

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

`pwl.fong@cs.uregina.ca`

April 14, 2007

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, and, specifically, their ability to enforce security-motivated stack invariants and confinement properties. Rather than modeling the execution of a specific bytecode stream, Featherweight JVM is a nondeterministic event model that captures all the possible access event sequences that may be generated by a JVM-like environment when well-typed bytecode programs are executed. The effect of deploying a type-based protection mechanism can be modeled by a safety policy that constrains 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 proved in the

policy-guarded VM model.

To demonstrate the utility of the proposed approach, Vitek et al’s Confined Types has been formulated as a safety policy for the Featherweight JVM, and a corresponding confinement theorem has been established. To reduce class loading overhead, a capability-based reformulation of Confined Types is then studied, and is shown to preserve the confinement theorem. This paper thus provides first evidence on the utility of Featherweight JVM in providing early feedback to the designer of type-based protection mechanisms for JVM-like environments.

Keywords: Java Virtual Machine, type-based protection mechanism, formal verification, safety, access control, stack invariant, confinement.

1 Introduction

Static type systems have been proposed in recent years for the Java programming language [19] or its derivatives [11] in order to enforce access control and confinement properties [44, 20, 47, 48, 1, 2, 3, 16, 23, 9, 38, 46, 41, 15]. 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) [24], 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. This is the standard security posture of language-based security works such as Proof-Carrying Code [35] and Typed Assembly Language [34]: the object code to be executed by the code consumer may be produced by a malicious code producer, generated by an untrusted compiler, or tampered with during transport. In order for the code consumer to enforce a given typing discipline, the latter must be preserved at the level of the Java Virtual Machine (JVM) bytecode language [29]. 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 [39]. 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 [17]. 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* (FJVM), is proposed for evaluating bytecode-level, type-based protection mechanisms in an early design stage. In particular, the goal is to determine if a type-based protection mechanism enforces some given safety properties. The intended application is *type-based access control*. To this end, this work considers two families of safety properties that have proven in the past to be particularly relevant to access control.

1. *Reachability and Confinement Properties*: These include topological properties such as “may reach” (reachability) or “must not reach” (confinement) over the underlying object graph of the heap¹ (e.g., [48, 15]).
2. *Stack Invariants*: These include properties satisfied by individual activation records of the run-time stack, as well as the relations between activation records within the same call chain (e.g., [16, 38]).

Designed to support reasoning about the two families of safety properties stated above, FJVM is a nondeterministic transition system that tracks access events generated by a JVM-like environment, and simulates their effects on the global VM state. Only the aspects of a JVM state that are relevant to the study of reachability, confinement and stack invariants are captured in FJVM.

¹A reachability property, or may-reach property, asserts that an object reference may be reachable from some given source (e.g., a field, a class, a stack frame, etc). A confinement property, or must-not-reach property, is the negation of reachability.

Other irrelevant details of the JVM are abstracted away aggressively to reduce the complexity of the model.

- FJVM **abstracts away** the complexity of control flow (e.g., unstructured branching, subroutines, etc) by way of nondeterminism. In particular, FJVM does not model the execution of a specific bytecode stream, but instead models all the possible access event sequences that may be generated by the VM when well-typed bytecode programs are executed. Nevertheless, FJVM **explicitly models** the evolution of the run-time stack so that stack invariants may be studied.
- FJVM **abstracts away** the internal structure of a stack frame, including that of the local variable array and the operand stack. The only information preserved is the *set* of references accessible from a stack frame (i.e., reachability). Nevertheless, FJVM **explicitly models** data dependencies via type instrumentation. Along with static annotation, such instrumentation is then made available to the protection mechanism as the basis of access control decisions.
- FJVM **abstracts away** destructive updates, but **explicitly models** reachability (i.e., may reach) and confinement (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 contexts and link contexts. For example, we are not so interested in identifying *which* field of an object contains a reference to another object, so long as we know *some* field declared in a certain class does. Similarly, we are not so interested in identifying *which* method generates an activation record, so long as we know *some* method declared in a certain class does. Execution and link contexts also take into account static annotations: e.g., an object is reachable from some *public* field of a class; the stack frame of a method belonging a certain *principal* is currently in the run-time stack.

A novel feature of FJVM is that type-based protection mechanisms are modeled as *execution*

*monitors*² [40, 27, 12, 28, 21] with restricted information access [12]. Specifically, FJVM can be customized by a domain-specific *safety policy*, which constrains the access event sequences produced by the VM. Adopting the information restriction approach of [12], such a policy can be seen as an execution monitor that controls access only 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 stack and confinement invariants can then be proved 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 security invariants it preserves, and does so without overwhelming the designer with the full complexity of the JVM bytecode language. Alternatively speaking, the question answered by Featherweight JVM is not “Does the type system give rise to a certain subset of access event sequences?”, but rather “Suppose my type-based protection mechanism indeed restricts the access event sequences of the VM in a certain way, what security theorems can I establish?” FJVM allows the second question to be answered conveniently in an early stage of design. See Sect. 7 for future work on addressing the first question for FJVM.

To demonstrate the utility of the proposed approach, a safety policy for Vitek et al’s Confined Types [44, 20, 47, 48] 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. In a separate work [15], FJVM has also been employed to establish the confinement properties of a capability type system. These results provide first evidence on the utility of Featherweight JVM as an early modeling tool for evaluating type-based protection mechanisms in

²An execution monitor is a mechanism by which the dynamic state of a computational process is monitored to determine if execution should be allowed to continue.

a JVM-like environment.

The next section discusses related work. Sect. 3 offers an exposition of a basic version of FJVM, together with a soundness theorem for the type instrumentation process. Sect. 4 builds on the basic model to obtain a policy-guarded version of FJVM, which supports the incorporation of type-based protection mechanisms. To demonstrate the utility of the policy-guarded FJVM, it is employed in Sect. 5 to model two variants of Confined Types as well as their corresponding confinement theorems. Sect. 6 describes ways in which the FJVM models may be extended to incorporate static members, arrays and exception handling. Sect. 7 concludes the paper.

2 Related Work

This work is related to various efforts in the formal modeling of the JVM and its bytecode verification process [22, 37]. Moore et al developed a series of executable JVM models on ACL2 [31, 33, 32, 30]. Nipkow et al produced machine-checkable soundness proofs of the JVM bytecode verifier [26, 36]. Börger et al employed Abstract State Machines to capture the operational semantics of the JVM [42]. The goal of this work, however, differs from these previous efforts. While previous work aims at providing increasingly accurate and comprehensive models of the JVM, the present work strives to distill the aspects of the JVM that are specifically relevant to the study of confinement properties and stack invariants. A limitation is that, unlike the work mentioned above, this work does not support machine checkable proofs. This extension belongs to future work (Sect. 7).

This work is similar in its goal to calculi such as FJ [24], NanoJava [45], and MJ [6], 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 on the scale 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 (see Sect. 6, though, on how other language features can be added back to the model with ease). 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 [47, 48]. In FJVM they are expressed as stack invariants.

An explicit assumption of this work is that the JVM is equipped with some form of type-based protection mechanism in the style of [43, 17, 18], which assigns to every program point in an execution trace a *type state*. The instrumentation component of FJVM reflects this association of VM states to type states. VM-level type systems have been proposed for modeling stack inspection [23] and information flow control [4]. These type systems manage the complexity of the bytecode language by modeling only a representative subset of instructions. The FJVM approach manages complexity in a radically different manner. By way of nondeterminism, FJVM models the VM state from the perspective of an observer that is not given information about the instruction stream and the program counter. FJVM is only aware of access events that mutate the topology of the heap, alter the run-time stack, or update the instrumentation of stack frames. In other words, FJVM is fully compatible with the assumption that a rich set of instructions (including, for example, subroutines, unstructured branching, intra-procedural data transfer, etc) is responsible for the generated execution traces. See Sect. 7 for further discussion of this point.

Featherweight JVM can be seen as an instance of Security Automata [40]. The latter and its variants [5, 27, 28, 12] have been employed to characterize security policies. Fong [12] character-

izes 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. This is achieved by restricting safety policies to only consume information made available by a specific style of static annotation and type instrumentation (i.e., roughly corresponding to typing disciplines enforceable by iterative dataflow analysis, such as those found in [43, 17, 18]). The resulting model does not account for all possible type systems, as that has never been our goal. By contrast, [21] attempts to characterize the classes of all safety policies enforceable by various enforcement mechanisms, including static analysis as a special case.

Confined Types [44, 20, 47, 48] 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 leakages. Our first safety policy (Fig. 7) closely mirrors the original formulation of Confined Types [47, 48]. The second, capability-based formulation (Fig. 9) is, to our best knowledge, original. The latter formulation renders type checking modular. A prototype of such a type checker has been implemented [14] in the framework of Pluggable Verification Modules [13]. 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 Figs. 1 and 2. *Basic FJVM* is a nondeterministic production system that describes how the VM state evolves over time in reaction to access events. Nondeterminism is employed because we are not modeling the execution of one particular bytecode stream, but rather all possible access events that may be generated by the VM when well-typed bytecode

declared types	$A, B, C \in \mathcal{C}$
object references	$p, q, r \in \mathcal{O}$
VM states	$S, T ::= \langle \Pi, \Gamma; \Phi, A, \sigma \rangle$
object pools	$\Pi ::= \emptyset \mid \Pi \cup \{r : C\}$
link graphs	$\Gamma ::= \emptyset \mid \Gamma \cup \{p : B \rightsquigarrow q : C\}$
stack frames	$\Phi ::= \emptyset \mid \Phi \cup \{r : C\}$
proper stacks	$\sigma ::= \diamond \mid \text{push}(\Phi, A, C, \sigma)$

Figure 1: States for Basic FJVM

sequences are executed.

Declared Types. The JVM type system tracks only *raw* reference types, that is, class, interface and array types, without genericity. Source-level generic types are erased during the compilation process. The VM model described here only accounts for *declared types*, namely, class and interface types. Array types can be added back to the model with ease (see Sect. 6). As usual, $A <: B$ denotes the (reflexive and transitive) subtyping relation of reference types.

