Reasoning about Safety Properties in a JVM-like Environment

Philip W. L. Fong

Technical Report CS-2006-02
February 2006

Department of Computer Science
University of Regina
Regina, Saskatchewan, S4S 0A2
Canada

ISBN 0-7731-0557-3 (print)
ISBN 0-7731-0558-1 (online)
ISSN 0828-3494

Reasoning about Safety Properties in a JVM-like Environment

Philip W. L. Fong
Department of Computer Science
University of Regina
Regina, Saskatchewan, Canada S4S 0A2
pwlfong@cs.uregina.ca

Abstract

Type-based protection mechanisms in a JVM-like environment must be administrated by the code consumer at the bytecode level. Unfortunately, formulating a sound static type system for the full JVM bytecode language can be a daunting task. It is therefore counter-productive for the designer of a bytecode-level type system to address the full complexity of the VM environment in the early stage of design.

In this work, a lightweight modeling tool, Featherweight JVM, is proposed to facilitate the early evaluation of bytecode-level, type-based protection mechanisms. In the style of Security Automata, Featherweight JVM is an event model that tracks interprocedural access events generated by a JVM-like environment. The effect of deploying a type-based protection mechanism can be modeled by a safety policy that restricts the event sequences produced by the VM model. To evaluate the effectiveness of the protection mechanism, security theorems in the form of state invariants can then be proven in the policy-guarded VM model. This paper provides first evidence on the utility of this approach in providing early feedback to the designer of type-based protection mechanisms for JVM-like environments.

1. Introduction

Static type systems have been proposed in recent years for the Java programming language [19] or its derivatives [10] in order to enforce access control and confinement properties [34, 20, 36, 37, 1, 2, 3, 16, 30, 35]. These type systems are usually designed for the source language, and intended to be enforced by the code producer at compile time. In many cases, the soundness of the type analyses is evaluated in a high-level core calculus, such as Featherweight Java (FJ) [23], that captures the essence of the source language. An objection to this approach is that, in a Java-like platform, in which code units are shipped and dynamically loaded as bytecode, program verification that is performed against source code, or administrated only by the code producer, cannot be trusted [28, 27]. In order for the code consumer to enforce the typing discipline, it must be preserved at the level of the Java Virtual Machine (JVM) bytecode language [26]. Unfortunately, because of the lack of structured control flow, especially in the presence of subroutines that do not conform to the last-in-first-out discipline, and also because of the complexity of data flow between the local variable array and the operand stack, formulating sound static analysis at the bytecode level is a daunting task [31]. Formal verification, however, has proven to be necessary when one works at the bytecode level, as security holes in early implementation of the JVM bytecode verifier were uncovered by formalizing type systems for non-trivial fragments of the JVM bytecode language [33, 17, 18]. Still, it is counter-productive for the designer of a bytecode-level type system to address the full complexity of the JVM execution environment in the early stage of design.

In this work, a lightweight modeling tool, *Featherweight JVM*, is proposed for evaluating bytecode-level, type-based protection mechanisms in an early design stage. In the style of Security Automata [32, 11], Featherweight JVM (or FJVM) is an event model that tracks interprocedural access events generated by a JVM-like environment, and simulates their effects on the global VM state. Specifically, Featherweight JVM captures the following aspects of a JVM-like run time:

- FJVM **abstracts away** the complex structure of intraprocedural control flow (e.g., subroutines, unstructured branching, etc), but **explicitly models** the evolution of the run-time stack.
- FJVM **abstracts away** data flow between the local variable array and operand stack, but **explicitly models** data dependencies via type-based instrumentation.
- FJVM **abstracts away** destructive updates, but **explicitly models** reachability (i.e., may reach) and confine-

ment (i.e., must not reach) in a (possibly cyclic) object graph.

- FJVM **abstracts away** the concrete identity of methods and fields, but **explicitly models** execution context and link context.

The basic VM model can be customized by a domain-specific *safety policy*, which further constrains the access event sequences produced by the VM. Adopting the information restriction approach of [11], such a policy may only control access by consuming information made available by type instrumentation and static annotation. Safety policies thus formulated model the effect of imposing a type-based protection mechanism on the run-time environment. To evaluate the effectiveness of the target protection mechanism, security theorems in the form of state invariants can then be proven for the policy-guarded VM model. Although such a modeling exercise does not establish the soundness of a bytecode-level type system, it provides a manageable formal model for articulating the structure of the would-be type system and the shape of its soundness proof, and does so without overwhelming the designer with the full complexity of the JVM bytecode language. The question answered by Featherweight JVM is not "Is the type system sound?", but rather "Suppose my type-based protection mechanism indeed restricts the VM in a certain way, what security theorems can I establish?"

To demonstrate the utility of the proposed approach, a safety policy for Vitek *et al*'s Confined Types [34, 20, 36, 37] has been formulated in Featherweight JVM, and a corresponding confinement theorem for the safety policy has been established. To reduce the amount of dynamic class loading required for type checking, the safety policy has been reformulated to reflect a capability-based implementation of Confined Types. The reformulation has been shown to preserve the confinement theorem. These results provide first evidence on the utility of Featherweight JVM as an early modeling tool for evaluating type-based protection mechanisms in a JVM-like environment.

2. Related Work

This work is related to various efforts in the formal modeling of the JVM and its bytecode verification process. Due to the vastness of the literature on this topic, the reader is referred to Hartel and Moreau's comprehensive survey [22] and the special issue of *Journal of Automated Reasoning* on Java Bytecode Verification [29]. The goal of this work, however, differs from previous efforts. While previous work aims at providing increasingly accurate and comprehensive models of the JVM, the present work strives to distill the essence

This work is similar in its goal to calculi such as FJ [23] and MJ [5], that is, to provide a manageable formal model for studying new language features. In fact, a design criterion of FJVM has been the following: "How do we expose enough details of the JVM to facilitate the reasoning of access control and confinement properties, while keeping the complexity of the resulting model similar to that of FJ?" Thus, inspired by FJ, FJVM is "functional" in the sense that destructive updates to object fields is not modeled. Links may be introduced by the **put** event, but thereafter immutable. Again, like FJ, FJVM focuses on a few interprocedural access events, such as object initialization, field access, dynamic method dispatching, and dynamic type casting. However, FJVM differs from FJ in important aspects. The operational semantics of FJ models the reduction of terms. As FJ terms are inductively specified, the resulting object graph is acyclic. FJVM execution, however, can spawn object graphs of arbitrary topology. Reachability and confinement properties thus obtained are more general and natural. Accessibility invariants for FJ are sometimes expressed in terms of a relation between a term and its subterms [36, 37]. In FJVM they are expressed as properties of the run-time stack.

Featherweight JVM can be seen as an instance of Security Automata [32]. The latter and its variants [4, 24, 25, 11] have been employed to characterize security policies. Fong [11] characterizes safety policies by the kind of information consumed by their enforcing protection mechanisms. This work is an attempt to employ the same information restriction principle to model the effect of imposing a type-based protection mechanism on JVM-like environments. We achieve this by restricting safety policies to only consume information made available by a specific style of type instrumentation and static annotation. The resulting model does not account for all possible type analyses, as that has never been our goal. On the contrary, [21] attempts to characterize the classes of all safety policies enforceable by various enforcement mechanisms, including static analysis as a special case.

This work is also related to Foster *et al*'s theory of type qualifiers [14, 15], in that FJVM attempts to provide a framework in which to evaluate augmented type systems at the bytecode level. While FJVM does not mandate any type-theoretic structure on its instrumentation process and safety policies, Foster *et al*'s work articulates a well thought out theory on a family of qualifier-based augmented type systems. This work may be seen as a first step to the development of an analogous theory for the JVM bytecode language (see Sect. 6).

Confined Types [34, 20, 36, 37] is originally proposed to provide a stronger measure of encapsulation than what is available in the standard Java type system. The design goal is to avoid software vulnerabilities caused by reference

$$\frac{\Phi \vdash r : C \qquad C <: B}{\Pi, \Gamma; \Phi, A, \sigma \to \Pi, \Gamma; \Phi \cup \{r : B\}, A, \sigma}\ (\textsc{Widen})$$

$$\frac{r \text{ is a fresh object reference from } \mathcal{O}}{\Pi, \Gamma; \Phi, A, \sigma \to \Pi \cup \{r : B\}, \Gamma; \Phi \cup \{r : B\}, A, \sigma}\ (\textsc{New})$$

$$\frac{\Phi \vdash r : C \qquad \Pi \vdash r : C' \qquad C' <: B}{\Pi, \Gamma; \Phi, A, \sigma \to \Pi, \Gamma; \Phi \cup \{r : B\}, A, \sigma}\ (\textsc{Cast})$$

$$\frac{\Phi \vdash p : B_0 \qquad B_0 <: B \qquad \Gamma \vdash p : B \leadsto q : C}{\Pi, \Gamma; \Phi, A, \sigma \to \Pi, \Gamma; \Phi \cup \{q : C\}, A, \sigma}\ (\textsc{Get})$$

$$\frac{\Phi \vdash p : B_0 \qquad B_0 <: B \qquad \Phi \vdash q : C}{\Pi, \Gamma; \Phi, A, \sigma \to \Pi, \Gamma \cup \{p : B \leadsto q : C\}; \Phi, A, \sigma}\ (\textsc{Put})$$

$$\frac{\begin{array}{ccc}\Phi \vdash r_0 : C_0 & C_0 <: B & \Phi \vdash \overline{r} : \overline{C} \\ \Pi \vdash r_0 : B'' & B'' <: B' & B' <: B\end{array}}{\begin{array}{c}\Pi, \Gamma; \Phi, A, \sigma \to \Pi, \Gamma; \Phi', B', \sigma' \\ \text{where } \Phi' = \{r_0 : B', \overline{r} : \overline{C}\} \\ \text{and } \sigma' = \mathit{push}(\Phi, A, C, \sigma)\end{array}}\ (\textsc{Invoke})$$

$$\frac{\Phi' \vdash r : C}{\begin{array}{l}\Pi, \Gamma; \Phi', B', \mathit{push}(\Phi, A, C, \sigma) \\ \qquad \to \Pi, \Gamma; \Phi \cup \{r : C\}, A, \sigma\end{array}}\ (\textsc{Return})$$

Figure 1. A Basic Model for the JVM

leakages. Our first safety policy (Fig. 5) closely mirrors the original formulation of Confined Types [36, 37]. The second, capability-based formulation (Fig. 7) is, to our best knowledge, original. The latter formulation renders type checking modular. A prototype of such a type checker has been implemented [13] in the framework of Pluggable Verification Modules [12]. The modeling exercise in Sect. 5.3 allowed us to uncover and correct some subtle bugs in the implementation.

