A caller-side inline reference monitor for object-oriented intermediate language: Extended version

Dries Vanoverberghe
Frank Piessens

Report CW 512, March 2008

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

A caller-side inline reference monitor for object-oriented intermediate language: Extended version

Dries Vanoverberghe
Frank Piessens

Report CW 512, March 2008

Department of Computer Science, K.U.Leuven

Abstract

Runtime security policy enforcement systems are crucial to limit the risks associated with running untrustworthy (malicious or buggy) code. The inlined reference monitor approach to policy enforcement, pioneered by Erlingsson and Schneider, implements runtime enforcement through program rewriting: security checks are inserted inside untrusted programs.

Ensuring complete mediation – the guarantee that every security-relevant event is actually intercepted by the monitor – is non-trivial when the program rewriter operates on an object-oriented intermediate language with state-of-the-art features such as virtual methods and delegates.

This paper proposes a caller-side rewriting algorithm for MSIL – the bytecode of the .NET virtual machine – where security checks are inserted around calls to security-relevant methods. We prove that this algorithm achieves sound and complete mediation and transparency for a simplified model of MSIL, and we report on our experiences with the implementation of the algorithm for full MSIL.

A Caller-Side Inline Reference Monitor for an Object-Oriented Intermediate Language: Extended version

Dries Vanoverberghe* and Frank Piessens

Dries.Vanoverberghe, Frank.Piessens@cs.kuleuven.be

Abstract. Runtime security policy enforcement systems are crucial to limit the risks associated with running untrustworthy (malicious or buggy) code. The inlined reference monitor approach to policy enforcement, pioneered by Erlingsson and Schneider, implements runtime enforcement through program rewriting: security checks are inserted inside untrusted programs.
Ensuring complete mediation – the guarantee that every security-relevant event is actually intercepted by the monitor – is non-trivial when the program rewriter operates on an object-oriented intermediate language with state-of-the-art features such as virtual methods and delegates.
This paper proposes a caller-side rewriting algorithm for MSIL – the bytecode of the .NET virtual machine – where security checks are inserted around calls to security-relevant methods. We prove that this algorithm achieves sound and complete mediation and transparency for a simplified model of MSIL, and we report on our experiences with the implementation of the algorithm for full MSIL.

Key words: security policy enforcement, inline reference monitor

1 Introduction

In today's networked world, code mobility is ubiquitous. Applications can be downloaded over the internet, or received as an attachment of emails. This support for applications from potentially untrustworthy sources comes with a serious risk: malicious or buggy applications can lead to denial of service, financial damage, leaking of confidential information and so forth.

One important class of countermeasures addresses this risk by monitoring the application at run time, and aborting it if it violates a predefined security policy. The events monitored are called *security-relevant events*, and they typically are operating system calls, or platform API method calls.

This paper is about such policy enforcement systems that are rich enough to enforce policies specified by means of a security automaton [1], an automaton that defines what sequences of security-relevant events are acceptable. Several

* Dries Vanoverberghe is a research assistant of the Fund for Scientific Research - Flanders (FWO)

such systems have been designed and prototyped [2–4], and security automata have been shown to express exactly the policies enforceable by run-time monitoring [1]. The code access security architectures present in Java and .NET are an instance of such systems [2].

In standard policy enforcement systems, monitoring applications is integrated into the execution system or the trusted libraries. This makes it fairly easy to show complete mediation [5], the property that the monitor sees every occurrence of a security-relevant event. This paper zooms in on the inlined reference monitor (IRM) [6] approach, that rewrites untrusted applications and embeds the monitor directly into the application itself. Ensuring complete mediation is non-trivial when the program rewriter operates on an object-oriented intermediate language with state-of-the-art features such as virtual methods and delegates.

This paper proposes a caller-side rewriting algorithm for MSIL – the bytecode of the .NET virtual machine – where security checks are inserted around calls to security-relevant methods. We prove that this algorithm achieves sound and complete mediation and transparency (the property that the behavior of the rewritten program does not change if it is compliant with the enforced policy) for a simplified model of MSIL, and we report on our experiences with the implementation of the algorithm for full MSIL.

Section 2 elaborates on the problem statement. In Section 3, we introduce a simplified model of MSIL and the .NET Common Language Runtime (CLR) – the .NET virtual machine, and we propose a program rewriter that achieves complete mediation. Sections 4 and 5 explain how this system can be extended to support virtual method dispatch and delegates. In Section 6, we describe the implementation of our program rewriter for the full .NET CLR. Finally, we cover related work and conclude in Sections 7 and 8.

2 Problem statement

The design space for IRM systems is rich, and different designs have different advantages and disadvantages. In particular, the problem of proving complete mediation is harder for some designs than for others. We discuss some of the design parameters, and motivate our design choices.

A first important design choice is the security-relevant events. Events can range from individual bytecode or machine instructions [7] over operating system calls [8] to Java method calls [6]. The trade-offs of these choices are discussed in [7], and broadly speaking the conclusion is that fine-grained monitoring allows expressive policies, but coarser-grained monitoring makes policies simpler to understand and write, and it is sufficiently expressive for practical purposes. Our system monitors method invocations.[1]

Furthermore, the abstraction level of the method calls monitored is important. Figure 1(a) shows a simplified architecture for a managed .NET application. The application calls methods of the API of the platform library, for instance a

[1] A method invocation is when the execution enters a new method. Method calls are first dispatched to find the actual target method before they are invoked.

method to send data over a network socket. The implementation of this method calls a lower-level native method, implemented in the runtime system. The native methods in turn performs system calls.

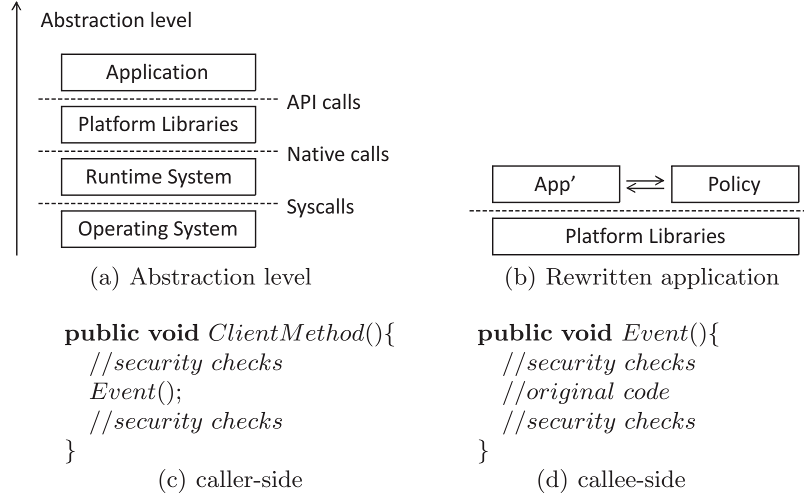

Fig. 1. Design decisions for a policy enforcement systems

A second important design choice is whether to monitor the high-level methods that the application calls directly, or the low-level methods that are the most primitive abstraction of the system resources. Current research usually expresses security policies in terms of low-level methods [6, 3]. This makes it easy to write policies that limit access to system resources, for instance limit the amount of network traffic, or file access. The higher level methods need not be monitored directly, because their implementations will call the lower level methods.