Object References. An *object reference* is an instance of exactly one reference type, called the *class* of that instance. An object has an arbitrary number of typed fields, each of which is declared either in the class of the object or in one of the supertypes. Each field in turn stores an object reference. Inspired by FJ, a field may only be initialized once but never updated. The null reference is modeled by the absence of link.

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

Heap. The *object pool* Π is a finite set of *allocations* $r : C$. Intuitively, Π records the objects that have been allocated by the VM, together with their classes. The *link graph* Γ is a finite set of

$$\begin{array}{c}
\frac{\Phi \vdash r : C \quad C <: B}{\langle \Pi, \Gamma; \Phi, A, \sigma \rangle \rightarrow \langle \Pi, \Gamma; \Phi \cup \{r : B\}, A, \sigma \rangle} \quad (\text{T-B-WIDEN}) \\
\\
\frac{r \text{ is a fresh object reference from } \mathcal{O}}{\langle \Pi, \Gamma; \Phi, A, \sigma \rangle \rightarrow \langle \Pi \cup \{r : B\}, \Gamma; \Phi \cup \{r : B\}, A, \sigma \rangle} \quad (\text{T-B-NEW}) \\
\\
\frac{\Phi \vdash r : C \quad \Pi \vdash r : C' \quad C' <: B}{\langle \Pi, \Gamma; \Phi, A, \sigma \rangle \rightarrow \langle \Pi, \Gamma; \Phi \cup \{r : B\}, A, \sigma \rangle} \quad (\text{T-B-CAST}) \\
\\
\frac{\Phi \vdash p : B_0 \quad B_0 <: B \quad \Gamma \vdash p : B \rightsquigarrow q : C}{\langle \Pi, \Gamma; \Phi, A, \sigma \rangle \rightarrow \langle \Pi, \Gamma; \Phi \cup \{q : C\}, A, \sigma \rangle} \quad (\text{T-B-GET}) \\
\\
\frac{\Phi \vdash p : B_0 \quad B_0 <: B \quad \Phi \vdash q : C}{\langle \Pi, \Gamma; \Phi, A, \sigma \rangle \rightarrow \langle \Pi, \Gamma \cup \{p : B \rightsquigarrow q : C\}; \Phi, A, \sigma \rangle} \quad (\text{T-B-PUT}) \\
\\
\frac{\begin{array}{c} \Phi \vdash r_0 : C_0 \quad C_0 <: B \quad \Phi \vdash \bar{r} : \bar{C} \\ \Pi \vdash r_0 : B'' \quad B'' <: B' \quad B' <: B \end{array}}{\langle \Pi, \Gamma; \Phi, A, \sigma \rangle \rightarrow \langle \Pi, \Gamma; \Phi', B', \sigma' \rangle} \quad (\text{T-B-INVOKE}) \\
\text{where } \Phi' = \{r_0 : B', \bar{r} : \bar{C}\} \text{ and } \sigma' = \text{push}(\Phi, A, C, \sigma) \\
\\
\frac{\Phi' \vdash r : C}{\langle \Pi, \Gamma; \Phi', B', \sigma' \rangle \rightarrow \langle \Pi, \Gamma; \Phi \cup \{r : C\}, A, \sigma \rangle} \quad (\text{T-B-RETURN}) \\
\text{where } \sigma' = \text{push}(\Phi, A, C, \sigma)
\end{array}$$

Figure 2: Transitions for Basic FJVM

links. A link $p : B \rightsquigarrow q : C$ asserts that p has a field declared in B , with field type C , storing the object reference q . Notice that fields are distinguishable only up to their declaring reference types, because our primary concern is to model reachability and confinement properties.

Stack. The *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 *execution context* A is the class in which the executing method is declared. Consequently, methods are distinguishable only up to their declaring classes. The *proper stack* σ models the call chain that leads to the current VM state. Specifically, σ is either an *empty stack*, \diamond , or a *non-empty*

$stack, push(\Phi, A, C, \sigma)$, 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.

Notations. We treat Π, Γ and Φ as finite sets. If $x \in X$, then we write the judgment $X \vdash x$, as in $\Pi \vdash r : C$. We also write \bar{x} for a list x_1, \dots, x_k . Obvious variations of these notations shall be clear from the context.

Instrumentation. FJVM 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 [43, 17, 18]) that assigns to every program point in an execution trace a *type state*. (A type state assigns a type label to every member of the local variable array and operand stack in a stack frame.)
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. 2 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. 2 define the state transition relation \rightarrow . The production T-B-WIDEN “promotes” the type label (C) of an object reference (r) in the stack frame (Φ) to a supertype label (B). This rule does not model a physical VM event, but instead captures the standard notion of *subsumption*. The transition rule T-B-NEW creates a fresh object reference (r) in the object pool (Π), and makes that reference accessible in the top stack frame (Φ). The rule T-B-CAST models dynamic type casting, and tags an object reference (r) in the stack frame (Φ) with an alternative type label (B) consistent with the class (C') of the object reference. (Note that the original type label C and the new type label B may or may not relate to one another by subtyping. For an example of the second case, consider a class C' that implements two interfaces C and B that are incomparable in the interface hierarchy.) The production T-B-GET models field getting, and makes the target (q) of an existing link ($p : B \rightsquigarrow q : C$) accessible in the current stack frame (Φ) if the source (p) of that link is accessible. The production T-B-PUT models field setting, and creates a link ($p : B \rightsquigarrow q : C$) between two object references (p, q) accessible in the current stack frame (Φ). Note that field identity has been abstracted away so that the model reflects every possible behavior of all properly typed bytecode programs. Consequently, the transition rule T-B-PUT does not specify a specific field be set. The same can be said about method identity in the following. The transition rule T-B-INVOKE models dynamic method dispatching: the caller (A) invokes a method declared in reference type B , with the actual dispatched method defined in reference type B' . The transition saves the caller stack frame (Φ), creates a new stack frame (Φ') for the callee, passes the receiver ($r_0 : C_0$) and the arguments ($\bar{r} : \bar{C}$) from the caller stack frame (Φ) to the callee stack frame (Φ'), and constrains the return type (C). Notice that the receiver r_0 receives a new type label B' inside the callee stack frame (Φ'). The transition rule T-B-RETURN pops the top stack frame (Φ'), resurrects the stack frame (Φ) of the caller (A), and makes the return

$$\begin{array}{c}
\frac{\forall p, q, B, C . (\Pi \vdash p : B \wedge \Pi \vdash q : C) \Rightarrow (p \neq q \vee B = C)}{\text{SafeHeap}(\Pi)} \\
\frac{\forall p, q, B, C, B' . (\Gamma \vdash p : B \rightsquigarrow q : C \wedge \Pi \vdash p : B') \Rightarrow B' <: B \quad \forall p, q, B, C, C' . (\Gamma \vdash p : B \rightsquigarrow q : C \wedge \Pi \vdash q : C') \Rightarrow C' <: C}{\text{SafeLinks}(\Gamma \mid \Pi)} \\
\frac{\forall r, C, C' . (\Phi \vdash r : C \wedge \Pi \vdash r : C') \Rightarrow C' <: C}{\text{SafeFrame}(\Phi \mid \Pi)} \\
\frac{}{\text{SafeStack}(\diamond \mid \Pi)} \\
\frac{\text{SafeFrame}(\Phi \mid \Pi) \quad \text{SafeStack}(\sigma \mid \Pi)}{\text{SafeStack}(\text{push}(\Phi, A, C, \sigma) \mid \Pi)} \\
\frac{\text{SafeHeap}(\Pi) \quad \text{SafeLinks}(\Gamma \mid \Pi) \quad \text{SafeFrame}(\Phi \mid \Pi) \quad \text{SafeStack}(\sigma \mid \Pi)}{\text{SafeState}(\langle \Pi, \Gamma; \Phi, A, \sigma \rangle)}
\end{array}$$

Figure 3: Type Safety Judgments

value ($r : C$) available in the caller stack frame.

3.2 A Type Safety Invariant

This section demonstrates that Basic FJVM preserves a notion of type safety: the type instrumentation in the model is always consistent with the actual class of allocated objects. Intuitively, the following two properties are desired: (i) every object reference in the object pool is associated with exactly one class, and (ii) every type label appearing in the object graph or a stack frame is either the class of the labeled reference or a supertype of that class. To this end, a number of type safety judgments are defined in Fig. 3. 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 la-

declared types	$A, B, C \in \mathcal{C}$
field designators	$f, g \in \mathcal{F}$
method designators	$m, n \in \mathcal{M}$
type annotations	$\alpha, \beta, \gamma \in \mathcal{A}$
object references	$p, q, r \in \mathcal{O}$
VM states	$S, T ::= \langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle$
object pools	$\Pi ::= \emptyset \mid \Pi \cup \{r : C\}$
link graphs	$\Gamma ::= \emptyset \mid \Gamma \cup \{p : B \xrightarrow{f} q : C\}$
stack frames	$\Phi ::= \emptyset \mid \Phi \cup \{r : C^\gamma\}$
proper stacks	$\sigma ::= \diamond \mid \text{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma)$
access events	$e \in \mathcal{E} ::= \text{widen}(C^\gamma)(B^\beta)$ $\mid \text{new}(B^\beta)$ $\mid \text{cast}(B^\beta)(C^\gamma)$ $\mid \text{get}(B.f : C^\gamma)(B_0^\beta)$ $\mid \text{put}(B.f : C^\gamma)(B_0^\beta)$ $\mid \text{invoke}(B.n : \overline{C}^{\overline{\gamma}/\overline{\beta}} \rightarrow C^{\beta/\gamma}[B'.n'](C_0^{\gamma_0/\beta_0}))$

Figure 4: States for Guarded FJVM

bels consistent with the references they annotate. A notion of type soundness can be proved for the productions on the basic model.