3. A Basic Model of the JVM

3.1. The Model

A basic model of the JVM is given in Fig. 1. In the style of Security Automata [32, 11], the model is a non-deterministic production system that describes how the VM state evolves over time in reaction to access events. Non-determinism is employed because we are not modeling the execution of one particular bytecode program, but rather all possible access events that may be generated by the VM when well-typed bytecode sequences are executed.

Reference Types and Object References In the VM model of Fig. 1 are two kinds of entities — reference types and their instances. It is assumed that there is a set $\mathcal{C}$ of *raw reference types*[1], which are class and interface types, without genericity. Generic types are source level constructs that do not play a part in bytecode execution. Array types are not modeled, but can be added back to the model with ease. Metavariables $A, B, C \in \mathcal{C}$ denote typical instances of $\mathcal{C}$. To simplify the notation, we write $\overline{C}$ as a shorthand[2] for the list $C_1, \ldots, C_k$. As usual, $A <: B$ denotes the subtyping of reference types. It is also assumed that there is a set $\mathcal{O}$ of *object references*. Metavariables $p, q, r \in \mathcal{O}$ denote typical instances of object references. Every object reference r is an instance of a unique class C. An object may contain an arbitrary number of typed fields, some of which are inherited from the supertypes of its class. Each field in turn stores an object reference. Inspired by FJ, a field may only be instantiated once but never updated. The null reference is modeled by the absence of link. As well, fields are distinguishable only up to their declaring classes.

VM State A *VM state* is a configuration $\Pi, \Gamma; \Phi, A, \sigma$. The components to the left of the semicolon model the current state of the heap, while those to the right models the stack of the executing thread.

Heap The *object pool* Π is a finite set of *allocations* $r : C$. Intuitively, Π records all the objects that have been created by the VM, together with their classes. The judgment $\Pi \vdash r : C$ means that the allocation $r : C$ is a member of the object pool Π. The *link graph* Γ is a finite set of *links* $p : B \leadsto q : C$. A link $p : B \leadsto q : C$ records that the object p contains a field declared in reference type B, with field type C, storing the object reference q. The judgment $\Gamma \vdash p : B \leadsto q : C$ indicates that the link is a member of the link graph. Together Π and Γ models the global state of the heap.

Stack The VM state components Φ, A, σ model the stack of the current thread of execution. At the top of the stack is a stack frame Φ and an execution context A. A *stack frame* is a finite set of *labeled references* $r : C$. Such a set models the references accessible in a JVM stack frame. The internal structure of the JVM stack frame is not modeled, because the data flow between the local variable array and the operand stack is not our concern. As well, each reference r is associated with a type label C. We will return to this point in the following. The judgment $\Phi \vdash r : C$ asserts that

[1]Source-level generic types are erased during the compilation process, resulting in non-generic raw types at the bytecode level. User of this model may explicitly encode source-level generic types as type annotations (Sect. 4).

[2]Obvious variations of this notational shorthand shall be clear from the context. For example, we write $\overline{C}^{\overline{\gamma}}$ for the list $C_1^{\gamma_1}, \ldots, C_k^{\gamma_k}$, and $\overline{r} : \overline{C}$ for the list $r_1 : C_1, \ldots, r_k : C_k$.

the labeled reference $r : C$ is a member of Φ. The *execution context* A is the class in which the executing method is declared. Consequently, methods are distinguishable only up to their declaring classes. The last component σ models the call chain that leads to the current VM state. It has the following structure.

$$\sigma ::= \diamond \mid \mathit{push}(\Phi, A, C, \sigma)$$

Essentially, a *proper stack* is either an *empty stack*, $\diamond$, or a *non-empty stack*, $\mathit{push}(\Phi, A, C, \sigma)$, where Φ is the caller stack frame, A is the execution context of the caller (i.e., the class in which the caller method is declared), C is the declared return type of the callee method, and σ is another proper stack.

Instrumentation The VM model is *instrumented*. As mentioned before, every reference r stored in a stack frame Φ is tagged by a type label C, as in $\Phi \vdash r : C$. The basic type labeling as described here and the custom instrumentation as we will see in the next section share the following assumptions:

1. The VM is equipped with some form of type-based protection mechanism (e.g., a type system in the style of [33, 17, 18]) that assigns to every program point in an execution trace a *type state*.

2. The said protection mechanism ensures that only some "*safe*" subset of execution traces will be generated. This subset is described in terms of constraints over the type states of consecutive program points in execution traces.

Many of the antecedents of the transition rules in Fig. 1 are intended to model the screening effect of this protection mechanism. The goal of this paper, as pointed out in the introduction, is not to evaluate the soundness of the protection mechanism: i.e., whether it indeed generates the mentioned subset of execution traces and assigns the right type states to the program points in the traces. Instead, the goal of modeling here is to explore if the subset of execution traces as specified by the model preserves a given safety property. This paper argues that such a verification step is relatively lightweight and provides quick feedback to the designer of a protection mechanism before a full-scale soundness proof is attempted.

Transition Rules The transition rules in Fig. 1 define the state transition relation $\rightarrow$. The production (WIDEN) "promotes" the type label of an object reference in the stack frame. This rule does not model a physical VM event, but instead captures the standard notion of *subsumption*. The transition rule (NEW) creates a fresh object reference in the

$$\frac{\forall p, q, B, C\,.\,(\Pi \vdash p : B \;\wedge\; \Pi \vdash q : C) \Rightarrow (p \neq q \vee B = C)}{\mathit{SafeHeap}(\Pi)}$$

$$\frac{\begin{array}{c}\forall p, q, B, C, B'\,.\,(\Gamma \vdash p : B \rightsquigarrow q : C \;\wedge\; \Pi \vdash p : B') \Rightarrow B' <: B\\ \forall p, q, B, C, C'\,.\,(\Gamma \vdash p : B \rightsquigarrow q : C \;\wedge\; \Pi \vdash q : C') \Rightarrow C' <: C\end{array}}{\mathit{SafeLinks}(\Gamma \mid \Pi)}$$

$$\frac{\forall r, C, C'\,.\,(\Phi \vdash r : C \;\wedge\; \Pi \vdash r : C') \Rightarrow C' <: C}{\mathit{SafeFrame}(\Phi \mid \Pi)}$$

$$\frac{}{\mathit{SafeStack}(\diamond \mid \Pi)}$$

$$\frac{\mathit{SafeFrame}(\Phi \mid \Pi) \qquad \mathit{SafeStack}(\sigma \mid \Pi)}{\mathit{SafeStack}(\mathit{push}(\Phi, A, C, \sigma) \mid \Pi)}$$

$$\frac{\begin{array}{cc}\mathit{SafeHeap}(\Pi) & \mathit{SafeLinks}(\Gamma \mid \Pi)\\ \mathit{SafeFrame}(\Phi \mid \Pi) & \mathit{SafeStack}(\sigma \mid \Pi)\end{array}}{\mathit{SafeState}(\Pi, \Gamma; \Phi, A, \sigma)}$$

Figure 2. Type Safety Judgments

object pool, and makes that reference accessible in the top stack frame. The rule (CAST) models dynamic type casting, and tags an object reference in the stack frame with an alternative type label consistent with the actual class of the object reference. The production (GET) models field getting, and makes the target of an existing link accessible in the current stack frame if the source of that link is accessible. The production (PUT) models field setting, and creates a link between two object references accessible in the current stack frame. The transition rule (INVOKE) models dynamic method dispatching: the caller invokes a method declared in reference type B, with the actual dispatched method defined in reference type B'. The transition pushes a new stack frame (Φ'), passes the receiver ($r_0 : C_0$) and the arguments ($\overline{r} : \overline{C}$) from the caller stack frame (Φ) to the callee stack frame (Φ'), and constrains the return type (C). The transition rule (RETURN) pops the top stack frame (Φ'), resurrects the stack frame (Φ) of the caller (A), and makes the return value ($r : C$) available in the caller stack frame.

3.2. A Type Safety Invariant

This section demonstrates that the basic VM model preserves a notion of type safety: the type instrumentation in the model is always consistent with the actual class of allocated objects. To this end, a number of type safety judgments are defined in Fig. 2. The judgment *SafeState* asserts that a VM state is type safe. It is defined in terms of four auxiliary judgments. The *SafeHeap* judgment ensures that every object in the heap has a unique class. The *SafeLinks* judgment ensures that the links in the link graph are well formed, in the sense that an object only contains fields declared in, or inherited by, the class of the object, and that objects are stored only in compatibly typed fields. The pair of judgments *SafeFrame* and *SafeStack* ensure that the stack frames in the run-time stack contain only type labels consistent with the references they annotate. A notion of type soundness can be proven for the productions on the basic model.

Theorem 1 (One-Step Soundness)

$$\begin{array}{l} \forall\, \Pi, \Gamma, \Phi, A, \sigma, \Pi', \Gamma', \Phi', A', \sigma' \,. \\ \quad (\ \Pi, \Gamma; \Phi, A, \sigma \to \Pi', \Gamma'; \Phi', A', \sigma' \\ \quad\quad \wedge\ \mathit{SafeState}(\Pi, \Gamma; \Phi, A, \sigma)\ ) \\ \quad\quad\quad \Rightarrow \mathit{SafeState}(\Pi', \Gamma'; \Phi', A', \sigma') \end{array}$$

A proof of this theorem can be found in Appendix A.