Unfortunately, as current system libraries can be complex, it can be hard for application developers to determine the security-relevant behavior of their application, since they make use of high-level methods. Furthermore, policy writers often want to selectively allow certain high-level methods, even if it executes a low-level method that is forbidden (e.g. using a logging method, even if access to the file system is not allowed).

In this paper, we use the high-level platform API methods to write policies. Security-relevant events are invocations of methods defined in the trusted system libraries from inside the untrusted application. This approach is similar to monitoring unmanaged applications at the boundary between kernelspace and userspace. In other words, we monitor the control flow transitions between Application and Platform Libraries in Figure 1(a).

A third design parameter is where to insert security checks. With *callee-side* program rewriting, the rewriter inserts checks inside the body of security-relevant methods (See Figure 1(d)). In a safe execution mechanism, it is fairly easy to

show that untrusted applications can not circumvent security code. On the other hand, selectively allowing calls based on the call site is much harder. Since we only want to monitor calls originating from the (untrusted) application, callee-side program rewriting is troublesome. Also, the rewriter needs to modify the trusted system libraries and this can be impossible, for instance when they are in ROM on a mobile device, or when a third party performs the inlining of the application as a service (as proposed in the S3MS project [9]).

In this paper, we use *caller-side* program rewriting, where the rewriter inserts checks at the call site (See Figure 1(c)).

Finally, a fourth design choice is what to inline: the policy enforcement code itself, or just a call to an external component that implements the policy. Both approaches are used in other systems, and the differences between the two approaches are minor. For convenience reasons – it is easier to formulate the complete mediation property we want to prove – we inline calls to a separate Policy Decision Point (PDP).

Figure 1(b) shows how the untrusted application is transformed into a new application and this new application invokes a method in the PDP before and after each security-relevant event.

Illustration of the issues

IRM's are a powerful policy enforcement mechanism, but showing that a monitored application cannot subvert the security checks can be complex, in particular for IRM's based on caller-side rewriting. The security checks are inlined statically, and executed before the method call has been dispatched. In modern execution systems, this raises a number of important challenges:

- When using virtual methods or interfaces, the target method can usually not be determined statically. At runtime, the target is determined using the runtime type of the target object. An attacker could try to circumvent the security policy by casting an object to its base type and executing a security-relevant method using virtual dispatch. Alternatively, an attacker might inherit from the trusted system libraries or override the behavior of a security-relevant method.
- With delegates, the situation is even worse. A delegate points to a set of methods, and methods can be added to or removed from this set at runtime. Invoking the delegate invokes all methods in the set. An attacker could try to hide a call to a security-relevant method by adding it to a delegate and calling this delegate.

In short, achieving complete mediation with a caller-side program rewriter is much more challenging than using callee-side program rewriter. This is the key problem addressed in this paper.

3 Base system

In this section, we prove sound and complete mediation and transparency for a simplified model of the .NET CLR [10].

3.1 Execution System

Our formal model of the .NET CLR is based on Fruja's formalization [11]. Our model in this section supports assemblies (the .NET components, similar to Java's jar files), classes with (possibly static) non-virtual methods, and exceptions. Later sections discuss the extension to virtual methods. We do not model interfaces, value types or multithreading. We briefly describe the formal model, but since it is relatively straightforward, the reader is referred to [11] for details.

The execution system is a virtual machine that loads two assemblies: the untrusted application (U), and the trusted platform libraries (T). We assume that U has a method *main* without input arguments to start the execution. Each assembly contains a set of classes. A *type* is a pair consisting of an assembly and a class name defined in that assembly.

A *method reference* is a pair of a type and a method name (written as $Type :: MethodName$). The function *retType* maps a method reference to the type of the return value (or the special return type **void**). *locTypes* maps a method reference to a list of types for the local variables of that method. The function *argTypes* defines a list of types for the arguments of a method reference. If the method is an instance method, the first argument is the implicit this argument. Finally, the function *code* maps a method reference into a list of instructions, the bytecode representation of the body of the method. Table 1 summarizes the instructions that we will need in this paper. The virtual machine supports more instructions, but their semantics is straightforward.

An *exception handler* is a five-tuple defining the position of the beginning (inclusive) and end (exclusive) of the try block, the type of exception that is caught, and the position of the beginning and end of the catch block of the handler. The function *excHa* maps a method reference into a list of exception handlers.

The *execution state* of the virtual machine consists of a heap, a list of activation records and an exception record. The *heap* is modeled as a finite map from addresses to values. *heapLookup(heap, address)* looks up the value at a given address in the heap. Furthermore, *heapUpdate(heap, address, value)* returns a new heap that results from replacing the content of heap at the given address by the given value. An *activation record* is a five-tuple consisting of the current program counter, a list of addresses of the local variables, a list of addresses for the arguments, a stack of values (the evaluation stack) and the method reference of the method that is being executed. The *exception record* is a pair containing the currently active exception, and the next exception handler that will be tested to handle an exception. In our formalization, the active exception is always a value of the type *(T,Exception)* or *(T, SecurityException)*(where T is the trusted assembly).

instruction	explanation
ldc.x	load a constant on the evaluation stack
ldloc i	load the value of the local variable at index i on the stack
stloc i	store the top value of the stack in the local variable at index i and pop it from the stack
brtrue k	branch to the instruction at index k if the top value of the stack is true and pop it from the stack
br k	unconditional branch to the instruction at index k
call *mref*	call the method mref using the arguments on the stack (and remove them from the stack) and put the return value on the stack (if return type not **void**)
ret	return from a method (and use the top value of the stack as return value if return type not **void**)
throw	throw the top value of the stack as an exception
newobj *mref*	create a new object using the constructor mref and put a reference to the new object on the stack

Table 1. Instructions

Operational semantics

In a given state, *instr* denotes the current instruction that will be considered by the execution system. It is shorthand for $code(mref)[pc]$ with $mref$ the method that is currently being executed (the top frame of the activation records), and *pc* the current program counter within this method.

An execution state is in *normal execution mode* if the active exception of the exception record is null. Table 1 informally explains the instructions that are relevant for our algorithm, and Appendix A defines the operational semantics for the instructions formally.