Theorem 1 (One-Step Soundness)

$$\forall S, T. (S \rightarrow T \wedge \text{SafeState}(S)) \Rightarrow \text{SafeState}(T)$$

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

4 Custom Instrumentation and Policy Enforcement

Basic FJVM provides a framework for articulating the dynamic behavior of the JVM. Our original goal, however, is to evaluate the effect of constraining the VM behavior via a type-based protection mechanism, and see if the type constraints are strong enough to uphold a given safety property. To this end, Basic FJVM is elaborated into *Guarded FJVM* (Figs. 4 and 5), which explicitly pro-

$$\begin{array}{c}
\frac{\Phi \vdash r : C^\gamma \quad C <: B}{\mathbf{widen}\langle C^\gamma \rangle(B^\beta) \in \Sigma[A.m]} \quad (\text{T-G-WIDEN}) \\
\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi \cup \{r : B^\beta\}, A.m, \sigma \rangle \\
\\
\frac{r \text{ is a fresh object reference from } \mathcal{O} \quad \mathbf{new}\langle B^\beta \rangle \in \Sigma[A.m]}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi \cup \{r : B\}, \Gamma; \Phi \cup \{r : B^\beta\}, A.m, \sigma \rangle} \quad (\text{T-G-NEW}) \\
\\
\frac{\Phi \vdash r : C^\gamma \quad \Pi \vdash r : C' \quad C' <: B}{\mathbf{cast}\langle B^\beta \rangle(C^\gamma) \in \Sigma[A.m]} \quad (\text{T-G-CAST}) \\
\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi \cup \{r : B^\beta\}, A.m, \sigma \rangle \\
\\
\frac{\Phi \vdash p : B_0^\beta \quad B_0 <: B \quad \Gamma \vdash p : B \xrightarrow{f} q : C}{\mathbf{get}\langle B.f : C^\gamma \rangle(B_0^\beta) \in \Sigma[A.m]} \quad (\text{T-G-GET}) \\
\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi \cup \{q : C^\gamma\}, A.m, \sigma \rangle \\
\\
\frac{\Phi \vdash p : B_0^\beta \quad B_0 <: B \quad \Phi \vdash q : C^\gamma}{\mathbf{put}\langle B.f : C^\gamma \rangle(B_0^\beta) \in \Sigma[A.m]} \quad (\text{T-G-PUT}) \\
\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma \cup \{p : B \xrightarrow{f} q : C\}; \Phi, A.m, \sigma \rangle \\
\\
\frac{\begin{array}{c} \Phi \vdash r_0 : C_0^{\gamma_0} \quad C_0 <: B \quad \Phi \vdash \bar{r} : \bar{C}^{\bar{\gamma}} \\ \Pi \vdash r_0 : B'' \quad B'' <: B' \quad B' <: B \end{array} \quad \mathbf{invoke}\langle B.n : \bar{C}^{\bar{\gamma}/\bar{\beta}} \rightarrow C^{\beta/\gamma} \rangle[B'.n'](C_0^{\gamma_0/\beta_0}) \in \Sigma[A.m]}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi', B'.n', \sigma' \rangle} \quad (\text{T-G-INVOKE}) \\
\text{where } \Phi' = \{r_0 : B'^{\beta_0}, \bar{r} : \bar{C}^{\bar{\beta}}\} \text{ and } \sigma' = \text{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma) \\
\\
\frac{\Phi' \vdash r : C^\beta}{\langle \Pi, \Gamma; \Phi', B'.n', \sigma' \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi \cup \{r : C^\gamma\}, A.m, \sigma \rangle} \quad (\text{T-G-RETURN}) \\
\text{where } \sigma' = \text{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma)
\end{array}$$

Figure 5: Transitions for Guarded FJVM

vides “*hooks*” for introducing behavioral constraints that are expressed in terms of domain-specific instrumentation.

Field and Method Designators. In Basic FJVM, 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, etc). 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*. *Method designators* 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 $p : B \xrightarrow{f} q : C$. (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 designator, as in $A.m$. (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 are 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*, as in $\Phi \vdash r : C^\gamma$. Whenever a labeled reference is introduced into the top stack frame, as in the cases of T-G-WIDEN, T-G-NEW, T-G-CAST, and T-G-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 T-G-PUT. The most complex of all the transition rules are the pair T-G-INVOKE and T-G-RETURN. As in other transition rules, when arguments $(\bar{r} : \bar{C}^\gamma)$ are passed into the callee stack frame (Φ') , their type labels acquire a new set of type annotations $(\bar{r} : \bar{C}^{\beta})$. 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'). A subtle 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 Basic FJVM (Figs. 1 and 2), method invocation pushes into the run-time stack the return type label C . In Guarded FJVM, 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 T-G-INVOKE and T-G-RETURN are designed to jointly produce this behavior. (As usual, if $|\mathcal{A}| = 1$ then all type annotations may be omitted.)

Policy Enforcement. The purpose of the aforementioned apparatus is to allow us to control the execution traces that the VM generates. In the Guarded FJVM, the transition relation \rightarrow_Σ is 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. The transition rules in Fig. 5 ensure that every transition produced by \rightarrow_Σ 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 [12], 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 Guarded FJVM 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 untrusted code outside of the platform library cannot tamper with the private state of the authorization procedure. 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 [44].

The idea of Confined Types [44, 20, 47, 48] 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 [44]. A class or interface type may be declared to be *confined*. The typing discipline ensures

-
- Ⓒ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.
 - Ⓒ2 A confined type must not be public.
 - Ⓒ3 Methods invoked on an expression of confined type must either be defined in a confined class or be anonymous methods.
 - Ⓒ4 Subtypes of a confined type must be confined [in the same package as the confined type].
 - Ⓒ5 Confined types can be widened only to other confined types.
 - Ⓒ6 Overriding must preserve anonymity of methods.
 - Ⓐ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 6: Type Rules for Confined Types [48]

the following property.

Confinement Property (Informal) [47, 48] An object of confined type is encapsulated within its defining scope [i.e., package].

Confined Types enforces the typing discipline of Fig. 6 (adopted from [48], 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 Boyland’s borrowed receiver [7].

5.2 A Safety Policy for Confined Types

We encode the type rules of Fig. 6 into a safety policy for Guarded FJVM (Fig. 7).

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 [47, 48], type rule $\mathcal{C}2$ is modeled by identifying the package private classes with the confined classes. The following shorthand is therefore defined:

$$confined(C) \triangleq \neg public(C)$$

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 .

$$C \triangleright B \triangleq public(C) \vee B \approx C$$

Notice that the visibility relation (\triangleright) is not transitive. (To see this, consider declared types A , B and C such that $B \approx C$, $A \not\approx C$, $confined(C)$ and $public(B)$. We thus have $C \triangleright B$ and $B \triangleright A$ but not $C \triangleright A$.) We define a transitive variant of the visibility relation:

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

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:

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

Following [47, 48], 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 declared by the programmer to be anonymous:

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

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:

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

The type annotation \top indicates no information for the underlying reference, while **this** (resp. $\overline{\mathbf{this}}$) indicates that the underlying reference is (resp. is not) the `this` pseudo-parameter of a method. These type annotations are not supplied by the programmer. Instead, they are generated and tracked

$$\begin{array}{c}
\frac{B \triangleright A \quad \gamma \sqsubseteq: \beta \quad C \blacktriangleright B \vee (\beta \neq \overline{\mathbf{this}} \wedge m = \mathbf{anon})}{\mathbf{widen}(C^\gamma)(B^\beta) \in \Sigma[A.m]} \quad (\text{P-CT-WIDEN}) \\
\frac{B \triangleright A \quad \beta = \overline{\mathbf{this}}}{\mathbf{new}(B^\beta) \in \Sigma[A.m]} \quad (\text{P-CT-NEW}) \\
\frac{B \triangleright A \quad \gamma \sqsubseteq: \beta \quad C \blacktriangleright B \vee (\beta \neq \overline{\mathbf{this}} \wedge m = \mathbf{anon})}{\mathbf{cast}(B^\beta)(C^\gamma) \in \Sigma[A.m]} \quad (\text{P-CT-CAST}) \\
\frac{B \triangleright A \quad C \blacktriangleright B \quad \gamma = \overline{\mathbf{this}}}{\mathbf{get}(B : C^\gamma)(B_0^\beta) \in \Sigma[A.m]} \quad (\text{P-CT-GET}) \\
\frac{B \triangleright A \quad C \blacktriangleright B \quad \gamma = \overline{\mathbf{this}} \vee m \neq \mathbf{anon}}{\mathbf{put}(B : C^\gamma)(B_0^\beta) \in \Sigma[A.m]} \quad (\text{P-CT-PUT}) \\
\frac{\begin{array}{l} B \triangleright A \quad n = \mathbf{anon} \Rightarrow n' = \mathbf{anon} \\ (C_0 \blacktriangleright B \wedge (\gamma_0 = \overline{\mathbf{this}} \vee m \neq \mathbf{anon})) \vee n = \mathbf{anon} \\ \forall i > 0. (C_i \blacktriangleright B \wedge (\gamma_i = \overline{\mathbf{this}} \vee m \neq \mathbf{anon})) \\ \beta_0 = \overline{\mathbf{this}} \quad \forall i > 0. \beta_i = \overline{\mathbf{this}} \\ C \blacktriangleright B \quad \beta = \overline{\mathbf{this}} \vee n' \neq \mathbf{anon} \quad \gamma = \overline{\mathbf{this}} \end{array}}{\mathbf{invoke}(B.n : \overline{C^{\gamma/\beta}} \rightarrow C^{\beta/\gamma})[B'.n'](C_0^{\gamma_0/\beta_0}) \in \Sigma[A.m]} \quad (\text{P-CT-INVOKE})
\end{array}$$

Figure 7: A Safety Policy for Confined Types

by the underlying type-based protection mechanism (e.g., via dataflow analysis). A subsumption relation $\sqsubseteq:$ is defined for the type annotations, so that $\sqsubseteq:$ is a partial ordering with \top being the maximal element:

$$\mathbf{this} \sqsubseteq: \top \quad \overline{\mathbf{this}} \sqsubseteq: \top$$