4. Custom Instrumentation and Policy Enforcement

The basic VM model provides a framework in which to articulate the dynamic behavior of the JVM. Our original goal, however, is to evaluate the effect of constraining the VM behavior, and see if the constraints are strong enough to uphold a given security property. To this end, the basic VM model is elaborated into the *guarded VM model* of Fig. 3, which explicitly provides "*hooks*" for introducing behavioral constraints that are expressed in terms of domain-specific instrumentation.

Field and method designators In the basic VM model, fields and methods are distinguishable only up to their declaring classes. In some verification domains, one may desire further differentiation (e.g., based on static annotations). It is thus assumed that fields are partitioned into a finite or countably infinite number of equivalence classes (e.g., public, protected, package private, and private). Fields belonging to the same equivalence class are indistinguishable from the perspective of policy enforcement. The exact set of equivalence classes is domain-dependent. Each equivalence class is identified by a unique *field designator*. We postulate that there is a finite or countably infinite set $\mathcal{F}$ of field designators, and use metavariables f, g to denote its members. *Method designators* $m, n \in \mathcal{M}$ are defined in a similar manner.

A number of notational revisions are necessitated by the introduction of field and method designators. Firstly, links are now annotated by field designators, as in $p : B \overset{f}{\leadsto} q : C$. With annotated links, the transition rules (GET) and (PUT) in Fig. 3 are now aware of the designator of the fields that are being accessed. (When $|\mathcal{F}| = 1$ for a verification domain, the link annotation can be omitted.) Secondly, the execution context of a method is identified not only by the class in which the method is declared, but also its method

$$\frac{\Phi \vdash r : C^{\gamma} \qquad C <: B \qquad \textbf{widen}(C^{\gamma})(B^{\beta}) \in \Sigma[A.m]}{\Pi, \Gamma; \Phi, A.m, \sigma \to_{\Sigma} \Pi, \Gamma; \Phi \cup \{r : B^{\beta}\}, A.m, \sigma}\ (\textsc{Widen})$$

$$\frac{r \text{ is a fresh object reference from } \mathcal{O} \qquad \textbf{new}\langle B^{\beta}\rangle \in \Sigma[A.m]}{\Pi, \Gamma; \Phi, A.m, \sigma \to_{\Sigma} \Pi \cup \{r : B\}, \Gamma; \Phi \cup \{r : B^{\beta}\}, A.m, \sigma}\ (\textsc{New})$$

$$\frac{\Phi \vdash r : C^{\gamma} \qquad \Pi \vdash r : C' \qquad C' <: B \qquad \textbf{cast}\langle B^{\beta}\rangle(C^{\gamma}) \in \Sigma[A.m]}{\Pi, \Gamma; \Phi, A.m, \sigma \to_{\Sigma} \Pi, \Gamma; \Phi \cup \{r : B^{\beta}\}, A.m, \sigma}\ (\textsc{Cast})$$

$$\frac{\Phi \vdash p : {B_0}^{\beta} \qquad B_0 <: B \qquad \Gamma \vdash p : B \overset{f}{\leadsto} q : C \qquad \textbf{get}\langle B.f : C^{\gamma}\rangle({B_0}^{\beta}) \in \Sigma[A.m]}{\Pi, \Gamma; \Phi, A.m, \sigma \to_{\Sigma} \Pi, \Gamma; \Phi \cup \{q : C^{\gamma}\}, A.m, \sigma}\ (\textsc{Get})$$

$$\frac{\Phi \vdash p : {B_0}^{\beta} \qquad B_0 <: B \qquad \Phi \vdash q : C^{\gamma} \qquad \textbf{put}\langle B.f : C^{\gamma}\rangle({B_0}^{\beta}) \in \Sigma[A.m]}{\Pi, \Gamma; \Phi, A.m, \sigma \to_{\Sigma} \Pi, \Gamma \cup \{p : B \overset{f}{\leadsto} q : C\}; \Phi, A.m, \sigma}\ (\textsc{Put})$$

$$\frac{\begin{array}{c} \Phi \vdash r_0 : {C_0}^{\gamma_0} \qquad C_0 <: B \qquad \Phi \vdash \overline{r} : \overline{C}^{\overline{\gamma}} \\ \Pi \vdash r_0 : B'' \qquad B'' <: B' \qquad B' <: B \\ \textbf{invoke}\langle B.n : \overline{C}^{\overline{\gamma}/\overline{\beta}} \to C^{\beta/\gamma}\rangle[B'.n']({C_0}^{\gamma_0/\beta_0}) \in \Sigma[A.m] \end{array}}{\Pi, \Gamma; \Phi, A.m, \sigma \to_{\Sigma} \Pi, \Gamma; \Phi', B'.n', \sigma'}\ (\textsc{Invoke})$$

$$\text{where } \Phi' = \{r_0 : {B'}^{\beta_0}, \overline{r} : \overline{C}^{\overline{\beta}}\} \text{ and } \sigma' = \mathit{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma)$$

$$\frac{\Phi' \vdash r : C^{\beta}}{\Pi, \Gamma; \Phi', B'.n', \mathit{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma) \to_{\Sigma} \Pi, \Gamma; \Phi \cup \{r : C^{\gamma}\}, A.m, \sigma}\ (\textsc{Return})$$

Figure 3. A Guarded JVM Model with Policy Enforcement

designator, as in $A.m$. As reflected throughout in Fig. 3, the execution context in the run-time stack assumes the new form.

$$\sigma ::= \diamond \mid \mathit{push}(\Phi, A.m, C, \sigma)$$

(Again, if $|\mathcal{M}| = 1$ then an execution context can be identified solely by the declaring class of the method.)

Annotated type labels Recall that references accessible from a stack frame is labeled by a type label, as in $\Phi \vdash r : C$. We now allow the instrumentation process to track data dependencies via the use of *type annotations*. Specifically, type annotations $\alpha, \beta, \gamma \in \mathcal{A}$ can be used to decorate the type labels of object references stored in a stack frame, as in $\Phi \vdash r : C^{\gamma}$. Whenever a labeled reference is introduced into the top stack frame, as in the cases of (WIDEN), (NEW), (CAST) and (GET), a type annotation is attached to the type label. The type annotation is erased when the labeled reference is stored into a field via (PUT). The most complex of all the transition rules are the pair (INVOKE) and (RETURN). As in other transition rules, when arguments ($\overline{r} : \overline{C}^{\overline{\gamma}}$) are passed into the callee stack frame (Φ'), their type labels acquire a new set of type annotations ($\overline{r} : \overline{C}^{\overline{\beta}}$). These annotations may differ from those in the caller stack frame (Φ). When a method returns, the return value (r) is introduced into the caller stack frame (Φ), and as such its type label (C) receives a new type annotation (γ) that may differ from the one (β) in the callee stack frame (Φ'). The tricky point is that the type annotation of the return object reference is decided at the time when the method is invoked (just as the type label of the return reference is decided at method invocation time). In the basic model of Fig. 1, method invocation pushes into the run-time stack the return type label C. In the new model, a doubly decorated type label $C^{\beta/\gamma}$ is pushed into the stack. The meaning is that (1) the callee must return an object reference with annotated type label C^{β}, and that (2) the returned object reference will be pushed into the caller stack frame with annotated type label C^{γ}. The transition rules (INVOKE) and (RETURN) are designed to jointly produce this behavior. (As usual, if $|\mathcal{A}| = 1$ then all type annotations may be omitted.)

Policy enforcement The ultimate goal of the aforementioned apparatus is to allow us to control the execution traces that the VM generates. In the guarded VM model, the transition relation $\rightarrow_{\Sigma}$ is now parameterized by a *safety policy* Σ. A safety policy is a function with signature $\mathcal{C} \times \mathcal{M} \rightarrow 2^{\mathcal{E}}$, where $2^{\mathcal{E}}$ denotes the powerset of the set of events $\mathcal{E}$ that may be generated by the VM. Intuitively, a policy Σ specifies for each execution context $A.m$ the set $\Sigma[A.m]$ of permitted events. An event $e \in \mathcal{E}$ has the following structure.

$$\begin{aligned} e ::= {} & \mathbf{widen}(C^{\gamma})(B^{\beta}) \mid \mathbf{new}\langle B^{\beta}\rangle \mid \mathbf{cast}\langle B^{\beta}\rangle(C^{\gamma}) \mid \\ & \mathbf{get}\langle B.f : C^{\gamma}\rangle({B_0}^{\beta}) \mid \mathbf{put}\langle B.f : C^{\gamma}\rangle({B_0}^{\beta}) \mid \\ & \mathbf{invoke}\langle B.n : \overline{C}^{\overline{\gamma}/\overline{\beta}} \rightarrow C^{\beta/\gamma}\rangle[B'.n']({C_0}^{\gamma_0/\beta_0}) \end{aligned}$$

The guarded model in Fig. 3 ensures that every transition produced by $\rightarrow_{\Sigma}$ satisfies the parameter policy Σ. One may instantiate the model by a concrete policy, and then verify if some global safety property (in the form of a state invariant) is preserved by the transitions.

Following the spirit of [11], we constrain the kind of information that may be consumed by the underlying protection mechanism for the purpose of access control. Event signatures have been carefully designed so that a safety policy may only control execution by examining type instrumentation and static annotation. Two transitions involving the same type instrumentation and static annotation are indistinguishable from the point of view of the safety policy. Information such as object identity, dynamic types and link graph topology are intentionally hidden from the protection mechanism. This set up allows us to model the effect of imposing a purely type-based protection mechanism.

5. Example: Confined Types

In this section, we look at how Featherweight JVM may be used to evaluate a realistic type system with security application. Specifically, we will examine two formulations of Confined Types, and attempt to establish a Confinement Theorem for each formulation. This modeling exercise allows us to point out a number of subtle implementation issues if Confined Types is to be enforced at the bytecode level.