An execution state is in *exception handling mode* if it is not in normal execution mode. Appendix A also defines the operational semantics for exception handling. Exception handling mode is entered with the **throw** instruction, and it starts by looking for suitable exception handlers in the current method. An exception handler is suitable if the current program counter is inside the scope of the try block, and if the type of the exception is a subtype of the type of the exception handler. If a suitable handler is found, the evaluation stack is cleared and the active exception is pushed onto the stack. Execution continues in normal mode with the first instruction of the catch block of the exception handler.

If all exception handlers of the current method have been considered, the exception is propagated to the caller of the current method, and the execution mechanism continues searching at the first handler of the caller.

3.2 The inlining algorithm

Recall from Figure 1(b) that an inlining algorithm takes an untrusted assembly *App* as input, and it outputs a new assembly *App*′ that notifies the PDP of all security-relevant events. The security-relevant events are the entering into and exiting (both normally or exceptionally) from *security-relevant methods*. The

security-relevant methods are a subset (designated by the policy writer) of the methods defined in the trusted assembly. Notification of the PDP is done by calling so-called *policy methods* in the PDP.

Definition 1 (Policy methods). *The functions $before$, $after$ and $excep$ map a security-relevant method $mref$ onto the corresponding policy methods $before(mref)$, $after(mref)$ and $excep(mref)$ that are called respectively before entering $mref$, after successful return, and after exceptional return. $before(mref)$ has the same argument types as $mref$, $after(mref)$ has an additional parameter for the return value, and $excep(mref)$ has an additional parameter for the exception.*

In our simplified model, we require policy methods to be part of the trusted assembly. They throw a SecurityException when the policy is violated, and return **void** otherwise. Policy methods are not allowed to call back into the untrusted assembly. The untrusted application (before inlining) should not contain any calls to policy methods.

Our inlining algorithm is defined in Figure 2. The algorithm goes through the instruction list of all methods in the untrusted assembly, looking for **call** instructions to a security-relevant method. Each such call instruction is replaced by a block of code generated using the *generateCode* function, defined below. We say that such call instruction has been *processed* by the inliner. For simplification purposes, we do not intercept constructor calls, but their treatment is identical to a call to a static method with the new object as return value.

$Body = (List(Instr), List(Type), List(ExceptionHandler))$

$inlinebody : Map(Body, Body)$
$inlinebody((body, localvars, excha)) = inlinebodyhelp([], body, 0, localvars, excha)$

$inlinebodyhelp(left, [], _, localvars, excha) =$
  return $(left, localvars, excha)$
$inlinebodyhelp(left, right, i, localvars, excha) =$
  $right == instr, right'$
  if $(instr == call\ mref\ and\ mref\ is\ a\ SRM)$ {
    $(newCode, localvars', newExcha) = generateCode(instr, localvars, excha, i)$
    $n = \#newCode$
    $left' = patchBranchTargets(left, n, i)$
    $right'' = patchBranchTargets(right', n, i)$
    $left'' = left', newCode$
    $excha' = newExcha, patchExchaPcs(excha, n, i)$
    return $inlinebodyhelp(left'', right'', i + n, localvars', excha')$
  } **else** {
    $left' = left, instr$
    return $inlinebodyhelp(left', right', i + 1, localvars, excha)$
  }

Fig. 2. Program rewriter algorithm

Definition 2 (generateCode). *Given an instruction instr of the form* **call** $mref$*, a list of types* $localvars$*, a list of exception handlers excha and the index of the instruction* i*, the function* $generateCode(instr, localvars, excha, i)$ *returns a block of code defined by the template in Figure 3, a modified list of types for local variables and a modified list of exception handlers.*

```
      stloc m+n
      ...                          1. Store arguments
      stloc m
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      ldloc m
      ...                          2. Reload arguments
      ldloc m+n
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      call before(mref)            3. Call before method
    try{
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      ldloc m
      ...                          4. Reload arguments
      ldloc m+n
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      call mref                    5. Call original method
      stloc m+n+1                  6. Store return value
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      ldloc m
      ...                          7. Reload arguments and return value
      ldloc m+n+1
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      call after(mref)             8. Call after method
      ldloc m+n+1                  9. Reload return value
      br end                       10. goto end
    } catch (SecurityException) {
      throw                        11. Rethrow exception
    } catch (Exception) {
      stloc m+n+2                  12. Store exception
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      ldloc m
      ...                          13. Reload arguments and exception
      ldloc m+n
      ldloc m+n+2
- - - - - - - - - - - - - - - - - - - - - - - - - - - - -
      call exception(mref)         14. Call exception method
      ldloc m+n+2                  15. Reload exception
      throw                        16. Rethrow exception
  end: }
```

Fig. 3. Generated code fragment where n is the number of arguments of the target method, and m is the number of pre-existing local variables in the method that is being rewritten.

The insertion of the code fragments generated by *generateCode* changes the locations of the original instructions in the method being rewritten. Since these locations are used in branch instructions and in exception handlers, we need to patch these. So the inlining algorithm patches branch targets and exception

handlers using the function $patchLocation(n, i) = \lambda loc.\textbf{if}\ (loc > i)\ \textbf{then}\ loc + n - 1\ \textbf{else}\ loc$, where i is the location of the call being processed, and n is the size of the generated code block being inserted.

We discuss in more detail the block of code generated by *generateCode* (Figure 3). First, we generate *stloc* instructions to store the arguments that are on the evaluation stack into new local variables. Because these variables are new, and this is the only place in the generated code where values are stored into these variables, their value does not change after this point. This ensures that each time the variables are loaded on the stack, the relevant values on the top of the stack are the same values as the values initially on the stack.

Next, we use *ldloc* instructions to load the arguments on the stack again, and a *call* instruction to invoke the before method of the policy.

The remaining part of the code is a try catch structure. In the try scope, we first generate instructions to load the arguments on the stack and call the original security-relevant method. Then we store the return value in a local variable (if the return type is not void). For the after method of the policy, we generate instructions to load the arguments and the saved return value on the stack and to call the after method. Finally, we load the return value on the stack again, and we branch out of the try block.

The type of the first exception handler is *SecurityException*. Because we assume that security-relevant methods do not throw security exceptions, this exception comes from the after method. We generate code to simply rethrow the original exception.

The second exception handler catches any type of exceptions. Because we assume that the policy methods are total, this exception is raised inside the security-relevant method and the exceptional method of the policy must be executed. We generate instructions to store the exception into a local variable, reload the arguments, reload the exception, call the exceptional method of the policy and finally reload and rethrow the exception.