5.2.2 Safety Policy

A safety policy for Confined Types is formulated in Fig. 7, which captures the effect of imposing the type rules in Fig. 6. The type rule $\mathcal{C}1$ is enforced by the antecedent $C \blacktriangleright B$ of P-CT-GET, P-CT-PUT and P-CT-INVOKE, 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 P-CT-INVOKE by the

antecedent $(C_0 \blacktriangleright B \wedge \dots) \vee n = \mathbf{anon}$, which requires that a confined receiver can temporarily escape from its confinement domain only if the invoked method is anonymous. The policy rules P-CT-WIDEN and P-CT-CAST 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 P-CT-INVOKE by the antecedent $n = \mathbf{anon} \Rightarrow n' = \mathbf{anon}$, which mandates that method dispatching preserves anonymity in the execution context. To enforce type rule $\mathcal{A}1$, the antecedent $\gamma = \overline{\mathbf{this}} \vee m \neq \mathbf{anon}$ is required in P-CT-PUT so that an anonymous method cannot store `this` into a field. Similarly, the antecedent $\forall i > 0 . (\dots \wedge (\gamma_i = \overline{\mathbf{this}} \vee m \neq \mathbf{anon}))$ disallows the passing of an anonymous `this` as method arguments. Within the same policy rule, the antecedent $\beta = \overline{\mathbf{this}} \vee n' \neq \mathbf{anon}$ forbids the returning of an anonymous `this`.

A number of subtle requirements in the policy of Fig. 7 are not explicitly mandated by the type rules of Fig. 6. 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 [29, Sect. 5.4.3]. This property is exploited in the confinement proof. The antecedent $\forall i > 0 . (C_i \blacktriangleright B \wedge \dots)$ in P-CT-INVOKE mandates that methods may only receive arguments from safe origins³. 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 $(\dots \wedge (\gamma_0 = \overline{\mathbf{this}} \vee m \neq \mathbf{anon})) \vee n = \mathbf{anon}$ of P-CT-INVOKE).

5.2.3 Confinement Theorem

The goal of imposing the safety policy of Fig. 7 is to ensure that the **Confinement Property (Informal)** of Sect. 5.1 is satisfied. Formally, given a VM state $\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle$, we want the

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

$$\begin{array}{c}
\forall p, q, B, C . \Gamma \vdash p : B \rightsquigarrow q : C \Rightarrow C \blacktriangleright B \\
\forall p, q, B, C, C' . (\Gamma \vdash p : B \rightsquigarrow q : C \wedge \Pi \vdash q : C') \Rightarrow C' \blacktriangleright C \\
\hline
\text{ConfinedLinks}(\Gamma \mid \Pi) \\
\\
\forall r, C . \Phi \vdash r : C^\gamma \Rightarrow C \triangleright A \\
\forall r, C, C' . (\Phi \vdash r : C^\gamma \wedge \Pi \vdash r : C') \\
\Rightarrow (C' \blacktriangleright C \vee (\gamma \neq \overline{\text{this}} \wedge m = \text{anon})) \\
\hline
\text{ConfinedFrame}(\Phi \mid \Pi, A.m) \\
\\
\overline{\text{ConfinedStack}(\diamond \mid \Pi, A.m)} \\
\\
\begin{array}{c}
C \triangleright A \quad \beta = \overline{\text{this}} \vee n' \neq \text{anon} \quad \gamma = \overline{\text{this}} \\
\text{ConfinedFrame}(\Phi \mid \Pi, A.m) \quad \text{ConfinedStack}(\sigma \mid \Pi, A.m) \\
\hline
\text{ConfinedStack}(\text{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma) \mid \Pi, B'.n') \\
\\
\text{ConfinedLinks}(\Gamma \mid \Pi) \\
\text{ConfinedFrame}(\Phi \mid \Pi, A.m) \quad \text{ConfinedStack}(\sigma \mid \Pi, A.m) \\
\hline
\text{ConfinedState}(\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle)
\end{array}
\end{array}$$

Figure 8: Confinement Judgments

following properties to hold:

$$\forall p, q, B, B', C, C' . (\Gamma \vdash p : B \rightsquigarrow q : C \wedge \Pi \vdash p : B' \wedge \Pi \vdash q : C') \Rightarrow C' \triangleright B' \quad (7)$$

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

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 remain in the stack frame of methods declared in the same package. To enforce the above properties, we postulate the state invariant $\text{ConfinedState}(\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle)$ as specified in Fig. 8.

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

Proof: Property (7) follows from $\text{ConfinedLinks}(\Gamma \mid \Pi)$ and $\text{SafeLinks}(\Gamma \mid \Pi)$ via (2), (6), and (3).

Property (8) follows from $\text{ConfinedFrame}(\Phi \mid \Pi, A.m)$ via (5).

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

Theorem 3 (Confinement)

$$\forall S, T . (S \rightarrow_{\Sigma} T \wedge \text{SafeState}(S) \wedge \text{ConfinedState}(S)) \Rightarrow \text{ConfinedState}(T)$$

See Appendix B for a proof of this theorem. Notice, in particular, the mild complexity of the proof.

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. 6. Although the safety policy in Fig. 7 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 [29], 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 [10, 8]. 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 the type’s confined-ness. We envision that this capability annotation is generated by the code producer using a customized compiler. Although the capabilities are only estimates, and neither the code producer nor the compiler is to be trusted, nevertheless the design of the typing

discipline is such that a classfile that is not honest about the annotations will fail to type check. As we shall see in Sect. 5.3.5, the availability of these estimates effectively reduces the amount of class loading. Implementation details for this approach can be found in [14]. This capability-based formulation of Confined Types is formalized in the following, with the dual purpose of establishing its confinement guarantee as well as demonstrating the utility of FJVM.

5.3.2 Notation

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

$$\alpha, \beta, \gamma ::= \perp \mid \mathbf{conf} \mid \mathbf{anon}$$

$$\perp \sqsubseteq \mathbf{conf} \sqsubseteq \mathbf{anon}$$

The capability \perp 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 these capabilities. It is assumed that a custom compiler will compile source-level confined type annotations into these capability annotations, and subsequently inject them into the generated classfile.

We use field designators to record the capability annotations of fields.

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

As we shall see, the safety policy will not permit the use of **anon** to annotate fields. Similarly,

method designators record the capability annotation of the pseudo-parameter `this`.

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

An anonymous method is represented by having a method designator **anon**. If a method designator is not **anon**, then it is an estimate of the confined-ness of `this`. Again, annotations represented by field and method designators are generated by the compiler.

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:

$$C^\gamma \blacktriangleright B^\beta \triangleq \gamma \sqsubseteq \beta \wedge (\gamma \neq \mathbf{conf} \vee \beta \neq \mathbf{conf} \vee C \approx B)$$

$$\begin{array}{c}
\frac{B^\beta \triangleright: A \vee \beta = \mathbf{anon} \quad C^\gamma \blacktriangleright: B^\beta}{\mathbf{widen}(C^\gamma)(B^\beta) \in \Sigma[A.\alpha]} \quad (\text{P-CAP-WIDEN}) \\
\\
\frac{B^* \triangleright: A}{\mathbf{new}\langle B^* \rangle \in \Sigma[A.\alpha]} \quad (\text{P-CAP-NEW}) \\
\\
\frac{B^\beta \triangleright: A \vee \beta = \mathbf{anon} \quad C^\gamma \blacktriangleright: B^\beta}{\mathbf{cast}\langle B^\beta \rangle(C^\gamma) \in \Sigma[A.\alpha]} \quad (\text{P-CAP-CAST}) \\
\\
\frac{B^* \triangleright: A \quad C^\gamma \blacktriangleright: B^*}{\mathbf{get}\langle B.\gamma : C^\gamma \rangle(B_0^\beta) \in \Sigma[A.\alpha]} \quad (\text{P-CAP-GET}) \\
\\
\frac{B^* \triangleright: A \quad C^\gamma \blacktriangleright: B^*}{\mathbf{put}\langle B.\gamma : C^\gamma \rangle(B_0^\beta) \in \Sigma[A.\alpha]} \quad (\text{P-CAP-PUT}) \\
\\
\frac{B^* \triangleright: A \quad C_0^{\gamma_0} \blacktriangleright: B^{\beta_0} \quad B^{\beta_0} \blacktriangleright: B'^{\beta'_0} \quad \forall i > 0. C_i^{\gamma_i} \blacktriangleright: B^* \quad C^\gamma \blacktriangleright: B^*}{\mathbf{invoke}\langle B.\beta_0 : \overline{C}^{\overline{\gamma}/\overline{\gamma}} \rightarrow C^{\gamma/\gamma} \rangle[B'.\beta'_0](C_0^{\gamma_0/\beta'_0}) \in \Sigma[A.\alpha]} \quad (\text{P-CAP-INVOKE})
\end{array}$$