5.1. Confined Types from 20,000 Feet

At the core of the Java security infrastructure is a strong type system, which provides non-bypassable encapsulation boundaries for controlling access to privileged services and sensitive data. To appreciate the connection between encapsulation and security, recall that the soundness of the JVM type system guarantees no type confusion may occur, and thus the security manager is properly encapsulated, and consequently the rest of the Java protection infrastructure can function as designed. Both the Java source language and the JVM bytecode language support access control modifiers (e.g., `public`, `protected`, etc) for enforcing the usual notion of *data encapsulation*. The standard Java platform, however, offers no provision for enforcing the stronger notion of *reference encapsulation*. The lack of programmatic support for preventing accidental reference leaking has led to a security breach in the `java.security` package of JDK 1.1 [34].

$\mathcal{C}1$ A confined type must not appear in the type of a public (or protected) field or the return type of a public (or protected) method.

$\mathcal{C}2$ A confined type must not be public.

$\mathcal{C}3$ Methods invoked on an expression of confined type must either be defined in a confined class or be anonymous methods.

$\mathcal{C}4$ Subtypes of a confined type must be confined.

$\mathcal{C}5$ Confined types can be widened only to other confined types.

$\mathcal{C}6$ Overriding must preserve anonymity of methods.

$\mathcal{A}1$ [In an anonymous method,] the `this` reference is used only to select fields and as the receiver in the invocation of other anonymous methods.

Figure 4. Type Rules for Confined Types [37]

The idea of Confined Types [34, 20, 36, 37] was proposed as a lightweight annotation system for supporting reference encapsulation in a Java-like safe language. It has been shown convincingly that proper adoption of confined types in Java could have prevented the aforementioned security breach [34]. A class or interface type may be declared to be *confined*. The typing discipline ensures the following property.

Confinement Property (Informal) [36, 37] An object of confined type is encapsulated within its defining scope [i.e., package].

Confined Types enforces the typing discipline of Fig. 4 (adopted from [37], with minor editing). The idea of an anonymous method requires explanation. A confined object may be leaked outside of its package when it acts as the receiver (i.e., `this`) of a method invocation in which the dispatched method is one that has been inherited from a non-confined superclass. Completely disallowing this will render the typing discipline too restrictive. The idea of an anonymous method is therefore introduced. Essentially, an anonymous method promises the classes which inherit the method that the `this` reference will never be stored into a field. As such, method anonymity is closely related to the Boyland's borrowed receiver [6].

5.2. A Safety Policy for Confined Types

We encode the the type rules of Fig. 4 into a safety policy for Featherweight JVM (Fig. 5).

5.2.1 Notation

Reference types are either public or package private. We use the predicate *public*(C) to assert that C is public. Following [36, 37], type rule $\mathcal{C}2$ is modeled by identifying the package private classes with the confined classes. The following shorthand is therefore defined:

$$\textit{confined}(C) \triangleq \neg\textit{public}(C)$$

We write $B \approx C$ to assert that reference types B and C belong to the same package. We define the relation, $C \triangleright B$ to assert that C is visible to B.

$$C \triangleright B \triangleq \textit{public}(C) \vee B \approx C$$

We also define a transitive variant of the visibility relation:

$$C \blacktriangleright B \triangleq \textit{public}(C) \vee (\textit{confined}(B) \wedge B \approx C)$$

The following properties can be easily verified:

$$C \blacktriangleright C \tag{1}$$
$$C \blacktriangleright B \wedge B \blacktriangleright A \Rightarrow C \blacktriangleright A \tag{2}$$
$$C \blacktriangleright A \Rightarrow C \triangleright A \tag{3}$$
$$C \triangleright C \tag{4}$$
$$C \blacktriangleright B \wedge B \triangleright A \Rightarrow C \triangleright A \tag{5}$$

We postulate that the underlying protection mechanism enforces type rule $\mathcal{C}4$ when a class is defined by a class loader:

$$C' <: C \quad \Rightarrow \quad C \blacktriangleright C' \tag{6}$$

Following [36, 37], all fields and methods are assumed to be public to simplify discussion. A field (or method) is encapsulated by being declared in a confined reference type. To facilitate the enforcement of the type rules $\mathcal{C}3$ and $\mathcal{A}1$, method designators are defined to indicate if a given method is anonymous:

$$m ::= \textbf{anon} \mid \overline{\textbf{anon}}$$

No field designator needs to be defined, and thus we will omit field designators in our further discussion. Three type annotations are defined to track if a given reference is the `this` pseudo-parameter of a method:

$$\alpha, \beta, \gamma ::= \top \mid \textbf{this} \mid \overline{\textbf{this}}$$

The type annotation $\top$ indicates no information for the underlying reference. A subsumption relation $\sqsubset:$ is defined for the type annotations, so that $\sqsubset:$ is a partial ordering with $\top$ being the maximal element:

$$\textbf{this} \sqsubset: \top \qquad \overline{\textbf{this}} \sqsubset: \top$$

$$\frac{\begin{array}{c} B \triangleright A \qquad \gamma \sqsubset: \beta \\ C \blacktriangleright B \vee (\beta \neq \overline{\textbf{this}} \wedge m = \textbf{anon}) \end{array}}{\textbf{widen}(C^{\gamma})(B^{\beta}) \in \Sigma[A.m]}$$

$$\frac{B \triangleright A \qquad \beta = \overline{\textbf{this}}}{\textbf{new}\langle B^{\beta}\rangle \in \Sigma[A.m]}$$

$$\frac{\begin{array}{c} B \triangleright A \qquad \gamma \sqsubset: \beta \\ C \blacktriangleright B \vee (\beta \neq \overline{\textbf{this}} \wedge m = \textbf{anon}) \end{array}}{\textbf{cast}\langle B^{\beta}\rangle(C^{\gamma}) \in \Sigma[A.m]}$$

$$\frac{B \triangleright A \qquad C \blacktriangleright B \qquad \gamma = \overline{\textbf{this}}}{\textbf{get}\langle B : C^{\gamma}\rangle({B_0}^{\beta}) \in \Sigma[A.m]}$$

$$\frac{B \triangleright A \qquad C \blacktriangleright B \qquad \gamma = \overline{\textbf{this}} \vee m \neq \textbf{anon}}{\textbf{put}\langle B : C^{\gamma}\rangle({B_0}^{\beta}) \in \Sigma[A.m]}$$

$$\frac{\begin{array}{c} B \triangleright A \qquad n = \textbf{anon} \Rightarrow n' = \textbf{anon} \\ (C_0 \blacktriangleright B \wedge (\gamma_0 = \overline{\textbf{this}} \vee m \neq \textbf{anon})) \vee n = \textbf{anon} \\ \forall i > 0 \,.\, (C_i \blacktriangleright B \wedge (\gamma_i = \overline{\textbf{this}} \vee m \neq \textbf{anon})) \\ \beta_0 = \textbf{this} \qquad \forall i > 0 \,.\, \beta_i = \overline{\textbf{this}} \\ C \blacktriangleright B \qquad \beta = \overline{\textbf{this}} \vee n' \neq \textbf{anon} \qquad \gamma = \overline{\textbf{this}} \end{array}}{\textbf{invoke}\langle B.n : \overline{C}^{\overline{\gamma}/\overline{\beta}} \to C^{\beta/\gamma}\rangle[B'.n']({C_0}^{\gamma_0/\beta_0}) \in \Sigma[A.m]}$$

Figure 5. A Safety Policy for Confined Types

5.2.2 Safety Policy

A safety policy for Confined Types is formulated in Fig. 5. The type rule $\mathcal{C}1$ is enforced by the antecedent $C \blacktriangleright B$ of the **get**, **put** and **invoke** policy rules, ensuring that confined objects are never exposed by an unprotected field or returned by an unprotected method. The type rule $\mathcal{C}3$ is enforced in the **invoke** policy rule by the antecedent $(C_0 \blacktriangleright B \wedge \ldots) \vee n = \textbf{anon}$, which requires that a confined receiver can temporarily escape from its confinement domain only if the invoked method is anonymous. The **widen** and **cast** policy rules enforce a slightly relaxed version of $\mathcal{C}5$, so that a confined reference may be widened or casted to a non-confined one only if it is the anonymous `this`. The type rule $\mathcal{C}6$ is enforced in the **invoke** policy rule by the antecedent $n = \textbf{anon} \Rightarrow n' = \textbf{anon}$, which mandates that method dispatching preserves anonymity in the execution context. To enforce type rule $\mathcal{A}1$, the antecedent $\gamma = \overline{\textbf{this}} \vee m \neq \textbf{anon}$ is required in the **put** policy rule so that an anonymous method cannot store `this` into a field. Similarly, the antecedent $\forall i > 0 \,.\, (\ldots \wedge (\gamma_i = \overline{\textbf{this}} \vee m \neq \textbf{anon}))$ disallows the passing of an anonymous `this` as method arguments. Within the same policy rule, the antecedent $\beta = \overline{\textbf{this}} \vee n' \neq \textbf{anon}$ forbids the returning of an anonymous `this`.