3.3 Properties of the inlining algorithm

The first property we consider is *complete mediation* [5]: every security-relevant event is seen by the monitor.

We say that a method invocation, return or exceptional return is *observable* when it crosses the boundary between the untrusted assembly and the trusted assembly. The function $observable(mref, mref')$ returns true if one of its arguments is defined in the trusted assembly, and the other one in the untrusted assembly.

An abstract trace of a program is a (possibly infinite) list of observable method invocations, returns and exceptional returns.

Definition 3 (Abstract trace). *The abstract trace of a program is the list of observable method invocations, returns and exceptional returns that occur when executing the main method of the program.*

Figure 4 shows how to compute the abstract trace, based on the operational semantics.

$$\frac{\begin{array}{c}(H,S,E) \rightarrow (H',S',E') \\ \#S' = \#S+1 \qquad S' = _,(_,_,argAdr,_,mref) \qquad S = _,(pc,_,_,_,oldmref) \\ code(oldmref)[pc]! = newobj_{_} \qquad observable(mref,oldmref) \\ argTypes(mref) == \overrightarrow{A}_{1..n} \qquad \overrightarrow{v} = heapLookup(H',argAdr)\end{array}}{trace(H,S,E) = Enter(mref,\overrightarrow{v}), trace(H',S',E')}$$

$$\frac{\begin{array}{c}(H,S,E) \rightarrow (H',S',E') \qquad \#S' = \#S-1 \qquad S = _,(pc,_,_,evalStack,mref) \\ S' = _,(_,_,_,_,oldmref) \qquad observable(mref,oldmref) \qquad E = (null,_) \\ code(mref)[pc] = ret \qquad retType(mref)! = void \qquad evalStack = evalStack',v\end{array}}{trace(H,S,E) = Return(v), trace(H',S',E')}$$

$$\frac{\begin{array}{c}(H,S,E) \rightarrow (H',S',E') \qquad \#S' = \#S-1 \\ S = _,(pc,_,_,_,mref) \qquad S' = _,(_,_,_,_,oldmref) \qquad observable(mref,oldmref) \\ E = (null,_) \qquad code(mref)[pc] = ret \qquad retType(mref) == void\end{array}}{trace(H,S,E) = Return, trace(H',S',E')}$$

$$\frac{\begin{array}{c}(H,S,E) \rightarrow (H',S',E') \qquad \#S' = \#S-1 \qquad S = _,(_,_,_,_,mref) \\ S' = _,(_,_,_,_,oldmref) \qquad observable(mref,oldmref) \qquad E = (v,_) \qquad v! = null\end{array}}{trace(H,S,E) = Exception(v), trace(H',S',E')}$$

$$\frac{(H,S,E) \rightarrow (H',S',E') \qquad \#S' = \#S}{trace(H,S,E) = trace(H',S',E')}$$

Fig. 4. Computation of the trace of a program

For simplification purposes, we assume that security-relevant methods do not call back into untrusted applications and they terminate. Furthermore, we assume that security-relevant methods always return a value to keep the properties simple.

Definition 4 (Complete mediation). *An abstract execution trace of an untrusted assembly program satisfies complete mediation if and only if for each index i in trace such that $trace[i] = Enter(mref, vals)$ and $mref$ is security-relevant:*

- *trace[i-2]=Enter(before(mref),vals) and trace[i-1]=Return*
- *trace[i+1]=Return(val) implies trace[i+2]=Enter(after(mref),vals . val)*
- *trace[i+1]=Exception(val) implies trace[i+2]=Enter(excep(mref),vals . val)*

We prove that our inliner always produces programs with completely mediated traces. First we need some technical lemmas.

Lemma 1 (No jumps into inlined code). *For every untrusted assembly P, let $P' = inline(P)$. For all possible runs of P', control flow can enter a code block inserted during inlining only through the first instruction of the block.*

Proof. The patching of the branch targets makes sure that there are no jumps into the generated code, thus control flow must enter through the first instruction.

Lemma 2 (Complete replacement). *Any call instruction that invokes a security-relevant method at run time, has been processed by the inliner.*

The proof of this lemma is trivial in this execution system. However, this is the key lemma that breaks when adding delegates or virtual method dispatch. Later sections discuss how to maintain this lemma in the presence of delegates and virtual calls, and this will require improvements to the inliner.

Lemma 3 (Abstract trace of generated code). *The code blocks output by $generateCode$ can only generate the following traces:*

1. *Enter(before(mref), vals), Exception(exc),... with exc a SecurityException.*
2. *Enter(before(mref), vals), Return, Enter(mref, vals), Exception(exc), Enter(excep(mref), vals . exc), Exception(exc2), ... with exc not a SecurityException and exc2 a SecurityException.*
3. *Enter(before(mref), vals), Return, Enter(mref, vals), Exception(exc), Enter(excep(mref), vals . exc), Return, ... with exc not a SecurityException.*
4. *Enter(before(mref), vals), Return, Enter(mref, vals), Return(val), Enter(after(mref), vals . val), Exception(exc), ... with exc a SecurityException.*
5. *Enter(before(mref), vals), Return, Enter(mref, vals), Return(val), Enter(after(mref), vals . val), Return, ...*

Proof. Taking into account the restrictions that (1) policy methods do not call back into the untrusted assembly, and (2) policy methods can only throw SecurityExceptions or return void, the proof is straightforward.

Theorem 1 (Complete mediation of the algorithm). *For every untrusted assembly P, let $P' = inline(P)$. The abstract trace $trace'$ of P' satisfies complete mediation.*

Proof. For each $Enter(mref, vals)$ in $trace'$ where $mref$ is security-relevant :

1. Using the computation rules for traces (Figure 4), the Enter can only be caused by a call instruction.
2. By Lemma 2, each call instruction that can potentially result in invoking a security-relevant method has been replaced by a block of generated code.
3. Using Lemma 1, the only way to start the execution of the generated block of code is by starting at the first instruction.
4. According to Lemma 3, the generated code can only lead to five possible traces. Because we have an $Enter(mref, vals)$ in the trace, trace 1 is not possible. In all other cases complete mediation holds by Definition 4.

The second property we consider is sound mediation.

Definition 5 (Sound mediation). *An abstract execution trace of an untrusted assembly program satisfies sound mediation if and only if for each index i in trace:*

- *trace[i]=Enter(before(mref),vals) and trace[i+1]=Return implies trace[i+2]=Enter(mref, vals)*
- *trace[i]=Enter(after(mref),vals . val) implies trace[i-2]=Enter(mref, vals) and trace[i-1]=Return(val)*
- *trace[i]=Enter(excep(mref),vals . val) implies trace[i-2]=Enter(mref, vals) and trace[i-1]=Exception(val)*

Theorem 2 (Sound mediation of the algorithm). *For every untrusted assembly P, let $P' = inline(P)$. The execution $trace'$ of P' satisfies sound mediation.*

Proof. Assuming that P did not contain any calls to the policy before rewriting, all calls to the policy in P' come from inside the generated code block. Using the generated traces (Lemma 3), completing the proof is trivial.