Figure 9: A Capability-Based Safety Policy for Confined Types

The following properties can be easily validated:

$$C^\gamma \blacktriangleright: C^\gamma \quad (9)$$

$$C^\gamma \blacktriangleright: B^\beta \wedge B^\beta \blacktriangleright: A^\alpha \Rightarrow C^\gamma \blacktriangleright: A^\alpha \quad (10)$$

$$C^* \blacktriangleright: B^* \Leftrightarrow C \blacktriangleright B \quad (11)$$

$$C^\gamma \blacktriangleright: B^* \Rightarrow C^\gamma \triangleright: B \quad (12)$$

$$C^\gamma \blacktriangleright: B^\beta \wedge B^\beta \triangleright: A \Rightarrow C^\gamma \triangleright: A \quad (13)$$

$$C^* \triangleright: B \Leftrightarrow C \triangleright B \quad (14)$$

$$B^\beta \triangleright: B \vee \beta = \mathbf{anon} \quad (15)$$

$$\begin{array}{c}
\frac{\forall p, q, B, C . \Gamma \vdash p : B \rightsquigarrow q : C \Rightarrow C^\gamma \blacktriangleright : B^*}{\text{ConfinedLinks}_{cap}(\Gamma \mid \Pi)} \\
\frac{\forall r, C . \Phi \vdash r : C^\gamma \Rightarrow (C^\gamma \triangleright : A \vee \gamma = \mathbf{anon})}{\text{ConfinedFrame}_{cap}(\Phi \mid \Pi, A)} \\
\frac{}{\text{ConfinedStack}_{cap}(\diamond \mid \Pi)} \\
\frac{C^\gamma \triangleright : A \quad \text{ConfinedFrame}_{cap}(\Phi \mid \Pi, A) \quad \text{ConfinedStack}_{cap}(\sigma \mid \Pi)}{\text{ConfinedStack}_{cap}(\text{push}(\Phi, A, \alpha, C^{\gamma/\gamma}, \sigma) \mid \Pi)} \\
\frac{\text{ConfinedLinks}_{cap}(\Gamma \mid \Pi) \quad \text{ConfinedFrame}_{cap}(\Phi \mid \Pi, A) \quad \text{ConfinedStack}_{cap}(\sigma \mid \Pi)}{\text{ConfinedState}_{cap}(\langle \Pi, \Gamma; \Phi, A, \alpha, \sigma \rangle)}
\end{array}$$

Figure 10: Revised Confinement Judgments

5.3.3 Safety Policy

A capability-based safety policy for Confined Types is given in Fig. 9. This policy is significantly cleaner than the previous one⁴. 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 cuts 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:

$$\forall p, C, C', \gamma . (\Phi \vdash p : C^\gamma \wedge \Pi \vdash p : C' \wedge \gamma \neq \mathbf{anon}) \Rightarrow C' \triangleright A \quad (16)$$

⁴An 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.

To accommodate the variation in notation and confinement goals, the confinement invariant has been reformulated, as shown in Fig. 10. Again, we assert that the invariant establishes the confinement goals.

Proposition 4 *If both $\text{SafeState}(\langle \Pi, \Gamma; \Phi, A.\alpha, \sigma \rangle)$ and $\text{ConfinedState}_{cap}(\langle \Pi, \Gamma; \Phi, A.\alpha, \sigma \rangle)$ hold, then properties (7) and (16) hold.*

Proof: Property (7) follows from $\text{ConfinedLinks}_{cap}(\Gamma \mid \Pi)$ and $\text{SafeLinks}(\Gamma \mid \Pi)$ via (10), (11), (6), (2), and (3). Property (8) follows from $\text{ConfinedFrame}_{cap}(\Phi \mid \Pi, A)$ via (13) and (14).

A revised confinement theorem can be proved:

Theorem 5 (Confinement (Revised))

$$\forall S, T. (S \rightarrow_{\Sigma} T \wedge \text{SafeState}(S) \wedge \text{ConfinedState}_{cap}(S)) \Rightarrow \text{ConfinedState}_{cap}(T)$$

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 is not always necessary. For example, checking $C^{\gamma} \blacktriangleright B^{\beta}$ sometimes involves only the comparison of capability labels. It is only when (i) both γ and β 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 package⁵.

⁵The *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 [29, Sect. 5.3].

Class loading can be avoided in all other cases.

6 Extensions

To facilitate exposition, several key features of the JVM have been abstracted away in the previous discussion. This section examines a number of extensions to Guarded FJVM, including static fields, static methods, arrays, and exception handling. The resulting VM model, *Extended FJVM* is summarized in Figs. 11, 12, 13 and 14. The goal of this exercise is to demonstrate that the modeling approach adopted so far applies equally well to other aspects of the JVM.

A cross-cutting extension is that array types are explicitly modeled. Reference types are either declared types or array types (Fig. 11). The subtyping relation is extended to model the covariant array subtyping rule of the JVM type system (Fig. 12). As a result of this extension, field types, method return types and formal parameter types can be arbitrary reference types instead of mere declared types. The change is reflected in Fig. 13, which shows a revision of the transition rules inherited from Guarded FJVM. To simplify the model, it is assumed that no array type is a subtype of any declared type. This assumption does not fully reflect the type structure of the JVM, but its adoption significantly simplifies the model.

New transitions are defined in Fig. 14. To model static fields, a link graph carries *global links* of the form $B \xrightarrow{f} q : R$. The transitions T-E-GETSTATIC, T-E-PUTSTATIC and T-E-INVOKESTATIC are straightforward simplifications of T-E-GET, T-E-PUT and T-E-INVOKE. A recurring motif is that link and execution contexts are distinguishable only up to field and method designators respectively. The identity of concrete fields and methods is abstracted away. As well, destructive updates are not modeled, and control flow remains nondeterministic.

To model arrays, a link graph records *array links* of the form $p : [] \rightsquigarrow q$. Array creation, read and write are modeled respectively by the transitions T-E-NEWARRAY, T-E-GETARRAY and T-E-PUTARRAY. Notice that, because array subtyping is covariant, and thus unsound, the

declared types	$A, B, C \in \mathcal{C}$
reference types	$P, Q, R \in A \mid R[]$
field designators	$f, g \in \mathcal{F}$
method designators	$m, n \in \mathcal{M}$
type annotations	$\alpha, \beta, \gamma \in \mathcal{A}$
object references	$p, q, r \in \mathcal{O}$
VM states	$S, T ::= \langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle$
object pools	$\Pi ::= \emptyset \mid \Pi \cup \{r : R\}$
link graphs	$\Gamma ::= \emptyset$ $\mid \Gamma \cup \{B \xrightarrow{f} q : R\}$ $\mid \Gamma \cup \{p : B \xrightarrow{f} q : R\}$ $\mid \Gamma \cup \{p : [] \rightsquigarrow q\}$
stack frames	$\Phi ::= \emptyset \mid \Phi \cup \{r : R^\gamma\}$
proper stacks	$\sigma ::= \diamond \mid \text{push}(\Phi, A.m, R^{\beta/\gamma}, \sigma)$
access events	$e \in \mathcal{E} ::= \mathbf{widen}(R^\gamma)(Q^\beta)$ $\mid \mathbf{new}(B^\beta)$ $\mid \mathbf{cast}(Q^\beta)(R^\gamma)$ $\mid \mathbf{get}(B.f : R^\gamma)(B_0^\beta)$ $\mid \mathbf{put}(B.f : R^\gamma)(B_0^\beta)$ $\mid \mathbf{invoke}(B.n : \overline{R}^{\gamma/\beta} \rightarrow R^{\beta/\gamma})[B'.n'](C_0^{\gamma_0/\beta_0})$ $\mid \mathbf{getstatic}(B.f : R^\gamma)$ $\mid \mathbf{putstatic}(B.f : R^\gamma)$ $\mid \mathbf{invokestatic}(B.n : \overline{R}^{\gamma/\beta} \rightarrow R^{\beta/\gamma})$ $\mid \mathbf{newarray}(R[]^\gamma)$ $\mid \mathbf{getarray}(R[]^\gamma)(R^\beta)$ $\mid \mathbf{putarray}(R[]^\gamma)(R^\beta)$ $\mid \mathbf{throw}(C^\gamma)(B^\beta)$

Figure 11: States for Extended FJVM

$$\begin{array}{c}
\frac{}{R <: R} \quad \frac{P <: Q \quad Q <: R}{P <: R} \quad \frac{P <: Q}{P[] <: Q[]} \\
\hline
\frac{\text{unwind}(\Phi, A.m, \sigma ; \Phi, A.m, \sigma) \quad \text{unwind}(\Phi', A'.m', \sigma' ; \Phi'', A''.m'', \sigma'')}{\text{unwind}(\Phi, A.m, \text{push}(\Phi', A'.m', R'^{\beta/\gamma}, \sigma') ; \Phi'', A''.m'', \sigma'')}
\end{array}$$

Figure 12: Auxiliary Judgments for Extended FJVM

JVM performs a dynamic check when a reference is stored into an array. This dynamic check is explicitly modeled in T-E-PUTARRAY. As in the rest of this paper, our concerns are reachability and confinement properties. Array capacity and indexing are abstracted away; only reachability information is preserved.

Lastly, to model exception handling, an auxiliary judgment *unwind* is defined to capture stack unwinding (Fig. 12). Throughout this work, the intricacy of control flow has been abstracted away by non-determinism. Here, stack unwinding is also performed non-deterministically. One of the stack frames in the call chain is selected to catch the exception with an appropriate catch type. The choices of the stack frame and the catch type are both non-deterministic.