A number of subtle requirements in the policy of Fig. 5 are not explicitly mandated by the type rules of Fig. 4. Nevertheless they are instrumental in the proof of the confinement theorem. The antecedent $B \triangleright A$ as found in all the policy rules is enforced by the JVM at the time of constant pool resolution [26, Sect. 5.4.3]. This property is exploited in the confinement proof. The antecedent $\forall i > 0 \,.\, (C_i \blacktriangleright B \wedge \ldots)$ in the **invoke** policy mandates that methods may only receive arguments from safe origins[3]. Furthermore, it is required that the anonymous `this` may be the receiver of a method call only if the target method is anonymous (see the antecedent $(\ldots \wedge (\gamma_0 = \overline{\textbf{this}} \vee m \neq \textbf{anon})) \vee n = \textbf{anon}$ of the **invoke** policy rule).

$$\frac{\begin{array}{c} \forall p, q, B, C \,.\, \Gamma \vdash p : B \rightsquigarrow q : C \;\Rightarrow\; C \blacktriangleright B \\ \forall p, q, B, C, C' \,.\, (\Gamma \vdash p : B \rightsquigarrow q : C \;\wedge\; \Pi \vdash q : C') \Rightarrow C' \blacktriangleright C \end{array}}{\mathit{ConfinedLinks}(\Gamma \mid \Pi)}$$

$$\frac{\begin{array}{c} \forall r, C \,.\, \Phi \vdash r : C^{\gamma} \;\Rightarrow\; C \triangleright A \\ \forall r, C, C' \,.\, (\Phi \vdash r : C^{\gamma} \wedge \Pi \vdash r : C') \\ \qquad \Rightarrow (C' \blacktriangleright C \vee (\gamma \neq \overline{\textbf{this}} \wedge m = \textbf{anon})) \end{array}}{\mathit{ConfinedFrame}(\Phi \mid \Pi, A.m)}$$

$$\frac{}{\mathit{ConfinedStack}(\diamond \mid \Pi, A.m)}$$

$$\frac{\begin{array}{c} C \triangleright A \qquad \beta = \overline{\textbf{this}} \vee n' \neq \textbf{anon} \qquad \gamma = \overline{\textbf{this}} \\ \mathit{ConfinedFrame}(\Phi \mid \Pi, A.m) \qquad \mathit{ConfinedStack}(\sigma \mid \Pi, A.m) \end{array}}{\mathit{ConfinedStack}(\mathit{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma) \mid \Pi, B'.n')}$$

$$\frac{\begin{array}{c} \mathit{ConfinedLinks}(\Gamma \mid \Pi) \\ \mathit{ConfinedFrame}(\Phi \mid \Pi, A.m) \qquad \mathit{ConfinedStack}(\sigma \mid \Pi, A.m) \end{array}}{\mathit{ConfinedState}(\Pi, \Gamma; \Phi, A.m, \sigma)}$$

Figure 6. Confinement Judgments

5.2.3 Confinement Theorem

The goal of imposing the safety policy of Fig. 5 is to ensure that the **Confinement Property (Informal)** of Sect. 5.1 is satisfied. Formally, given a VM state $\Pi, \Gamma; \Phi, A.m, \sigma$, we want the following properties to hold:

$$\begin{array}{r} \forall p, q, B, B', C, C' \,.\, (\, \Gamma \vdash p : B \rightsquigarrow q : C \;\wedge \\ \Pi \vdash q : B' \;\wedge\; \Pi \vdash q : C' \,) \Rightarrow C' \triangleright B' \end{array} \quad (7)$$

$$\begin{array}{r} \forall p, C, C', \gamma \,.\, (\, \Phi \vdash p : C^{\gamma} \;\wedge\; \Pi \vdash p : C' \;\wedge \\ (\gamma = \overline{\textbf{this}} \vee m \neq \textbf{anon}) \,) \Rightarrow C' \triangleright A \end{array} \quad (8)$$

Property (7) asserts that instances of confined classes are only stored in fields declared within the same package. Property (8) states that, except for an anonymous `this`, confined objects remains in the stack frame of methods declared in the same package. To enforce the above properties, we postulate the state invariant $\mathit{ConfinedState}(\Pi, \Gamma; \Phi, A.m, \sigma)$ as specified in Fig. 6.

[3]A weaker antecedent $\forall i > 0 \,.\, (C_i \triangleright B' \wedge \ldots)$ will also do. The stronger antecedent was adopted for notational uniformity.

Proposition 2 *If both* *SafeState*$(\Pi, \Gamma; \Phi, A, \sigma)$ *and* *ConfinedState*$(\Pi, \Gamma; \Phi, A.m, \sigma)$ *hold, then properties (7) and (8) hold.*

Proof: Property (7) follows from *ConfinedLinks*$(\Gamma \mid \Pi)$ and *SafeLinks*$(\Gamma \mid \Pi)$ via (2), (6), and (3). Property (8) follows from *ConfinedFrame*$(\Phi \mid \Pi, A.m)$ via (5).

Analogous to **One-Step Soundness** (Theorem 1), the following **Confinement Theorem** can be proven:

Theorem 3 (Confinement)

$$\begin{array}{l}
\forall\, \Pi, \Gamma, \Phi, A, m, \sigma, \Pi', \Gamma', \Phi', A', m', \sigma' \,. \\
\quad (\ \Pi, \Gamma; \Phi, A.m, \sigma \rightarrow \Pi', \Gamma'; \Phi', A'.m', \sigma' \\
\quad\quad \wedge\ \textit{SafeState}(\Pi, \Gamma; \Phi, A, \sigma) \\
\quad\quad \wedge\ \textit{ConfinedState}(\Pi, \Gamma; \Phi, A.m, \sigma)\ ) \\
\quad\quad\quad \Rightarrow \textit{ConfinedState}(\Pi', \Gamma'; \Phi', A'.m', \sigma')
\end{array}$$

See Appendix B for a proof of this theorem.

5.3. A Capability-Based Safety Policy for Confined Types

5.3.1 Motivation

The above formulation of Confined Types closely mirrors the type rules of Fig. 4. Although the safety policy in Fig. 5 successfully preserves the Confinement Property, the price of enforcing such a safety policy at link time is non-trivial. To understand this cost, a reference type C is said to be an auxiliary type for reference type A if C appears in the constant pool of A as a field type, or a method parameter or return type. To enforce the above safety policy at link time, auxiliary types for A must be loaded in order for the type checker to confirm their confined-ness. As the loading of auxiliary classes is not mandated by the JVM Specification [26], such an eager class loading strategy could slow down the start up time of an application, and increase the memory footprint of the VM unnecessarily.

In this section, we explore an alternative formulation of Confined Types based on the notion of capability types [9, 7]. Intuitively, rather than relying on class loading to confirm the confined-ness of auxiliary types, every auxiliary type is explicitly annotated by a capability that provides an estimate of its confined-ness. Although the capabilities are only estimates, the design of the typing discipline is such that a classfile that is not honest about the annotations will fail to type check.

$$\frac{B^{\beta} \triangleright\!: A \vee \beta = \textbf{anon} \qquad C^{\gamma} \blacktriangleright\!: B^{\beta}}{\textbf{widen}(C^{\gamma})(B^{\beta}) \in \Sigma[A]}$$

$$\frac{B^{*} \triangleright\!: A}{\textbf{new}\langle B^{*}\rangle \in \Sigma[A]}$$

$$\frac{B^{\beta} \triangleright\!: A \vee \beta = \textbf{anon} \qquad C^{\gamma} \blacktriangleright\!: B^{\beta}}{\textbf{cast}\langle B^{\beta}\rangle(C^{\gamma}) \in \Sigma[A]}$$

$$\frac{B^{*} \triangleright\!: A \qquad C^{\gamma} \blacktriangleright\!: B^{*}}{\textbf{get}\langle B.\gamma : C^{\gamma}\rangle({B_0}^{\beta}) \in \Sigma[A]}$$

$$\frac{B^{*} \triangleright\!: A \qquad C^{\gamma} \blacktriangleright\!: B^{*}}{\textbf{put}\langle B.\gamma : C^{\gamma}\rangle({B_0}^{\beta}) \in \Sigma[A]}$$

$$\frac{\begin{array}{c} B^{*} \triangleright\!: A \qquad {C_0}^{\gamma_0} \blacktriangleright\!: {B'}^{\beta'_0} \qquad {B'}^{*} \blacktriangleright\!: {B'}^{\beta'_0} \\ \forall i > 0 \,.\, {C_i}^{\gamma_i} \blacktriangleright\!: B^{*} \qquad C^{\gamma} \blacktriangleright\!: B^{*} \end{array}}{\textbf{invoke}\langle B : \overline{C}^{\overline{\gamma}/\overline{\gamma}} \rightarrow C^{\gamma/\gamma}\rangle[B']({C_0}^{\gamma_0/\beta'_0}) \in \Sigma[A]}$$

Figure 7. A Capability-Based Safety Policy for Confined Types

5.3.2 Notation

A partially-ordered set of capabilities are defined to track the confined-ness of references.

$$\begin{array}{c} \alpha, \beta, \gamma ::= \bot \mid \textbf{conf} \mid \textbf{anon} \\ \bot \sqsubset\!: \textbf{conf} \sqsubset\!: \textbf{anon} \end{array}$$

The capability $\bot$ is used for tagging references that are believed to be public, and the **conf** capability for confined references. References tagged with **anon** may temporarily escape from its confinement domain as a method receiver so long as it is never deposited into a field. We postulate that field types, method parameter types (including the pseudo-parameter `this`) and method return types are all annotated with the these capabilities. Consequently, an anonymous method is represented by having `this` annotated by the **anon** capability. Method designators are therefore not needed in this model[4]. We, however, use field designators

[4]A more accurate instantiation of the model would explicitly record the capability of the `this` formal parameter with a method designator.

$$m, n ::= \gamma$$

Under this set up the policy rule for **invoke** would look like the following:

$$\frac{\begin{array}{c} B^{*} \triangleright\!: A \qquad {C_0}^{\gamma_0} \blacktriangleright\!: B^{\beta_0} \qquad B^{\beta_0} \blacktriangleright\!: {B'}^{\beta'_0} \qquad {B'}^{*} \blacktriangleright\!: {B'}^{\beta'_0} \\ \forall i > 0 \,.\, {C_i}^{\gamma_i} \blacktriangleright\!: B^{*} \qquad C^{\gamma} \blacktriangleright\!: B^{*} \end{array}}{\textbf{invoke}\langle B.\beta_0 : \overline{C}^{\overline{\gamma}/\overline{\gamma}} \rightarrow C^{\gamma/\gamma}\rangle[B'.\beta'_0]({C_0}^{\gamma_0/\beta'_0}) \in \Sigma[A.\alpha]}$$

The added notational complexity only plays a minor role in the confinement proof. Specifically, the antecedents $C_0^{\gamma_0} \blacktriangleright\!: B^{\beta_0}$ and $B^{\beta_0} \blacktriangleright\!: {B'}^{\beta'_0}$ allow us to deduce $C_0^{\gamma_0} \blacktriangleright\!: {B'}^{\beta'_0}$. We omit the method designator, and to record the capabilities of fields.

$$f, g ::= \gamma$$

The default type label C^* of a reference type C is defined as follows:

$$C^* \triangleq \begin{cases} C^{\perp} & \text{if } \mathit{public}(C) \\ C^{\mathbf{conf}} & \text{otherwise} \end{cases}$$