The third property we consider is *transparency*: a policy enforcement system is transparent if it has no observable effect on programs that satisfy the policy. We formalize this by considering the effect when inlining calls to policy methods that do nothing, they return immediately. We call such methods *passive*.

Definition 6 (Equality modulo policy calls). *Two abstract execution traces trace and $trace'$ are equal modulo policy calls if and only if they are equal after removing all calls to and returns from policy methods.*

Definition 7 (Transparency). *Let I be an inlining algorithm. Let P be a program, and let P' be the inlining of P with I using only passive policy methods. I is* transparent *if, for all such P, the trace of P' is equal modulo policy calls to the trace of P.*

Clearly, if policy methods are non-passive, a program may behave differently after inlining. In general, the desirable situation is that policy methods are indistinguishable from passive policy methods *until the untrusted program violates the policy.*

Theorem 3 (Transparency of the algorithms). *The inlining algorithm in Figure 2 is transparent.*

The algorithm defined in Section 3.2 transforms an existing program P in a new program P' that has a strong static relation with the original program.

First, we define a function $instrMap$ that maps the instructions of the original program P to the corresponding instructions of P'. If the instruction is a call to a security-relevant method in P, then the function maps to the first instruction of the generated block of code in P'.

Definition 8 (Mapping of instructions). *For each method reference* $mref$*, let* $instrMap(mref, i)$ *be the position in* $mref$ *in* P' *of the instruction that was at position* i *in* $mref$ *in* P*.*

Lemma 4. *Let* $instr$ *be an instruction at index* i *in* $mref$ *in* P *and* $instr'$ *be the instruction at index* $instrMap(mref, i)$ *in* $mref$ *in* P'*. Then* $instr = instr'$ *except*

- *If* $instr = call\ mref$ *and mref is security-relevant, then* $instrMap(mref, k)$ *is the index of the first instruction of the generated block of code.*
- *If* $instr = br\ k$ *then* $instr' = br\ instrMap(mref, k)$*.*
- *If* $instr = brtrue\ k$ *then* $instr' = brtrue\ instrMap(mref, k)$*.*

Next, we define a second function $callMap$ that maps the position of all call instructions in P to the corresponding position in P'.

Definition 9 (Mapping of call instructions). *For each method reference* $mref$*, let* $callMap(mref, i)$ *be the position in* $mref$ *in* P' *of the call instruction that was at position* i *in* $mref$ *in* P*.*

For all security-relevant calls, this function maps to the original call instruction in the middle of the block of generated code. For the rest, it maps to the corresponding instruction according to $instrMap$.

The algorithm also induces a strong relation between the exception handlers of P and P' which we call $excMap$.

Definition 10 (Mapping of exception handlers). *For each method reference* $mref$*,* $excMap(mref, i)$ *is the position in* $mref$ *in* P' *of the exception handler that was at position* i *in* $mref$ *in* P*.*

Lemma 5. *Let* $(tb, te, type, cb, ce)$ *be the exception handler at index* i *in* $mref$ *in* P *and* $(tb', te', type', cb', ce')$ *be the exception handler at index* $excMap(mref, i)$ *in* $mref$ *in* P'*. Then*

- $type = type'$
- $(tb, tb'), (te, te'), (cb, cb'), (ce, ce') \in instrMap(mref, i)$*.*

Finally, for each local variable defined in the body of each method reference of P, P' has a local variable with the same type at the same index in the body of that method reference, but the body of that method reference in P' can have additional local variables.

This strong static relation between P and P' create a strong equivalence relation between the states of P and P':

Definition 11 (Structural equivalence modulo policy enforcement). *An execution state* (H, S, E) *is structurally equivalent modulo policy enforcement with an execution state* (H', S', E') *if and only if (H,S,E) is equal to (H',S',E') except for*

1. *Alpha renaming of the addresses in the heap.*

2. *Each activation record S'_i may have additional variables.*
3. *The program counters pc_i and pc'_i of all activation records S_i and S'_i except the top records satisfy $pc'_i = callMap(mref, pc_i)$.*
4. *During normal execution mode, the program counters pc_n and pc'_n of the top activation record S_n and S'_n satisfy $pc'_n = instrMap(mref, pc_n)$.*
5. *During exception handling mode, the next exception handler indexes i of E and i' of E' satisfy $i' = excMap(mref, i)$ and the program counters pc_n and pc'_n of the top activation record S_n and S'_n are in the same relative position with respect to all remaining exception handlers.*

Lemma 6. *For every untrusted assembly P, let $P' = inline(P)$. For every execution state s_k reachable by P in k steps, there exists an execution state s'_l reachable by P' in $l \geq k$ steps, such that s_k and s'_l are structurally equivalent modulo policy enforcement and the abstract traces of both runs are equal modulo policy enforcement.*

Proof. By induction on k:

1. Initial step: When $k = 0$, choose $l = 0$ and the property trivially holds.
2. Induction step: Let $k' = k - 1$. Using the induction hypothesis, we know that there exists an l' such that $s_{k'}$ and $s'_{l'}$ are structurally equivalent modulo policy enforcement and the abstract traces of both runs are equal modulo policy enforcement.
   We case split on s'_k:
   (a) If the execution is in normal execution mode, and the instruction is a call instruction to a security-relevant method and the current method is inside P, then $s'_{l'}$ is also in normal execution mode and the current instruction is the first instruction of the generated code fragment (Figure 3).
   In step 1. of the generated code inside P', all arguments are stored in local variables, but the variables that also exist in P in the same method are not modified. Step 2. reloads all arguments on the stack again, and step 3. calls the before method of the policy. As the policy methods are passive, the execution returns to step 4. where the arguments are reloaded. Finally, the call to the security relevant method is executed and we enter the trusted system libraries.
   Let α be the amount of evaluation steps to reach the state after entering the security-relevant method, and let $l = l' + \alpha$, then s_l is structurally equivalent modulo policy enforcement with s_k. Since calls to policy methods are ignored, the abstract traces are equivalent modulo policy enforcement.
   (b) If the execution is in normal execution mode, and the instruction is a return instruction from a security-relevant method and the previous method is inside P, then in $s'_{l'+1}$ the execution is in normal mode and the instruction is the first instruction in the generated code fragment after the security-relevant call.