Given the revised *SafeState* judgment in Fig. 15, one can easily show that the formulation of Extended FJVM preserves **One-Step Soundness**:

Theorem 6 (One-Step Soundness (Extended))

$$\forall S, T . (S \rightarrow_{\Sigma} T \wedge \text{SafeState}(S)) \Rightarrow \text{SafeState}(T)$$

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

$$\begin{array}{c}
\frac{\Phi \vdash r : R^\gamma \quad R <: Q \quad \mathbf{widen}\langle R^\gamma \rangle(Q^\beta) \in \Sigma[A.m]}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi \cup \{r : Q^\beta\}, A.m, \sigma \rangle} \quad (\text{T-E-WIDEN}) \\
\\
\frac{r \text{ is a fresh object reference from } \mathcal{O} \quad \mathbf{new}\langle B^\beta \rangle \in \Sigma[A.m]}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi \cup \{r : B\}, \Gamma; \Phi \cup \{r : B^\beta\}, A.m, \sigma \rangle} \quad (\text{T-E-NEW}) \\
\\
\frac{\Phi \vdash r : R^\gamma \quad \Pi \vdash r : R' \quad R' <: Q \quad \mathbf{cast}\langle Q^\beta \rangle(R^\gamma) \in \Sigma[A.m]}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi \cup \{r : Q^\beta\}, A.m, \sigma \rangle} \quad (\text{T-E-CAST}) \\
\\
\frac{\Phi \vdash p : B_0^\beta \quad B_0 <: B \quad \Gamma \vdash p : B \xrightarrow{f} q : R \quad \mathbf{get}\langle B.f : R^\gamma \rangle(B_0^\beta) \in \Sigma[A.m]}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi \cup \{q : R^\gamma\}, A.m, \sigma \rangle} \quad (\text{T-E-GET}) \\
\\
\frac{\Phi \vdash p : B_0^\beta \quad B_0 <: B \quad \Phi \vdash q : R^\gamma \quad \mathbf{put}\langle B.f : R^\gamma \rangle(B_0^\beta) \in \Sigma[A.m]}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma \cup \{p : B \xrightarrow{f} q : R\}; \Phi, A.m, \sigma \rangle} \quad (\text{T-E-PUT}) \\
\\
\frac{\Phi \vdash r_0 : C_0^{\gamma_0} \quad C_0 <: B \quad \Phi \vdash \bar{r} : \bar{R}^{\bar{\gamma}} \quad \Pi \vdash r_0 : B'' \quad B'' <: B' \quad B' <: B \quad \mathbf{invoke}\langle B.n : \bar{R}^{\bar{\gamma}/\bar{\beta}} \rightarrow R^{\beta/\gamma} \rangle[B'.n'](C_0^{\gamma_0/\beta_0}) \in \Sigma[A.m]}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi', B'.n', \sigma' \rangle} \quad (\text{T-E-VOKE}) \\
\text{where } \Phi' = \{r_0 : B'^{\beta_0}, \bar{r} : \bar{R}^{\bar{\beta}}\} \text{ and } \sigma' = \text{push}(\Phi, A.m, R^{\beta/\gamma}, \sigma) \\
\\
\frac{\Phi' \vdash r : R^\beta}{\langle \Pi, \Gamma; \Phi', B'.n', \sigma' \rangle \rightarrow_\Sigma \langle \Pi, \Gamma; \Phi \cup \{r : R^\gamma\}, A.m, \sigma \rangle} \quad (\text{T-E-RETURN}) \\
\text{where } \sigma' = \text{push}(\Phi, A.m, R^{\beta/\gamma}, \sigma)
\end{array}$$

Figure 13: Revised Transitions for Extended FJVM

$$\begin{array}{c}
\frac{\Gamma \vdash B \xrightarrow{f} q : R}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_{\Sigma} \langle \Pi, \Gamma; \Phi \cup \{q : R^{\gamma}\}, A.m, \sigma \rangle} \text{getstatic}\langle B.f : R^{\gamma} \rangle \in \Sigma[A.m] \quad (\text{T-E-GETSTATIC}) \\
\frac{\Phi \vdash q : R^{\gamma}}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_{\Sigma} \langle \Pi, \Gamma \cup \{B \xrightarrow{f} q : R\}; \Phi, A.m, \sigma \rangle} \text{getstatic}\langle B.f : R^{\gamma} \rangle \in \Sigma[A.m] \quad (\text{T-E-PUTSTATIC}) \\
\frac{\Phi \vdash \bar{r} : \overline{R^{\gamma}}}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_{\Sigma} \langle \Pi, \Gamma; \Phi', B'.n', \sigma' \rangle} \text{invokestatic}\langle B'.n' : \overline{R^{\gamma}/\beta} \rightarrow R^{\beta/\gamma} \rangle \in \Sigma[A.m] \quad (\text{T-E-INVOKESTATIC}) \\
\text{where } \Phi' = \{\bar{r} : \overline{R^{\gamma}}\} \text{ and } \sigma' = \text{push}(\Phi, A.m, R^{\beta/\gamma}, \sigma) \\
r \text{ is a fresh object reference from } \mathcal{O} \\
\frac{\text{newarray}\langle R[]^{\gamma} \rangle \in \Sigma[A.m]}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_{\Sigma} \langle \Pi \cup \{r : R[]\}, \Gamma; \Phi \cup \{r : R[]^{\gamma}\}, A.m, \sigma \rangle} \quad (\text{T-E-NEWARRAY}) \\
\frac{\Phi \vdash p : R[]^{\gamma} \quad \Gamma \vdash p : [] \rightsquigarrow q}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_{\Sigma} \langle \Pi, \Gamma; \Phi \cup \{q : R^{\beta}\}, A.m, \sigma \rangle} \text{getarray}(R[]^{\gamma})(R^{\beta}) \in \Sigma[A.m] \quad (\text{T-E-GETARRAY}) \\
\frac{\Phi \vdash p : R[]^{\gamma} \quad \Phi \vdash q : R^{\beta} \quad \Pi \vdash p : P[] \quad \Pi \vdash q : Q \quad Q <: P}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_{\Sigma} \langle \Pi, \Gamma \cup \{p : [] \rightsquigarrow q\}; \Phi, A.m, \sigma \rangle} \text{putarray}(R[]^{\gamma})(R^{\beta}) \in \Sigma[A.m] \quad (\text{T-E-PUTARRAY}) \\
\frac{\Phi \vdash r : C^{\gamma} \quad \Pi \vdash r : C' \quad C' <: B \quad \text{unwind}(\Phi, A.m, \sigma ; \Phi', A'.m', \sigma')}{\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle \rightarrow_{\Sigma} \langle \Pi, \Gamma; \Phi' \cup \{r : B^{\beta}\}, A'.m', \sigma' \rangle} \text{throw}(C^{\gamma})(B^{\beta}) \in \Sigma[A.m] \quad (\text{T-E-THROW})
\end{array}$$

Figure 14: New Transitions for Extended FJVM

$$\begin{array}{c}
\frac{\forall p, q, P, Q . (\Pi \vdash p : P \wedge \Pi \vdash q : Q) \Rightarrow (p \neq q \vee P = Q)}{\text{SafeHeap}(\Pi)} \\
\\
\frac{\forall p, q, f, B, Q, B' . (\Gamma \vdash p : B \xrightarrow{f} q : Q \wedge \Pi \vdash p : B') \Rightarrow B' <: B \\
\forall p, q, f, B, Q, Q' . (\Gamma \vdash p : B \xrightarrow{f} q : Q \wedge \Pi \vdash q : Q') \Rightarrow Q' <: Q \\
\forall p, q, P, Q . (\Gamma \vdash p : [] \rightsquigarrow q \wedge \Pi \vdash p : P[] \wedge \Pi \vdash q : Q) \Rightarrow Q <: P}{\text{SafeLinks}(\Gamma | \Pi)} \\
\\
\frac{\forall r, R, R' . (\Phi \vdash r : R^\gamma \wedge \Pi \vdash r : R') \Rightarrow R' <: R}{\text{SafeFrame}(\Phi | \Pi)} \\
\\
\frac{}{\text{SafeStack}(\diamond | \Pi)} \\
\\
\frac{\text{SafeFrame}(\Phi | \Pi) \quad \text{SafeStack}(\sigma | \Pi)}{\text{SafeStack}(\text{push}(\Phi, A.m, R^{\beta/\gamma}, \sigma) | \Pi)} \\
\\
\frac{\text{SafeHeap}(\Pi) \quad \text{SafeLinks}(\Gamma | \Pi) \quad \text{SafeFrame}(\Phi | \Pi) \quad \text{SafeStack}(\sigma | \Pi)}{\text{SafeState}(\langle \Pi, \Gamma; \Phi, A.m, \sigma \rangle)}
\end{array}$$

Figure 15: Extended Type Safety Judgments

7 Conclusion and Future Work

In this paper a lightweight formal model of JVM-like environments is proposed for evaluating the effectiveness of type-based protection mechanisms at an early stage of design. A rational reconstruction of Confined Types at the JVM bytecode level is presented, and this formulation is verified to successfully enforce the **Confinement Property**. Also articulated is a novel 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. This paper has therefore provided first evidence on the utility of FJVM as a tool for lightweight evaluation of type-based protection mechanisms. Lastly, it has been demonstrated that FJVM can be extended to model language features such as static members, arrays and exception handling.

A number of future directions are suggested by this work. Firstly, the author is particularly interested in the application of type-based access control mechanisms to enforce security properties at the bytecode level. The Featherweight JVM will be employed to validate early designs. [15]

represents a first example of this approach. Secondly, the author plans to establish the soundness of FJVM with respect to a more standard operational semantic model of the JVM, so that stack invariants and confinement properties established in the FJVM can be transferred to the semantic model. To this end, the following plan will be adopted. A family of type systems will be defined over the standard semantic model of the JVM. A standard translation of the instances of this type system family to their corresponding FJVM safety policies will be specified. It will then be shown that stack invariants and reachability properties provable in the standard semantic model are preserved in the policy-guarded FJVM. This amounts to showing that FJVM is an abstract interpretation of a standard VM model. Notice that the nondeterministic nature of FJVM is compatible with a semantic model equipped with a rich set of control transfer primitives. The link graph can be seen as an abstraction of the heap, in which only may-reach information is tracked. Thirdly, the author plans to embed FJVM into a programming logic such as ACL2 [25], so as to provide mechanized theorem proving support to the users of FJVM.

References

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

- [4] Gilles Barthe and Tamara Rezk. Non-interference for a JVM-like language. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 103–112, Long Beach, California, USA, January 2005.
- [5] Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. In *Proceedings of the Workshop on Foundations of Computer Security (FCS'02)*, Copenhagen, Denmark, July 2002.
- [6] 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, April 2003.
- [7] John Boyland. Alias burying: Unique variables without destructive reads. *Software — Practice and Experience*, 31(6):533–553, 2001.
- [8] John Boyland, James Noble, and William 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.
- [9] John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1029–1052, November 2004.
- [10] Karl Crary, David Walker, and Greg 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, January 1999.
- [11] ECMA. C# language specification (3rd edition). Standard ECMA-334, ECMA, June 2005.