We define a variant of the visibility relation in terms of capability type annotations:

$$C^{\gamma} \triangleright\!: B \;\triangleq\; \gamma = \perp \,\vee\, C \approx B$$

As before, a transitive variant of visibility is also defined:

$$C^{\gamma} \blacktriangleright\!: B^{\beta} \triangleq \\ \gamma \sqsubset\!: \beta \;\wedge\; (\gamma \neq \mathbf{conf} \;\vee\; \beta \neq \mathbf{conf} \;\vee\; C \approx B)$$

The following properties can be easily validated:

$$C^{\gamma} \blacktriangleright\!: C^{\gamma} \tag{9}$$

$$C^{\gamma} \blacktriangleright\!: B^{\beta} \wedge B^{\beta} \blacktriangleright\!: A^{\alpha} \Rightarrow C^{\gamma} \blacktriangleright\!: A^{\alpha} \tag{10}$$

$$C^* \blacktriangleright\!: B^* \Leftrightarrow C \blacktriangleright B \tag{11}$$

$$C^{\gamma} \blacktriangleright\!: B^* \Rightarrow C^{\gamma} \triangleright\!: B \tag{12}$$

$$C^{\gamma} \triangleright\!: C^{\gamma} \tag{13}$$

$$C^{\gamma} \blacktriangleright\!: B^{\beta} \wedge B^{\beta} \triangleright\!: A \Rightarrow C^{\gamma} \triangleright\!: A \tag{14}$$

$$C^* \triangleright\!: B \Leftrightarrow C \triangleright B \tag{15}$$

$$B^* \blacktriangleright\!: B^{\beta} \Rightarrow (B^{\beta} \triangleright\!: B \vee \beta = \mathbf{anon}) \tag{16}$$

5.3.3 Safety Policy

A capability-based safety policy for Confined Types is given in Fig. 7. This policy is significantly cleaner than the previous one[5]. Notice that accesses are granted by examining not the actual confined-ness of auxiliary types (i.e., field types, and method parameter and return types), but rather type labels annotated with capabilities. This effectively cut down the amount of class loading required to type check a method body.

5.3.4 Confinement Theorem Revisited

The goal of confinement is still to uphold property (7), plus property (8) adapted as follows:

$$\forall p, C, C', \gamma \,.\, (\, \Phi \vdash p : C^{\gamma} \;\wedge\; \Pi \vdash p : C' \;\wedge \\ \gamma \neq \mathbf{anon} \,) \Rightarrow C' \triangleright A \tag{17}$$

$$\frac{\begin{array}{c} \forall p, q, B, C \,.\, p : B \overset{\gamma}{\rightsquigarrow} q : C \;\Rightarrow\; C^{\gamma} \blacktriangleright\!: B^* \\ \forall p, q, B, C, C' \,.\, (p : B \overset{\gamma}{\rightsquigarrow} q : C \;\wedge\; \Pi \vdash q : C') \Rightarrow C'^* \blacktriangleright\!: C^{\gamma} \end{array}}{\mathit{ConfinedLinks}_{\mathit{cap}}(\Gamma \mid \Pi)}$$

$$\frac{\begin{array}{c} \forall r, C \,.\, \Phi \vdash r : C^{\gamma} \;\Rightarrow\; (C^{\gamma} \triangleright\!: A \vee \gamma = \mathbf{anon}) \\ \forall r, C, C' \,.\, (\Phi \vdash r : C^{\gamma} \wedge \Pi \vdash r : C') \Rightarrow C'^* \blacktriangleright\!: C^{\gamma} \end{array}}{\mathit{ConfinedFrame}_{\mathit{cap}}(\Phi \mid \Pi, A)}$$

$$\frac{}{\mathit{ConfinedStack}_{\mathit{cap}}(\diamond \mid \Pi)}$$

$$\frac{\begin{array}{c} C^{\gamma} \triangleright\!: A \\ \mathit{ConfinedFrame}_{\mathit{cap}}(\Phi \mid \Pi, A) \qquad \mathit{ConfinedStack}_{\mathit{cap}}(\sigma \mid \Pi) \end{array}}{\mathit{ConfinedStack}_{\mathit{cap}}(\mathit{push}(\Phi, A, C^{\gamma/\gamma}, \sigma) \mid \Pi)}$$

$$\frac{\begin{array}{c} \mathit{ConfinedLinks}_{\mathit{cap}}(\Gamma \mid \Pi) \\ \mathit{ConfinedFrame}_{\mathit{cap}}(\Phi \mid \Pi, A) \qquad \mathit{ConfinedStack}_{\mathit{cap}}(\sigma \mid \Pi) \end{array}}{\mathit{ConfinedState}_{\mathit{cap}}(\Pi, \Gamma; \Phi, A, \sigma)}$$

Figure 8. Revised Confinement Judgments

To accommodate the variation in notation and confinement goals, the confinement invariant has been reformulated, as shown in Fig. 8. Again, we assert that the invariant establishes the confinement goals.

Proposition 4 *If both* $\mathit{SafeState}(\Pi, \Gamma; \Phi, A, \sigma)$ *and* $\mathit{ConfinedState}_{\mathit{cap}}(\Pi, \Gamma; \Phi, A, \sigma)$ *hold, then properties (7) and (17) hold.*

Proof: Property (7) follows from $\mathit{ConfinedLinks}_{\mathit{cap}}(\Gamma \mid \Pi)$ and $\mathit{SafeLinks}(\Gamma \mid \Pi)$ via (10), (11), (6), (2), and (3). Property (8) follows from $\mathit{ConfinedFrame}_{\mathit{cap}}(\Phi \mid \Pi, A)$ via (14) and (15).

A revised confinement theorem can be proven:

Theorem 5 (Confinement (Revised))

$$\begin{array}{l} \forall\, \Pi, \Gamma, \Phi, A, \sigma, \Pi', \Gamma', \Phi', A', \sigma' \,. \\ \qquad (\;\; \Pi, \Gamma; \Phi, A, \sigma \to \Pi', \Gamma'; \Phi', A', \sigma' \\ \qquad\quad \wedge\;\; \mathit{SafeState}(\Pi, \Gamma; \Phi, A, \sigma) \\ \qquad\quad \wedge\;\; \mathit{ConfinedState}_{\mathit{cap}}(\Pi, \Gamma; \Phi, A, \sigma) \;\;) \\ \qquad\qquad \Rightarrow \mathit{ConfinedState}_{\mathit{cap}}(\Pi', \Gamma'; \Phi', A', \sigma') \end{array}$$

See Appendix C for the proof of this theorem.

5.3.5 Benefits of Reformulation

We started by promising that the capability-based reformulation reduces the amount of class loading performed by the link-time type checking algorithm. With the original formulation, every time the checks $C \blacktriangleright B$ and $C \triangleright B$ are carried out, class loading must be performed in order for the type checker to figure out the confined-ness of the classes C and B. With the capability-based reformulation, class loading

require $C_0^{\gamma_0} \blacktriangleright\!: B'^{\beta'_0}$ directly as an antecedent so that we do not need to carry a method designator in every execution context.

[5]An alternative formulation of the **invoke** policy rule would have the antecedent $\forall i > 0 \,.\, C_i^{\gamma_i} \blacktriangleright\!: B^*$ replaced by the weaker condition $\forall i > 0 \,.\, C_i^{\gamma_i} \triangleright\!: B'$. The stronger antecedent was adopted for notational uniformity.

is not always necessary. For example, checking $C^\gamma \blacktriangleright\!: B^\beta$ sometimes only involves the comparison of capability labels. It is only when (i) both γ and $beta$ are **conf**, (ii) the fully qualified names of B and C are different, and (iii) B and C share the same name qualification, that we need to load the definitions of B and C to verify if they belong to the same run-time package[6]. Class loading can be avoided in other cases.

6. Conclusion and Future Work

In this paper we proposed a lightweight formal model of JVM-like environments for evaluating the effectiveness of type-based protection mechanisms at an early stage of design. We presented a rational reconstruction of Confined Types at the JVM bytecode level, and verified that our formulation successfully enforces the **Confinement Property**. We then articulate a capability-based reformulation of Confined Types that can reduce the amount of class loading at type checking time. The **Confinement Property** is preserved by this alternative formulation. We have therefore provided first evidence on the utility of FJVM as a tool for lightweight evaluation of type-based protection mechanisms.

We plan to explore a number of future directions. Firstly, we are particularly interested in the application of type-based access control mechanisms to enforce security properties [2, 35] at the bytecode level. We plan to use Featherweight JVM as a model to validate early designs. Secondly, we plan to expand our model to cover Java features such as exception handling and arrays, and to explore the encoding of source-level generic types [8, 23] in FJVM type annotations. Thirdly, we want to devise a semantic linkage between FJVM and a bytecode-level type system in the style of [33, 17, 18], so that results established in FJVM may be transferred to the bytecode-level type system.

References

[1] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, Cape Breton, Nova Scotia, Canada, June 2002.

[2] A. Banerjee and D. A. Naumann. Using access control for secure information flow in a Java-like language. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW'03)*, Pacific Grove, CA, USA, June 2003.

[3] A. Banerjee and D. A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 15(2):131–177, Mar. 2005.

[4] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Proceedings of the Workshop on Foundations of Computer Security (FCS'02)*, Copenhagen, Denmark, July 2002.

[5] G. M. Bierman, M. J. Parkinson, and A. M. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report UCAM-CL-TR-563, University of Cambridge Computer Laboratory, Apr. 2003.

[6] J. Boyland. Alias burying: Unique variables without destructive reads. *Software — Practice and Experience*, 31(6):533–553, 2001.

[7] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalization of uniqueness and read-only. In *Proceedings of the 2001 European Conference on Object-Oriented Programming*, pages 2–27, Budapest, Hungary, July 2001.

[8] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 183–200, Vancouver, BC, Canada, Oct. 1998.

[9] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, pages 262–275, San Antonio, Texas, USA, Jan. 1999.