In step 6. of the generated code inside P', the return value is stored in a local variable, but again this does not modify the pre-existing variables in P. Step 7. reloads the arguments and the return value, and step 8. calls the after method of the policy. As the policy methods are passive, the execution returns to step 9. Finally, the return value is placed on the stack again, and the execution continues with the first instruction after the generated code fragment.

Let α be the amount of evaluation steps to reach the end label, and let $l = l' + \alpha$, then s_l is structurally equivalent modulo policy enforcement with s_k. Since calls to policy methods are ignored, the abstract traces are equivalent modulo policy enforcement.

(c) If the execution is in exception handling mode inside a security-relevant method, the last exception handler is unsuitable to handle the exception and the previous method is inside P, then in $s'_{l'+1}$, the exception handling mechanism continues searching inside the previous method, where it will find the exception handler inside the generated code fragment.

In step 12. of the generated code, the exception value is stored (again this does not modify the pre-existing variables in P). Step 13. reloads the arguments and the exception and step 14. calls the exceptional clause of the policy. As the policy methods are passive, the execution returns to step 15. where the exception value is restored. Next, step 16. rethrows the exception. Finally, the execution system steps through all exception handlers inserted by the rewriter, but will not find one.

Let α be the amount of evaluation steps to the first exception handler of original code, and let $l = l' + \alpha$, then s_l is structurally equivalent modulo policy enforcement with s_k. Since calls to policy methods are ignored, the abstract traces are equivalent modulo policy enforcement.

(d) If the execution is in normal execution mode, and the instruction is a throw instruction inside a method of P, or the execution is in exception handling mode and will propagate the exception to a method inside P and the current method is not a security-relevant method, then in $s'_{l'+1}$ we might need to execute exception handlers in P' that are not present in P.

Let α be the amount of evaluation steps to the first exception handler of the original code, and let $l = l' + \alpha$, then s_l is structurally equivalent modulo policy enforcement with s_k. Since calls to policy methods are ignored, the abstract traces are equivalent modulo policy enforcement.

(e) All other cases are done by executing just one instruction of both P and P'.

The transparency theorem is a direct corollary of this lemma.

4 Virtual Methods

To support virtual method calls in our model of the .NET CLR, the following extensions are needed. The VM keeps track of a map of object references to their

actual type (the function $actualTypeOf$). The new instruction *callvirt* performs a virtual call: it looks up the method to be invoked dynamically based on the run-time type of the receiver, and then behaves like a regular call. Figure 11 in the appendix gives the evaluation rule for the *callvirt* instruction.

If our inliner would treat *callvirt* in the same ways as *call*, complete mediation would break. There are three issues to be dealt with.

First, lemma 2 would break. Suppose a security-relevant method $B :: m$ overrides a non-security-relevant method $A :: m$ in a superclass A (Figure 5). A virtual call with static target $A :: m$ would not be processed by the inliner, yet it could lead to an invocation of $B :: m$ at run time.

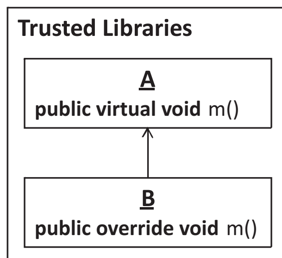

Fig. 5. Violating complete replacement

We deal with this first issue by requiring that the set of security-relevant methods is upward closed.

Definition 12 (Upwards closure of a set of methods). *A set of methods is upwards closed if and only if no method in the set overrides a method that is not in the set.*

In practice, this is not a real restriction, as methods can always be added to the set of security-relevant methods, even if the policy does impose constraints on them.

The second issue arises when inheriting security-relevant methods. Suppose $A :: m$ is a security-relevant method, and the untrusted assembly creates a subclass C of A that does not redefine m (Figure 6). Then a call that statically looks like $C :: m$ is actually a call to $A :: m$. Clearly, we can not require $C :: m$ to be in the set of security-relevant methods, as it is in the untrusted application, and the policy writer typically does not even know it exists.

We address this issue by forbidding inheritance of security-relevant methods in the untrusted assembly. Again, this is no restriction, as overriding is allowed, and the untrusted application could override the security-relevant method and then simply do a base call if the same behavior as the original method is desired. In fact, such a transformation could even be done automatically when no redefinition is found. The base call in the transformed program will be recognized by the inliner as a statically bound call to a security-relevant method.

The third and last issue to address is how to deal with dynamic dispatch to policy methods. Since the method called is determined dynamically, the determi-

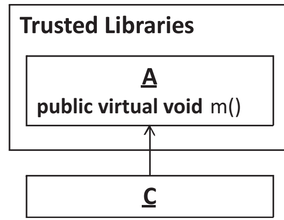

Fig. 6. Inheriting security-relevant methods

nation of the appropriate policy methods now also needs to be done dynamically. We handle this by creating a dispatching method for each virtual method that is security-relevant (See Figure 7). The dispatching method uses runtime tests on the actual type of the target to determine the actual method that will be invoked. If this target method is security-relevant, then the dispatching method calls the corresponding policy method. Otherwise, it returns from the dispatching method. The program rewriter now inserts calls to these dispatching methods instead of the actual policy methods.

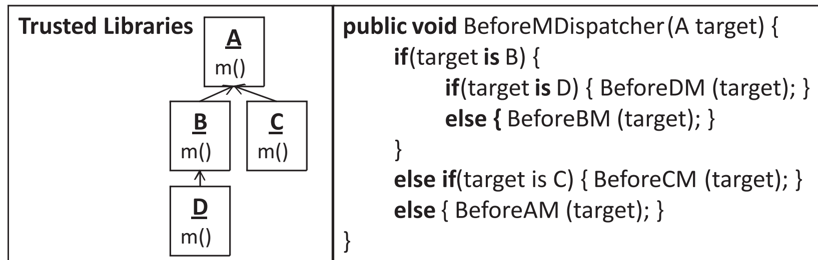

Fig. 7. Dispatching methods

With these three extensions, the resulting inliner is completely and soundly mediating and transparent in the presence of virtual calls.

5 Delegates

Delegates are essentially type safe function pointers. A delegate encapsulates a set of method references of the same signature as the delegate. Calling the delegate invokes all the methods in the set. A delegate can be passed on to other code where it is possible to call the delegate without statically knowing its target methods. Therefore delegates are challenging for a caller-side inliner.

To maintain the complete mediation property of our inliner, we enforce the property that the untrusted application never holds a reference to a delegate that contains a pointer to a security-relevant method. By consequence, a call to a delegate is never a call to a security-relevant method.