- [12] Philip 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.
- [13] Philip 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, October 2004.
- [14] Philip 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, October 2005.
- [15] Philip W. L. Fong. Discretionary capability confinement. In *Proceedings of the 11th European Symposium on Research in Computer Security (ESORICS'06)*, volume 4189 of *LNCS*, pages 127–144, Hamburg, Germany, September 2006.
- [16] Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, May 2003.
- [17] Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, November 1999.
- [18] Stephen N. Freund and John C. Mitchell. A type system for the Java bytecode language and verifier. *Journal of Automated Reasoning*, 30(3–4):271–321, September 2003.
- [19] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 3rd edition, 2005.

- [20] Christian Grothoff, Jens Palsberg, and Jan 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, October 2001.
- [21] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions on Programming Languages and Systems*, 28(1):175–205, January 2006.
- [22] Pieter H. Hartel and Luc Moreau. Formalizing the safety of Java, the Java Virtual Machine, and Java Card. *ACM Computing Surveys*, 33(4):517–558, December 2001.
- [23] Tomoyuki Higuchi and Atsushi Ohori. A static type system for JVM access control. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 227–237, Uppsala, Sweden, August 2003.
- [24] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.
- [25] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Springer, 2000.
- [26] Gerwin Klein and Tobias Nipkow. Verified lightweight bytecode verification. *Concurrency and Computation: Practice and Experience*, 13(13):1133–1151, November 2001.
- [27] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for runtime security policies. *International Journal of Information Security*, 4(1–2):2–16, February 2005.

- [28] Jay Ligatti, Lujo Bauer, and David 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, September 2005.
- [29] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999.
- [30] Hanbing Liu and J. Strother Moore. Executable JVM model for analytical reasoning: A study. *Science of Computer Programming*, 57(3):253–274, September 2005.
- [31] J. Strother Moore, Robert Krug, Hanbing Liu, and George Porter. Formal models of Java at the JVM level: A survey from the ACL2 perspective. In *ECOOP'01 Workshop on Formal Techniques for Java Programs*, Budapest, Hungary, June 2001.
- [32] J. Strother Moore and George Porter. The apprentice challenge. *ACM Transactions on Programming Languages and Systems*, 24(3):193–216, May 2002.
- [33] J. Strother Moore and George M. Porter. An executable formal Java Virtual Machine thread model. In *Proceedings of the First Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 91–104, Monterey, California, USA, April 2001.
- [34] Greg Morrisett, David Walker, Karl Cray, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.
- [35] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, January 1997.
- [36] Tobias Nipkow. Verified bytecode verifiers. In *Proceedings of the 4th International Conference on Foundations of Software Science and Computation (FOSSACS'01)*, Genova, Italy, April 2001.

- [37] Tobias Nipkow. Java bytecode verification. *Journal of Automated Reasoning*, 30(3–4), September 2003.
- [38] François Pottier, Christian Skalka, and Scott Smith. A systematic approach to static access control. *ACM Transactions on Programming Languages and Systems*, 27(2):344–382, March 2005.
- [39] Zhenyu Qian. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems*, 22(4):638–672, July 2000.
- [40] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, February 2000.
- [41] Christian Skalka and Scott Smith. Static use-based object confinement. *International Journal of Information Security*, 4(1–2):87–104, February 2005.
- [42] Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer, 2001.
- [43] Raymie Stata and Martin Abadi. A type system for Java bytecode subroutine. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, January 1999.
- [44] Jan Vitek and Boris Bokowski. Confined types in Java. *Software — Practice and Experience*, 31(6):507–532, 2001.
- [45] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects and virtual machines revisited. In *Proceedings of the 2002 International Symposium of Formal Methods Europe (FME’02)*, pages 89–105, Copenhagen, Denmark, July 2002.

- [46] Tian Zhao and John 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.
- [47] Tian Zhao, Jens Palsberg, and Jan 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, October 2003.
- [48] Tian Zhao, Jens Palsberg, and Jan Vitek. Type-based confinement. *Journal of Functional Programming*, 16(1):83–128, January 2006.

A Proof of Theorem 1

We confirm $\text{SafeState}(T)$ by a case analysis on the transition rule employed to obtain $S \rightarrow T$.

Case T-B-WIDEN: To show $\text{SafeFrame}(\Phi \cup \{r : B\} \mid \Pi)$, notice that $\text{SafeFrame}(\Phi \mid \Pi)$ ensures $C' <: C$ if C' is the class of r . As T-B-WIDEN guarantees $C <: B$, we obtain, by transitivity, $C' <: B$ as required.

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

Case T-B-CAST: To see $\text{SafeFrame}(\Phi \cup \{r : B\} \mid \Pi)$, observe that T-B-CAST explicitly requires the antecedents $\Pi \vdash r : C'$ and $C' <: B$.

Case T-B-GET: To see $\text{SafeFrame}(\Phi \cup \{q : C\} \mid \Pi)$, notice that T-B-GET guarantees $\Gamma \vdash p : B \rightsquigarrow q : C$, which, by $\text{SafeLinks}(\Gamma \mid \Pi)$, implies that $C' <: C$ when $\Pi \vdash q : C'$.

Case T-B-PUT: We show $\text{SafeLinks}(\Gamma \cup \{p : B \rightsquigarrow q : C\} \mid \Pi)$ in 2 steps. First, by $\text{SafeFrame}(\Phi \mid \Pi)$, $\Pi \vdash q : C'$ implies $C' <: C$ as required. Second, by $\text{SafeFrame}(\Phi \mid \Pi)$

again, $\Pi \vdash p : B'$ implies $B' <: B_0$. As T-B-PUT guarantees $B_0 < B$, the latter in turn implies $B' <: B$ as required.

Case T-B-INVOKE: We need to show $\text{SafeFrame}(\Phi' | \Pi)$ and $\text{SafeStack}(\sigma' | \Pi)$.

To show $\text{SafeFrame}(\Phi' | \Pi)$, where $\Phi' = \{r_0 : B', \bar{r} : \bar{C}\}$, notice the following two points. Firstly, T-B-INVOKE guarantees $\Phi \vdash \bar{r} : \bar{C}$, which, by $\text{SafeFrame}(\Phi | \Pi)$ implies that $C'_i <: C_i$ follows from $\Pi \vdash r_i : C'_i$. Secondly, T-B-INVOKE guarantees that $\Pi \vdash r_0 : B''$ and $B'' <: B'$ hold simultaneously.

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

Case T-B-RETURN: Firstly, $\text{SafeStack}(\sigma | \Pi)$ is guaranteed by $\text{SafeStack}(\text{push}(\Phi, A, C, \sigma) | \Pi)$.

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

B Proof of Theorem 3

We confirm $\text{ConfinedState}(T)$ by a case analysis on the transition rule employed to obtain $S \rightarrow_{\Sigma} T$.

Case T-G-WIDEN: We show $\text{ConfinedFrame}(\Phi \cup \{r : B^\beta\} | \Pi, A.m)$ in 2 steps. First, P-CT-

WIDEN guarantees that $B \triangleright A$. Second, by $\text{ConfinedFrame}(\Phi | \Pi, A.m)$, $\Phi \vdash r : C^\gamma$ and $\Pi \vdash r : C'$ jointly imply that either (i) $C' \blacktriangleright C$ or (ii) $(\gamma \neq \overline{\mathbf{this}} \wedge m = \mathbf{anon})$. If case (i) holds, then P-CT-WIDEN ensures that $C \blacktriangleright B \vee (\beta \neq \overline{\mathbf{this}} \wedge m = \mathbf{anon})$, and thus by (2) we deduce $(C' \blacktriangleright B \vee (\gamma \neq \overline{\mathbf{this}} \wedge m = \mathbf{anon}))$ as required. Otherwise, case (ii) holds. Because P-CT-WIDEN ensures that $\gamma \sqsubseteq: \beta$, we deduce $(\beta \neq \overline{\mathbf{this}} \wedge m = \mathbf{anon})$ as required.

Case T-G-NEW: $\text{ConfinedFrame}(\Phi \cup \{r : B\} \mid \Pi \cup \{r : B^\beta\}, A.m)$ holds⁶, because (i) P-CT-NEW mandates that $B \triangleright A$, and (ii) $B \blacktriangleright B$ by (1).

Case T-G-CAST: Similar to T-G-WIDEN.

Case T-G-GET: We show $\text{ConfinedFrame}(\Phi \cup \{q : C^\gamma\} \mid \Pi, A.m)$ in 2 steps. First, P-CT-GET ensures⁷ that $C \blacktriangleright B$ and $B \triangleright A$. Thus, by (5), we have $C \triangleright A$ as required. Second, $\text{ConfinedLinks}(\Gamma \mid \Pi)$ guarantees that $\Gamma \vdash p : B \rightsquigarrow q : C$ and $\Pi \vdash q : C'$ jointly imply $C' \blacktriangleright C$ as required.

Case T-G-PUT: We show $\text{ConfinedLinks}(\Gamma \cup \{p : B \rightsquigarrow q : C\} \mid \Pi)$ in 2 steps. First, $C \blacktriangleright B$ is guaranteed by P-CT-PUT. Second, because P-CT-PUT requires that $(\gamma = \overline{\mathbf{this}} \vee m \neq \mathbf{anon})$, $\text{ConfinedFrame}(\Phi \mid \Pi, A.m)$ therefore implies $C' \blacktriangleright C$ as required⁸.

Case T-G-INVOKE: We want to demonstrate that $\text{ConfinedFrame}(\Phi' \mid \Pi, B'.n')$ and $\text{ConfinedStack}(\sigma' \mid \Pi, B'.n')$.