[10] ECMA. C# language specification (3rd edition). Standard ECMA-334, ECMA, June 2005.

[11] P. W. L. Fong. Access control by tracking shallow execution history. In *Proceeding of the 2004 IEEE Symposium on Security and Privacy (S&P'04)*, pages 43–55, Berkeley, California, May 2004.

[12] P. W. L. Fong. Pluggable verification modules: An extensible protection mechanism for the JVM. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 404–418, Vancouver, BC, Canada, Oct. 2004.

[13] P. W. L. Fong. Link-time enforcement of confined types for JVM bytecode. In *Proceedings of the Third Annual Conference on Privacy, Security and Trust (PST'05)*, St. Andrews, New Brunswick, Canada, Oct. 2005.

[14] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proceedings of the ACM 1999 Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, Georgia, USA, May 1999.

[15] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM 2002 Conference on Programming Language Design and Implementation*, pages 1–12, Berlin Germany, June 2002.

[16] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, May 2003.

[17] S. N. Freund and J. C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, Nov. 1999.

[18] S. N. Freund and J. C. Mitchell. A type system for the Java bytecode language and verifier. *Journal of Automated Reasoning*, 30(3–4):271–321, Sept. 2003.

[6]The *run-time package* of a class or an interface is uniquely determined by the package name and the defining class loader of the class or interface [26, Sect. 5.3].

[19] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 3rd edition, 2005.

[20] C. Grothoff, J. Palsberg, and J. Vitek. Encapsulating objects with confined types. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 241–255, Tampa Bay, FL, USA, Oct. 2001.

[21] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, to appear. Also available as Cornell Computer Science Department Technical Report TR 2003-1908, August 2003.

[22] P. H. Hartel and L. Moreau. Formalizing the safety of Java, the Java Virtual Machine, and Java Card. *ACM Computing Surveys*, 33(4):517–558, Dec. 2001.

[23] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[24] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, Feb. 2005.

[25] J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS'05)*, Milan, Italy, Sept. 2005.

[26] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999.

[27] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[28] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, Jan. 1997.

[29] T. Nipkow. Java bytecode verification. *Journal of Automated Reasoning*, 30(3–4), Sept. 2003.

[30] F. Pottier, C. Skalka, and S. Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, 27(2):344–382, Mar. 2005.

[31] Z. Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems*, 22(4):638–672, July 2000.

[32] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, Feb. 2000.

[33] R. Stata and M. Abadi. A type system for Java bytecode subroutine. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, Jan. 1999.

[34] J. Vitek and B. Bokowski. Confined types in Java. *Software — Practice and Experience*, 31(6):507–532, 2001.

[35] T. Zhao and J. Boyland. Type annotations to improve stack-based access control. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 197–210, Aix-en-Provence, France, June 2005.

[36] T. Zhao, J. Palsberg, and J. Vitek. Lightweight confinement for Featherweight Java. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 135–148, Anaheim, CA, USA, Oct. 2003.

[37] T. Zhao, J. Palsberg, and J. Vitek. Type-based confinement. *Journal of Functional Programming*, 16(1):83–128, Jan. 2006.

A. Proof of Theorem 1

A sketch of the proof for Theorem 1 is given in this appendix. We check *SafeState*$(\Pi', \Gamma; \Phi', A', \sigma')$ by a case analysis on the transition rule (Fig. 1) employed to obtain $\Pi, \Gamma; \Phi, A, \sigma \to \Pi', \Gamma'; \Phi', A', \sigma'$.

Case (WIDEN):
To show *SafeFrame*$(\Phi \cup \{r : B\} \,|\, \Pi)$, notice that *SafeFrame*$(\Phi \,|\, \Pi)$ ensures $C' <: C$ if C' is the class of r. As (WIDEN) guarantees $C <: B$, we obtain, by transitivity, $C' <: B$ as required.

Case (NEW):
Because r is a fresh object reference from $\mathcal{O}$, *SafeHeap*$(\Pi \cup \{r : B\})$ holds trivially. As well, *SafeFrame*$(\Phi \cup \{r : B\} \,|\, \Pi \cup \{r : B\})$ holds because $B <: B$.

Case (CAST):
To see *SafeFrame*$(\Phi \cup \{r : B\} \,|\, \Pi)$, observe that (CAST) explicitly requires the antecedents $\Pi \vdash r : C'$ and $C' <: B$.

Case (GET):
To see *SafeFrame*$(\Phi \cup \{q : C\} \,|\, \Pi)$, notice that (GET) guarantees $\Gamma \vdash p : B \leadsto q : C$, which, by *SafeLinks*$(\Gamma \,|\, \Pi)$, implies that $C' <: C$ when $\Pi \vdash q : C'$.

Case (PUT):
We show *SafeLinks*$(\Gamma \cup \{p : B \leadsto q : C\} \,|\, \Pi)$ in 2 steps. First, by *SafeFrame*$(\Phi \,|\, \Pi)$, $\Pi \vdash q : C'$ implies $C' <: C$ as required. Second, by *SafeFrame*$(\Phi \,|\, \Pi)$ again, $\Pi \vdash p : B'$ implies $B' <: B_0$. As (PUT) guarantees $B_0 < B$, the latter in turn implies $B' <: B$ as required.

Case (INVOKE):
We need to show *SafeFrame*$(\Phi' \,|\, \Pi)$ and *SafeStack*$(\sigma' \,|\, \Pi)$.

To show *SafeFrame*$(\Phi' \,|\, \Pi)$, where $\Phi' = \{r_0 : B', \overline{r} : \overline{C}\}$, notice the following two points. Firstly, (INVOKE) guarantees $\Phi \vdash \overline{r} : \overline{C}$, which, by *SafeFrame*$(\Phi \,|\, \Pi)$ implies that $C'_i <: C_i$ follows from $\Pi \vdash r_i : C'_i$. Secondly, (INVOKE) guarantees that $\Pi \vdash r_0 : B''$ and $B'' <: B'$ hold simultaneously.

To show *SafeStack*$(\sigma' \mid \Pi)$, where $\sigma' = \mathit{push}(\Phi, A, C, \sigma)$, notice that *SafeFrame*$(\Phi \mid \Pi)$ and *SafeStack*$(\sigma \mid \Pi)$ are both given by the precondition.

Case (RETURN):

Firstly, *SafeStack*$(\sigma \mid \Pi)$ is guaranteed by *SafeStack*$(\mathit{push}(\Phi, A, C, \sigma) \mid \Pi)$. Secondly, we show *SafeFrame*$(\Phi \cup \{r : C\} \mid \Pi)$. Because *SafeFrame*$(\Phi \mid \Pi)$ is guaranteed by *SafeStack*$(\mathit{push}(\Phi, A, C, \sigma) \mid \Pi)$, it suffices to show that $\Pi \vdash r : C'$ implies $C' <: C$. The implication is ensured by *SafeFrame*$(\Phi' \mid \Pi)$.

B. Proof of Theorem 3

A sketch of the proof for Theorem 3 is given in this appendix. We check *ConfinedState*$(\Pi', \Gamma'; \Phi', A'.m', \sigma')$ by a case analysis on the transition rule (Fig. 3) employed to obtain $\Pi, \Gamma; \Phi, A.m, \sigma \to \Pi', \Gamma'; \Phi', A'.m', \sigma'$. (All references to policy rules refer to Fig. 5.)

Case (WIDEN):

We show *ConfinedFrame*$(\Phi \cup \{r : B^{\beta}\} \mid \Pi, A.m)$ in 2 steps. First, the **widen** policy rule guarantees that $B \triangleright A$. Second, by *ConfinedFrame*$(\Phi \mid \Pi, A.m)$, $\Phi \vdash r : C^{\gamma}$ and $\Pi \vdash r : C'$ jointly imply that either (i) $C' \blacktriangleright C$ or (ii) $(\gamma \neq \overline{\textbf{this}} \wedge m = \textbf{anon})$. If case (i) holds, then the **widen** policy rule ensures that $C \blacktriangleright B \vee (\beta \neq \overline{\textbf{this}} \wedge m = \textbf{anon})$, and thus by (2) we deduce $(C' \blacktriangleright B \vee (\gamma \neq \overline{\textbf{this}} \wedge m = \textbf{anon}))$ as required. Otherwise, case (ii) holds. Because the **widen** policy rule ensures that $\gamma \sqsubset: \beta$, we deduce $(\beta \neq \overline{\textbf{this}} \wedge m = \textbf{anon})$ as required.

Case (NEW):

ConfinedFrame$(\Phi \cup \{r : B\} \mid \Pi \cup \{r : B^{\beta}\}, A.m)$ holds[7], because (i) the **new** policy rule mandates that $B \triangleright A$, and (ii) $B \blacktriangleright B$ by (1).

Case (CAST):

Similar to (WIDEN).

Case (GET):

We show *ConfinedFrame*$(\Phi \cup \{q : C^{\gamma}\} \mid \Pi, A.m)$ in 2 steps. First, the **get** policy rule ensures[8] that $C \blacktriangleright B$ and $B \triangleright A$. Thus, by (5), we have $C \triangleright A$ as required. Second, *ConfinedLinks*$(\Gamma \mid \Pi)$ guarantees that $\Gamma \vdash p : B \rightsquigarrow q : C$ and $\Pi \vdash q : C'$ jointly imply $C' \blacktriangleright C$ as required.

[7]The antecedent $\beta = \overline{\textbf{this}}$ in the **new** policy rule is not needed for establishing the Confinement Theorem. It is introduced to make the type analysis more accurate.

[8]The antecedent $C \blacktriangleright B$ is redundant in the **get** policy rule, because the condition is already implied by *ConfinedLinks*$(\Gamma \mid \Pi)$. It is introduced for symmetry.

Case (PUT):

We show *ConfinedLinks*$(\Gamma \cup \{p : B \rightsquigarrow q : C\} \mid \Pi)$ in 2 steps. First, $C \blacktriangleright B$ is guaranteed by the **put** policy rule. Second, because the **put** policy rule requires that $(\gamma = \overline{\textbf{this}} \vee m \neq \textbf{anon})$, *ConfinedFrame*$(\Phi \mid \Pi, A.m)$ therefore implies $C' \blacktriangleright C$ as required[9].