First, we extend the program rewriter to deal with the case where the untrusted assembly creates delegates. To create a delegate containing a method

pointer, an application must first load the method pointer on the stack (using the *ldftn* or *ldvirtftn* instruction), and then it must call a Delegate subclass constructor. When the program rewriter parses an attempt to load a security-relevant method pointer on the stack, it creates a new wrapper method inside the application that does a normal call to the security-relevant method, and a pointer to this wrapper method is pushed on the stack. Hence the delegate will not point to the security-relevant method, but to the wrapper method. The program rewriter transforms the body of the wrapper method to insert calls to the policy, thus preserving complete mediation.

Second we need to impose a constraint on the trusted libraries: no method in the trusted API should create delegates that contain security-relevant methods and pass them to the untrusted assembly in any way. For the security-relevant methods we have considered so far, the trusted libraries never create delegates that contain security-relevant methods.

With these extensions, our program rewriter is soundly and completely mediating and transparent, even in the presence of delegates.

6 Implementation

The research reported on in this paper is done in the context of the project Security of Software and Services for Mobile Systems (S3MS) [9]. The inlining algorithm described in this paper has been implemented for the full .NET Framework, and for the .NET Compact Framework running on mobile devices such as smartphones. We used the Mono.Cecil [12] libraries to parse and generate MSIL.

The execution system we describe in this paper hides a part of the complexity of the real .NET CLR. While implementing our approach, we encountered some challenging issues (in addition to the ones we already discussed). For example, the real CLR forbids entering a protected region (a try block) with a non-empty evaluation stack and it resets the evaluation stack when leaving a protected region. Our approach inserts new try-catch handlers, thus we store the entire evaluation stack before entering the try block, instead of just saving the arguments for the security-relevant method. To do so, we implemented a simple one-pass data flow analysis to compute the types on the stack. We store the values on the stack in local variables, and reload them after the catch-blocks.

Furthermore, the real CLR forbids entering a protected region when the current object is not fully initialized in constructors. Initialization statements for fields are executed before an object is fully initialized. If these statements use security-relevant methods, the program rewriter inserts try-catch handlers and the current object is not fully initialized upon entering the try block. We solved this issue in our implementation by generating wrapper methods for these security-relevant methods (like we did with delegates).

Our experience with the implementation is promising. The performance overhead of the inlining is very small (this confirms performance measurements done for other IRM's [2]), and the inliner can handle real-world applications that are used as case studies in the S3MS project.

7 Related Work

Both the current Java Virtual Machine [13] and Microsofts Common Language Runtime [10] use stack inspection to enforce security policies on untrusted code. The security policies that can be enforced in our approach are more flexible, since the entire history of events can be used. Resource constraints are an example of the kind of policies that are not enforceable using stack inspection.

SASI [7] and PoET/PSLang [6] are two policy enforcement tools based on security automata [1]. Both techniques are based on inline reference monitors. Sasi's event system is very powerfull, as arbitrary machine instructions can be monitored, but it has a higher performance impact. PoET/PSLang targets Java applications and Erlingsson and Schneider have shown that it can be used to enforce stack inspection based policies, and that the performance is competitive [2]. In contrast with our approach, they make no claims about complete mediation or transparency.

Naccio [4] also monitors security-sensitive operations, but instead of inserting instructions inside the application or the system libraries, a new wrapper library is created that enforces the security policy and delegates control back to the original libraries.

Polymer [3] is a policy enforcement system based on edit automata [14]. To master the complexity of policies, Polymer supports composition of complex security policies using smaller building blocks. In contrast with our approach, polymer uses callee-side rewriting to enforce security policies.

The work in this paper can been seen as a particular instantiation of Aspect Oriented Programming [15] targeting security policy enforcement on untrusted code. Because our rewriting algorithm is much simpler than the weaving mechanisms in full AOP, it is easier to prove correctness.

The abstract traces in our system are similar to the fully abstract trace semantics of JavaJr [16].

8 Conclusions

In this paper, we propose a caller-side rewriting algorithm for MSIL – the bytecode of the .NET virtual machine – where security checks are inserted around calls to security-relevant methods.

The algorithm has been implemented and can deal with real-world .NET applications. Moreover, the algorithm has been proven correct for a simplified model of MSIL and the .NET virtual machine. To the best of our knowledge, this is the first provably correct inlining algorithm for an intermediate language that supports both virtual methods and delegates.

References

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

2. Erlingsson, U., Schneider, F.B.: IRM enforcement of Java stack inspection. In: IEEE Symposium on Security and Privacy. (2000) 246–255
3. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with polymer. In: PLDI '05, New York, NY, USA, ACM Press (2005) 305–314
4. Evans, D., Twyman, A.: Flexible policy-directed code safety. In: IEEE Symposium on Security and Privacy. (1999) 32–45
5. Saltzer, J., Schroeder, M.: The protection of information in computer systems. IEEE, Vol. 9(63) (1975)
6. Erlingsson, U.: The inlined reference monitor approach to security policy enforcement. PhD thesis, Cornell University (2004) Adviser-Fred B. Schneider.
7. Erlingsson, Schneider: SASI enforcement of security policies: A retrospective. In: WNSP: New Security Paradigms Workshop, ACM Press (2000)
8. Provos, N.: Improving host security with system call policies. In: SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium, Berkeley, CA, USA, USENIX Association (2003) 18–18
9. S3MS: Security of software and services for mobile systems. http://www.s3ms.org/ (2007)
10. European Computer Machinery Association: Standard ECMA-335: Common Language Infrastructure. 4th edition edn. (June 2006)
11. Fruja, N.G.: Type Safety of C# and .NET CLR. PhD thesis, ETH Zurich (2006)
12. Evain, J.: Cecil. http://www.mono-project.com/Cecil
13. Lindholm, T., Yellin, F.: The Java(TM) Virtual Machine Specification (2nd Edition). Prentice Hall PTR (April 1999)
14. Ligatti, J., Bauer, L., Walker, D.: Edit automata: Enforcement mechanisms for run-time security policies. International Journal of Information Security **4**(1–2) (February 2005) 2–16
15. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: Proceedings European Conference on Object-Oriented Programming. Volume 1241. Springer-Verlag, Berlin, Heidelberg, and New York (1997) 220–242
16. Jeffrey, A.S.A., Rathke, J.: Java jr.: Fully abstract trace semantics for a core Java language. In: Proc. European Symposium on Programming. Volume 3444 of Lecture Notes in Computer Science., Springer-Verlag (2005) 423–438

Appendix

A Operational Semantics

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \qquad instr = \mathbf{ldc.x}\ v \\ evalStack' = evalStack, v \qquad S'' = S', (pc+1, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H, S'', E)}$$