To show $\text{ConfinedFrame}(\Phi' \mid \Pi, B'.n')$, where $\Phi' = \{r_0 : B'^{\beta_0}, \bar{r} : \overline{C'^{\beta}}\}$, we treat the labeled references $r_0 : B'^{\beta_0}$ and $\bar{r} : \overline{C'^{\beta}}$ separately. We first consider $\bar{r} : \overline{C'^{\beta}}$. First, because P-CT-INVOKE mandates $\beta_i = \overline{\mathbf{this}}$, we need $C'_i \blacktriangleright C_i$ whenever $\Pi \vdash r_i : C'_i$. This holds because $\text{ConfinedFrame}(\Phi \mid \Pi, A.m)$ implies $(C'_i \blacktriangleright C_i \vee (\gamma_i \neq \overline{\mathbf{this}} \wedge m = \mathbf{anon}))$ while P-CT-INVOKE requires $(\gamma_i = \overline{\mathbf{this}} \vee m \neq \mathbf{anon})$. Second, we need $C_i \triangleright B'$. Observe that P-CT-INVOKE 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

⁶The antecedent $\beta = \overline{\mathbf{this}}$ in P-CT-NEW is not needed for establishing the **Confinement Theorem**. It is introduced to make the type analysis more accurate.

⁷The antecedent $C \blacktriangleright B$ is redundant in P-CT-GET, because the condition is already implied by $\text{ConfinedLinks}(\Gamma \mid \Pi)$. It is introduced for symmetry.

⁸The antecedent $B \triangleright A$ in P-CT-PUT is not needed for establishing the **Confinement Theorem**. It is introduced for symmetry.

to show that $\Pi \vdash r_0 : C'_0$ entails $(C'_0 \blacktriangleright B' \vee (\beta_0 \neq \overline{\mathbf{this}} \wedge n' = \mathbf{anon}))$. There are 2 subcases: $n = \mathbf{anon}$ or $n \neq \mathbf{anon}$. If $n = \mathbf{anon}$, then P-CT-INVOKE guarantees $n' = \mathbf{anon}$ and $\beta_0 = \mathbf{this}$ as required. If $n \neq \mathbf{anon}$, then P-CT-INVOKE ensures that $(C_0 \blacktriangleright B \wedge (\gamma_0 = \overline{\mathbf{this}} \vee m \neq \mathbf{anon}))$. By $\mathit{ConfinedFrame}(\Phi \mid \Pi, A.m)$, 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 $\mathit{ConfinedStack}(\sigma' \mid \Pi, B'.n')$, where $\sigma' = \mathit{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma)$, notice that the preconditions of the theorem already guarantee $\mathit{ConfinedFrame}(\Phi \mid \Pi, A.m)$ and $\mathit{ConfinedStack}(\sigma \mid \Pi, A.m)$. As well, both $(\beta = \overline{\mathbf{this}} \vee n' \neq \mathbf{anon})$ and $\gamma = \overline{\mathbf{this}}$ are antecedents of P-CT-INVOKE. 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 P-CT-INVOKE.

Case T-G-RETURN: We show $\mathit{ConfinedFrame}(\Phi \cup \{r : C^\gamma\} \mid \Pi, A.m)$ in 2 steps. First, $\mathit{ConfinedStack}(\mathit{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma) \mid \Pi, B'.n')$ guarantees $C \triangleright A$ as required. Second, because $\mathit{ConfinedStack}(\mathit{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma) \mid \Pi, B'.n')$ requires $\gamma = \overline{\mathbf{this}}$, we need to show that $\Pi \vdash r : C'$ implies $C' \blacktriangleright C$. But then $\mathit{ConfinedStack}(\mathit{push}(\Phi, A.m, C^{\beta/\gamma}, \sigma) \mid \Pi, B'.n')$ also guarantees $(\beta = \overline{\mathbf{this}} \vee n' \neq \mathbf{anon})$, which, by $\mathit{ConfinedFrame}(\Phi' \mid \Pi, B'.n')$, entails $C' \blacktriangleright C$ as required.

C Proof of Theorem 5

We confirm $\mathit{ConfinedState}_{cap}(T)$ by a case analysis on the transition rule employed to obtain $S \rightarrow_\Sigma T$.

Case T-G-WIDEN: We show $\mathit{ConfinedFrame}_{cap}(\Phi \cup \{r : B^\beta\} \mid \Pi, A)$ in 2 steps. First, P-CAP-WIDEN guarantees that $B^\beta \triangleright : A \vee \beta = \mathbf{anon}$. Second, by $\mathit{ConfinedFrame}_{cap}(\Phi \mid \Pi, A)$, $\Pi \vdash r : C'$ implies $C'^* \blacktriangleright : C^\gamma$, which in turn implies $C'^* \blacktriangleright : B^\beta$ via (10) because P-CAP-WIDEN guarantees $C^\gamma \blacktriangleright : B^\beta$.

Case T-G-NEW: $ConfinedFrame_{cap}(\Pi \cup \{r : B\} \mid \Phi \cup \{r : B^*\}, A)$ holds, because (i) P-CAP-NEW mandates that $B^* \triangleright : A$, and (ii) $B^* \blacktriangleright : B^*$ by (9).

Case T-G-CAST: Similar to T-G-WIDEN.

Case T-G-GET: We show $ConfinedFrame_{cap}(\Phi \cup \{q : C^\gamma\} \mid \Pi, A)$ in 2 steps. First, P-CAP-GET ensures that $C^\gamma \blacktriangleright : B^*$ and $B^* \triangleright : A$. Thus, by (13), we have $C^\gamma \triangleright : A$ as required. Second, $ConfinedLinks_{cap}(\Gamma \mid \Pi)$ guarantees that $\Gamma \vdash p : B \xrightarrow{\gamma} q : C$ and $\Pi \vdash q : C'$ jointly imply $C'^* \blacktriangleright : C^\gamma$ as required.

Case T-G-PUT: We show $ConfinedLinks_{cap}(\Gamma \cup \{p : B \xrightarrow{\gamma} q : C\} \mid \Pi)$ in 2 steps. First, $C^\gamma \blacktriangleright : B^*$ is guaranteed by P-CAP-PUT. Second, by $ConfinedFrame_{cap}(\Phi \mid \Pi, A)$, $\Pi \vdash r : C'$ implies $C'^* \blacktriangleright : C^\gamma$ as required.

Case T-G-INVOKE: We show that $ConfinedFrame_{cap}(\Phi' \mid \Pi, B')$ and $ConfinedStack_{cap}(\sigma' \mid \Pi)$.

To show $ConfinedFrame_{cap}(\Phi' \mid \Pi, B')$, where $\Phi' = \{r_0 : B'^{\beta'_0}, \bar{r} : \overline{C'^{\beta}}\}$, we treat the labeled references $r_0 : B'^{\beta'_0}$ and $\bar{r} : \overline{C'^{\beta}}$ separately. We first consider $\bar{r} : \overline{C'^{\beta}}$. 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, observe that P-CAP-INVOKE guarantees $C_i'^{\gamma_i} \blacktriangleright : B^*$, which, by (6), (11), (10) and (12), implies $C_i'^{\gamma_i} \triangleright : B'$ as required.

We now consider the labeled reference $r_0 : B'^{\beta'_0}$. First, P-CAP-INVOKE requires $C_0'^{\gamma_0} \blacktriangleright : B^{\beta_0}$ and $B^{\beta_0} \blacktriangleright : B'^{\beta'_0}$, which, by (10), implies $C_0'^{\gamma_0} \blacktriangleright : B'^{\beta'_0}$. By $ConfinedFrame_{cap}(\Phi \mid \Pi, A)$ and (10), this in turn implies $C_0'^* \blacktriangleright : B'^{\beta'_0}$ when $\Pi \vdash r_0 : C_0'$. Second, the requirement $(B'^{\beta'_0} \triangleright : B' \vee \beta'_0 = \mathbf{anon})$ is satisfied trivially by (15).

To show $ConfinedStack_{cap}(\sigma' \mid \Pi)$, where $\sigma' = push(\Phi, A, C^{\gamma/\gamma}, \sigma)$, 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 (13), follows from $C^\gamma \blacktriangleright : B^*$ and $B^* \triangleright : A$, both guaranteed by P-CAP-INVOKE.

Case T-G-RETURN: We show $ConfinedFrame_{cap}(\Phi \cup \{r : C^\gamma\} \mid \Pi, A)$ in 2 steps. First, $ConfinedFrame_{cap}(push(\Phi, A.\alpha, C^{\gamma/\gamma}, \sigma) \mid \Pi, B')$ 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.

D Proof of Theorem 6

The proof parallels that of Theorem 1 closely. Only the following cases are interesting.

Case T-E-NEWARRAY: Because r is a fresh object reference from \mathcal{O} , $SafeHeap(\Pi \cup \{r : R[]\})$ holds trivially. As well, $SafeFrame(\Phi \cup \{r : R[]\} \mid \Pi \cup \{r : R[]^\gamma\})$ holds because $R[] <: R[]$.

Case T-E-GETARRAY: Notice that T-E-GETARRAY guarantees $\Gamma \vdash p : [] \rightsquigarrow q$, which, by $SafeLinks(\Gamma \mid \Pi)$, implies that $Q <: P$ whenever $\Pi \vdash p : P[]$ and $\Pi \vdash q : Q$. By $SafeFrame(\Phi \mid \Pi)$, $P[] <: R[]$, and thus $P <: R$. By transitivity of subtyping, we deduce $Q <: R$, and thus obtain $SafeFrame(\Phi \cup \{q : R^\beta\} \mid \Pi)$ as required.

Case T-E-PUTARRAY: We have $SafeLinks(\Gamma \cup \{p : [] \rightsquigarrow q\} \mid \Pi)$ as required, because T-E-PUTARRAY guarantees that $Q <: P$ whenever $\Pi \vdash p : P[]$ and $\Pi \vdash q : Q$.

Case T-E-THROW: By straightforward induction, one can show that,

$$\begin{aligned} & (SafeFrame(\Phi \mid \Pi) \wedge SafeStack(\sigma \mid \Pi) \wedge unwind(\Phi, A.m, \sigma ; \Phi', A'.m', \sigma')) \\ & \Rightarrow (SafeFrame(\Phi' \mid \Pi) \wedge SafeStack(\sigma' \mid \Pi)) . \end{aligned}$$

What remains to be shown is $SafeFrame(\Phi' \cup \{r : B^\beta\} \mid \Pi)$. But then that is straightforward because T-E-THROW guarantees that $C' <: B$ whenever $\Pi \vdash r : C'$.