Case (INVOKE):

We need to show *ConfinedFrame*$(\Phi' \mid \Pi, B'.n')$ and *ConfinedStack*$(\sigma' \mid \Pi, B'.n')$.

To show *ConfinedFrame*$(\Phi' \mid \Pi, B'.n')$, where $\Phi' = \{r_0 : B'^{\beta_0}, \overline{r} : \overline{C}^{\overline{\beta}}\}$, we treat the labeled references $r_0 : B'^{\beta_0}$ and $\overline{r} : \overline{C}^{\overline{\beta}}$ separately. We first consider $\overline{r} : \overline{C}^{\overline{\beta}}$ First, because the **invoke** policy rule mandates $\beta_i = \overline{\textbf{this}}$, we need $C'_i \blacktriangleright C_i$ whenever $\Pi \vdash r_i : C'_i$. This holds because *ConfinedFrame*$(\Phi \mid \Pi, A.m)$ implies $(C'_i \blacktriangleright C_i \vee (\gamma_i \neq \overline{\textbf{this}} \wedge m = \textbf{anon}))$ while the **invoke** policy rule requires $(\gamma_i = \overline{\textbf{this}} \vee m \neq \textbf{anon})$. Second, we need $C_i \triangleright B'$. Observe that the **invoke** policy rule guarantees $C_i \blacktriangleright B$, while $B \blacktriangleright B'$ because of (6). By (2) and (3), we deduce $C_i \triangleright B'$.

We now consider the labeled reference $r_0 : B'^{\beta_0}$. By (4), we have $B' \triangleright B'$, and so it remains to show that $\Pi \vdash r_0 : C'_0$ entails $(C'_0 \blacktriangleright B' \vee (\beta_0 \neq \overline{\textbf{this}} \wedge n' = \textbf{anon}))$. There are 2 subcases: $n = \textbf{anon}$ or $n \neq \textbf{anon}$. If $n = \textbf{anon}$, then the **invoke** policy rule guarantees $n' = \textbf{anon}$ and $\beta_0 = \textbf{this}$ as required. If $n \neq \textbf{anon}$, then the **invoke** policy rule ensures that $(C_0 \blacktriangleright B \wedge (\gamma_0 = \overline{\textbf{this}} \vee m \neq \textbf{anon}))$. By *ConfinedFrame*$(\Phi \mid \Pi, A.m)$, the second conjunct implies $C'_0 \blacktriangleright C_0$. By (6), we also have $B \blacktriangleright B'$. We therefore obtain $C'_0 \blacktriangleright B'$ from (2).

To show *ConfinedStack*$(\sigma' \mid \Pi, B'.n')$, where $\sigma' = \mathit{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma)$, notice that *ConfinedFrame*$(\Phi \mid \Pi, A.m)$ and *ConfinedStack*$(\sigma \mid \Pi, A.m)$ are already guaranteed by the preconditions. As well, both $(\beta = \overline{\textbf{this}} \vee n' \neq \textbf{anon})$ and $\gamma = \overline{\textbf{this}}$ are antecedents of the **invoke** policy rule. What remains to be shown is $C \triangleright A$, which, by (5), follows from $C \blacktriangleright B$ and $B \triangleright A$, both being antecedent of the **invoke** policy rule.

Case (RETURN):

We show *ConfinedFrame*$(\Phi \cup \{r : C^{\gamma}\} \mid \Pi, A.m)$ in 2 steps. First, *ConfinedStack*$(\mathit{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma) \mid \Pi, B'.n')$ guarantees $C \triangleright A$ as required. Second, because *ConfinedStack*$(\mathit{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma) \mid \Pi, B'.n')$ requires $\gamma = \overline{\textbf{this}}$, we need to show that $\Pi \vdash r : C'$ implies $C' \blacktriangleright C$. But then

[9]The antecedent $B \triangleright A$ in the **put** policy rule is not needed for establishing the Confinement Theorem. It is introduced for symmetry.

ConfinedStack$(\textit{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma) \mid \Pi, B'.n')$ also guarantees $(\beta = \overline{\textbf{this}} \vee n' \neq \textbf{anon})$, which, by *ConfinedFrame*$(\Phi' \mid \Pi, B'.n')$, entails $C' \blacktriangleright C$ as required.

C. Proof of Theorem 5

A sketch of the proof for Theorem 5 is given in this appendix. We check *ConfinedState*$_{cap}(\Pi', \Gamma'; \Phi', A', \sigma')$ by a case analysis on the transition rule (Fig. 3) employed to obtain $\Pi, \Gamma; \Phi, A, \sigma \rightarrow \Pi', \Gamma'; \Phi', A', \sigma'$. (All references to policy rules refer to Fig. 7.)

Case (WIDEN):
We show *ConfinedFrame*$_{cap}(\Phi \cup \{r : B^{\beta}\} \mid \Pi, A)$ in 2 steps. First, the **widen** policy rule guarantees that $B^{\beta} \triangleright: A \vee \beta = \textbf{anon}$. Second, by *ConfinedFrame*$_{cap}(\Phi \mid \Pi, A)$, $\Pi \vdash r : C'$ implies $C'^{*} \blacktriangleright: C^{\gamma}$, which in turn implies $C'^{*} \blacktriangleright: B^{\beta}$ via (10) because the **widen** policy rule guarantees $C^{\gamma} \blacktriangleright: B^{\beta}$.

Case (NEW):
ConfinedFrame$_{cap}(\Pi \cup \{r : B\} \mid \Phi \cup \{r : B^{*}\}, A)$ holds, because (i) the **new** policy rule mandates that $B^{*} \triangleright: A$, and (ii) $B^{*} \blacktriangleright: B^{*}$ by (13).

Case (CAST):
Similar to (WIDEN).

Case (GET):
We show *ConfinedFrame*$_{cap}(\Phi \cup \{q : C^{\gamma}\} \mid \Pi, A)$ in 2 steps. First, the **get** policy rule ensures that $C^{\gamma} \blacktriangleright: B^{*}$ and $B^{*} \triangleright: A$. Thus, by (14), we have $C^{\gamma} \triangleright: A$ as required. Second, *ConfinedLinks*$_{cap}(\Gamma \mid \Pi)$ guarantees that $\Gamma \vdash p : B \overset{\gamma}{\rightsquigarrow} q : C$ and $\Pi \vdash q : C'$ jointly imply $C'^{*} \blacktriangleright: C^{\gamma}$ as required.

Case (PUT):
We show *ConfinedLinks*$_{cap}(\Gamma \cup \{p : B \overset{\gamma}{\rightsquigarrow} q : C\} \mid \Pi)$ in 2 steps. First, $C^{\gamma} \blacktriangleright: B^{*}$ is guaranteed by the **put** policy rule. Second, by *ConfinedFrame*$_{cap}(\Phi \mid \Pi, A)$, $\Pi \vdash r : C'$ implies $C'^{*} \blacktriangleright: C^{\gamma}$ as required.

Case (INVOKE):
We need to show *ConfinedFrame*$_{cap}(\Phi' \mid \Pi, B')$ and *ConfinedStack*$_{cap}(\sigma' \mid \Pi)$.

To show *ConfinedFrame*$_{cap}(\Phi' \mid \Pi, B')$, where $\Phi' = \{r_0 : B'^{\beta'_0}, \overline{r} : \overline{C}^{\overline{\beta}}\}$, we treat the labeled references $r_0 : B'^{\beta'_0}$ and $\overline{r} : \overline{C}^{\overline{\beta}}$ separately. We first consider $\overline{r} : \overline{C}^{\overline{\beta}}$. First, observe that *ConfinedFrame*$_{cap}(\Phi \mid \Pi, A)$ ensures that $C_i'^{*} \blacktriangleright: C_i^{\gamma_i}$ when $\Pi \vdash r_i : C_i'$. Second, the **invoke** policy rule guarantees $C_i^{\gamma_i} \blacktriangleright: B^{*}$, which, by (6), (10) and (12), implies ${C_i}^{\gamma_i} \triangleright: B'$ as required.

We now consider the labeled reference $r_0 : B'^{\beta'_0}$. First, the **invoke** policy rule requires $C_0^{\gamma_0} \blacktriangleright: B'^{\beta'_0}$, which, by *ConfinedFrame*$_{cap}(\Phi \mid \Pi, A)$ and (10), implies $C_0'^{*} \blacktriangleright: B'^{\beta'_0}$ when $\Pi \vdash r_0 : C_0'$. Second, the **invoke** policy rule guarantees $B'^{*} \blacktriangleright: B'^{\beta'_0}$, which, by (16), ensures $(B'^{\beta'_0} \triangleright: B'^{*} \vee \beta_0 = \textbf{anon})$ as required.

To show *ConfinedStack*$_{cap}(\sigma' \mid \Pi)$, where $\sigma' = \textit{push}(\Phi, A, C^{\gamma/\gamma}, \sigma)$, notice that *ConfinedFrame*$_{cap}(\Phi \mid \Pi, A)$ and *ConfinedStack*$_{cap}(\sigma \mid \Pi)$ are already guaranteed by the preconditions. What remains to be shown is $C^{\gamma} \triangleright: A$, which, by (14), follows from $C^{\gamma} \blacktriangleright: B^{*}$ and $B^{*} \triangleright: A$, both guaranteed by the **invoke** policy rule.

Case (RETURN):
We show *ConfinedFrame*$_{cap}(\Phi \cup \{r : C^{\gamma}\} \mid \Pi, A)$ in 2 steps. First, *ConfinedFrame*$_{cap}(\textit{push}(\Phi, A, C^{\gamma/\gamma}, \sigma) \mid \Pi, B')$ guarantees $C^{\gamma} \triangleright: A$ as required. Second, by *ConfinedFrame*$_{cap}(\Phi' \mid \Pi, B')$, $\Pi \vdash r : C'$ implies $C'^{*} \blacktriangleright: C^{\gamma}$ as required.