained interface specifications allowed by this approach, enabling greater modifier expressivity— e.g. publicly read-only but privately read/write instance variables.

6 Conclusion

As shown in [1], object confinement is an essential aspect of securing OO programming languages. Related work on this topic includes the *confinement types* of [21], which have been implemented as an extension to Java [3]. The mechanism is simple: classes marked **confined** must not have references escape their defining package. Most closely related are the *ownership types* of [4]. In fact, this system can be embedded in ours, albeit with a different basis for enforcement: as discussed in Sect. 2, these previous type approaches treat a communication-based mechanism, whereas ours is use-based. One of the main points of our paper is the importance of studying the use-based approach as an alternative to the communication-based approach. Furthermore, our type system is polymorphic, with inference methods readily available due to its basis in row types.

Topics for future work include an extension of the language to capture *inheritance*, an important OO feature that presents challenges for type analysis. Also, we hope to study capability *revocation*.

In summary, contributions of this work include a focus on the more expressive use-based security model, the first type-based characterization of weak capabilities, and a general mechanism for fine-grained, use-based security specifications that includes flexible domain naming, precise object interface definitions, and domain-specific interface casting. Furthermore, we have defined a static analysis that enforces the security model, with features including flexibility due to polymorphism and subtyping, declarative benefits due to readability, and ease of proof due to the use of transformational techniques.

References

[1] Anindya Banerjee and David Naumann. Representation independence, confinement and access control. In *Conference Record of POPL02: The 29TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 166–177, Portland, OR, January 2002.

[2] Borris Bokowski and Jan Vitek. Confined types. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on ObjectOriented Programming Systems, Languages, and Applications (OOPSLA)*, November 1999.

[3] Ciaran Bryce and Jan Vitek. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, 1999.

[4] David Clarke, James Noble, and John Potter. Simple ownership types for object containment. In *ECOOP'01 — Object-Oriented Programming, 15th European Conference*, Lecture Notes in Computer Science, Berlin, Heidelberg, New York, 2001. Springer.

[5] Mark Miller *et. al.* The E programming language. URL: `http://www.erights.org`.

[6] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *1998 USENIX Annual Technical Conference*, pages 259–270, New Orleans, LA, 1998.

[7] C. Hawblitzel and T. von Eicken. Type system support for dynamic revocation, 1999. In ACM SIGPLAN Workshop on Compiler Support for System Software, May 1999.

[8] Richard Y. Kain and Carl E. Landwehr. On access checking in capability-based systems. *IEEE Transactions on Software Engineering*, 13(2):202–207, February 1987.

[9] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999.

[10] François Pottier. A versatile constraint-based type inference system. *Nordic Journal of Computing*, 7(4):312–347, November 2000.

[11] François Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of the the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, pages 46–57, September 2000.

[12] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. In David Sands, editor, *Proceedings of the 10th European Symposium on Programming (ESOP'01)*, volume 2028 of *Lecture Notes in Computer Science*, pages 30–45. Springer Verlag, April 2001.

[13] Didier Rémy. Projective ML. In *1992 ACM Conference on Lisp and Functional Programming*, pages 66–75, New-York, 1992. ACM Press.

[14] Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, INRIA, 1993.

[15] Didier Rémy. Typing record concatenation for free. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[16] Jonathan Shapiro and Samuel Weber. Verifying the EROS confinement mechanism. In *21st IEEE Computer Society Symposium on Research in Security and Privacy*, 2000.

[17] Christian Skalka. Syntactic type soundness for HM(X). Technical report, The Johns Hopkins University, 2001.

[18] Christian Skalka. *Types for Programming Language-Based Security*. PhD thesis, The Johns Hopkins University, 2002.

[19] Christian Skalka and Scott Smith. Set types and applications. In *Workshop on Types in Programming (TIP02)*, 2002. To appear.

[20] L. van Doorn, M. Abadi, M. Burrows, and E. Wobber. Secure network objects. In *IEEE Symposium on Security and Privacy*, May 1996.

[21] Jan Vitek and Boris Bokowski. Confined types in java. *Software—Practice and Experience*, 31(6):507–532, May 2001.