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \\ instr = \mathbf{ldloc}\ i \qquad adr = locAdr[i] \qquad v = heapLookup(H, adr) \\ evalStack' = evalStack, v \qquad S'' = S', (pc+1, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H, S'', E)}$$

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \\ instr = \mathbf{stloc}\ i \qquad evalStack = evalStack', v \qquad adr = locAdr[i] \\ H' = heapUpdate(H, adr, v) \qquad S'' = S', (pc+1, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H', S'', E)}$$

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \\ instr = \mathbf{ldarg}\ i \qquad adr = argAdr[i] \qquad v = heapLookup(H, adr) \\ evalStack' = evalStack, v \qquad S'' = S', (pc+1, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H, S'', E)}$$

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \\ instr = \mathbf{starg}\ i \qquad evalStack = evalStack', v \qquad adr = argAdr[i] \\ H' = heapUpdate(H, adr, v) \qquad S'' = S', (pc+1, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H', S'', E)}$$

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \\ instr = \mathbf{br}\ target \qquad S'' = S', (target, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H, S'', E)}$$

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \qquad instr = \mathbf{brtrue}\ target \\ evalStack = evalStack', v \qquad v == 0 \qquad S'' = S', (pc+1, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H, S'', E)}$$

$$\frac{S = S', (pc, locAdr, argAdr, evalStack, mref) \qquad instr = \mathbf{brtrue}\ target \\ evalStack = evalStack', v \qquad v! = 0 \qquad S'' = S', (target, locAdr, argAdr, evalStack', mref)}{(H, S, E) \rightarrow (H, S'', E)}$$

Fig. 8. Evaluation rules for normal execution (Part 1)

$$
\frac{\begin{array}{c}
S = S', (pc, locAdr, argAdr, evalStack, mref) \\
instr = \mathbf{call}\ T :: M \qquad argTypes(T :: M) = \overrightarrow{A}_{1..n} \\
evalStack = evalStack', \overrightarrow{v}_{1..n} \qquad argAdr' = \overrightarrow{a}_{1..n}\ (with\ a_i\ fresh) \\
H' = heapUpdate(H, \overrightarrow{a}_{1..n}, \overrightarrow{v}_{1..n}) \qquad locTypes(T :: M) = \overrightarrow{L}_{1..m} \\
locAdr' = \overrightarrow{l}_{1..m}\ (with\ l_i\ fresh) \qquad H'' = heapUpdate(H', \overrightarrow{l}_{1..m}, defVal(\overrightarrow{L}_{1..m})) \\
S'' = S', (pc, locAdr, argAdr, evalStack', mref), (0, locAdr', argAdr', [], T :: M)
\end{array}}{(H, S, E) \rightarrow (H'', S'', E)}
$$

$$
\frac{\begin{array}{c}
S = S', (pc2, locAdr2, argAdr2, evalStack2, mref2), (pc, locAdr, argAdr, evalStack, mref) \\
instr = \mathbf{ret} \\
retType(mref) = void \qquad S'' = S', (pc2 + 1, locAdr2, argAdr2, evalStack2, mref2)
\end{array}}{(H, S, E) \rightarrow (H, S'', E)}
$$

$$
\frac{\begin{array}{c}
S = S', (pc2, locAdr2, argAdr2, evalStack2, mref2), (pc, locAdr, argAdr, evalStack, mref) \\
instr = \mathbf{ret} \qquad retType(mref) \neq void \qquad evalStack = evalStack', v \\
evalStack2' = evalStack2, v \qquad S'' = S', (pc2 + 1, locAdr2, argAdr2, evalStack2', mref2)
\end{array}}{(H, S, E) \rightarrow (H, S'', E)}
$$

$$
\frac{\begin{array}{c}
S = S', (pc, locAdr, argAdr, evalStack, mref) \\
instr = \mathbf{throw} \qquad evalStack = evalStack', val \\
E' = (val, 0) \qquad cal \neq null \qquad S'' = S', (pc, locAdr, argAdr, evalStack', mref)
\end{array}}{(H, S, E) \rightarrow (H, S'', E')}
$$

Fig. 9. Evaluation rules for normal execution (Part 2)

$$
\frac{\begin{array}{c}
S = S', (pc, locAdr, argAdr, evalStack, mref) \\
E = (val, n) \qquad n = \#excHa(mref) \qquad E' = (val, 0)
\end{array}}{(H, S, E) \rightarrow (H, S', E')}
$$

$$
\frac{\begin{array}{c}
S = S', (pc, locAdr, argAdr, evalStack, mref) \\
E = (val, n) \qquad n < \#excHa(mref) \qquad excHa(mref)[n] = (from, to, type, cfrom, cto) \\
from > pc\ \vee\ pc \geq to\ \vee\ actualTypeOf(val) \not< type \qquad E' = (val, n + 1)
\end{array}}{(H, S, E) \rightarrow (H, S, E')}
$$

$$
\frac{\begin{array}{c}
S = S', (pc, locAdr, argAdr, evalStack, mref) \\
E = (val, n) \qquad n < \#excHa(mref) \qquad excHa(mref)[n] = (from, to, type, cfrom, cto) \\
from \leq pc < to \qquad actualTypeOf(val) < type \\
E' = (null, 0) \qquad evalStack' = val \qquad S'' = S', (cfrom, locAdr, argAdr, evalStack', mref)
\end{array}}{(H, S, E) \rightarrow (H, S'', E')}
$$

Fig. 10. Evaluation rules for exception handling

$$
\frac{\begin{array}{c}
S = S', (pc, locAdr, argAdr, evalStack, mref) \\
instr = \mathbf{callvirt}\ T :: M \qquad isVirtual(T :: M) = true \\
argTypes(T :: M) = T, \overrightarrow{A}_{1..n} \qquad evalStack = evalStack', t, \overrightarrow{v}_{1..n} \qquad \overrightarrow{args} = t, \overrightarrow{v}_{1..n} \\
T' :: M = lookupMethod(actualTypeOf(t), T :: M) \qquad argAdr' = \overrightarrow{a}_{0..n}\ (with\ a_i\ fresh) \\
H' = heapUpdate(H, \overrightarrow{a}_{1..n}, \overrightarrow{args}) \qquad locTypes(T' :: M) = \overrightarrow{L}_{1..m} \\
locAdr' = \overrightarrow{l}_{1..m}\ (with\ l_i\ fresh) \qquad H'' = heapUpdate(H', \overrightarrow{l}_{1..m}, defVal(\overrightarrow{L}_{1..m})) \\
S'' = S', (pc, locAdr, argAdr, evalStack', mref), (0, locAdr', argAdr', [], T' :: M)
\end{array}}{(H, S, E) \rightarrow (H'', S'', E)}
$$

Fig. 11. Evaluation rule for virtual calls