From Stack Inspection to Access Control: A Security Analysis for Libraries

Frédéric Besson
Microsoft Research

Tomasz Blanc
INRIA Rocquencourt

Cédric Fournet
Microsoft Research

Andrew D. Gordon
Microsoft Research

Abstract

We present a new static analysis for reviewing the security of libraries for systems, such as JVMs or the CLR, that rely on stack inspection for access control. We describe its implementation for the CLR. Our tool inputs a set of libraries plus a description of the permissions granted to unknown, potentially hostile code. It constructs a permission-sensitive call graph, which can be queried to identify potential security defects. It has been applied to large pre-existing libraries.

We also develop a new formal model of the essentials of access control in the CLR (types, classes and inheritance, access modifiers, permissions, and stack inspection). In this model, we state and prove the correctness of the analysis.

1. Motivation and Outline

In modern, networked systems, the addition of software components is frequent and largely automated. These components may have diverse origins; they can be applets, plug-ins, macros in documents, or programs downloaded from the Web. Their intermingled code ends up sharing the same local resources (CPU, memory, files), but not necessarily the same level of trust.

To enforce access control in the presence of potentially hostile code, extensible systems such as the Java Virtual Machine (JVM) and the Common Language Runtime (CLR) provide fine-grained security mechanisms, including a stack inspection mechanism that can determine the permissions of each running piece of code as a function of the stack [10, 6, 16]. Permissions are first associated with pieces of code according to their level of trust, which typically depends on the origin of the code and the local security policy. Then, before accessing a sensitive resource, the call stack is inspected to verify that every caller has been granted the requested permissions.

Stack inspection is a flexible preventative measure but is also a source of complications. For instance, library code should be able to interact with a variety of programs; however, the behaviour of the library (and its security) now depends on the local security configuration and the runtime stack. As may be expected, it becomes quite hard to validate the security of a library by testing and code review. Related difficulties include optimising performance, and constructing and maintaining accurate documentation.

This paper describes the design, formalisation, and implementation of a new static analysis tool that addresses these difficulties. Our tool analyses the use of runtime permissions in the CLR, with its existing mechanisms and libraries, but the principles seem applicable in other settings, such as the JVM.

The tool constructs a call graph from two inputs: (1) a collection of compiled input libraries, and (2) a description of the permissions assigned to the as yet unknown code to be loaded at runtime. We say the known, library code (typically granted many permissions) is *trusted*, whereas the unknown, dynamically-loaded code (typically granted few permissions) is *semi-trusted*. Our main purpose is to help find honest mistakes in trusted code that might be exploited by maliciously crafted semi-trusted code.

The call graph includes nodes for both known and unknown code, with multiple nodes for each piece of code that can be executed with different run-time permissions. The construction is otherwise simple in principle—if not in detail, as our implementation handles all CLR instructions. Significant novelties, compared to previous call graph constructions, include the sensitivity to permissions when generating the graph, and the analysis of an open system, where arbitrary unknown code may call into (or inherit from) known libraries. Given this permission-sensitive call graph, we run a variety of queries to detect potential security defects, such as the unintended reachability of dangerous methods. These queries are inspired by typical defects in CLR code. Our aim is not to fully verify access control, but instead to focus human effort during security code reviews. We summarise experimental results from analysing substantial existing libraries.

To provide a formal foundation for our call graph construction, we define a new model of stack inspection within

To appear in the proceedings of the 17th IEEE Computer Security Foundations Workshop, June 28–30, 2004.

the CLR. Our model, BIL-SEC, is a variation of Baby IL [11], a subset of the CLR's intermediate language (IL) previously introduced for the study of type safety. BIL-SEC reflects the essential features as regards access control (types, classes and inheritance, access modifiers, permissions, and stack inspection). Hence, whilst avoiding many details of the full CLR, it better captures the specific characteristics of our implementation than previous work on abstract λ-calculus models of stack inspection [21, 9].

Suppose we have a call graph for a particular trusted library, and consider an arbitrary choice of semi-trusted code to be loaded at runtime. Our main formal result states that, if there can be a sequence of calls starting from semi-trusted code and ending with a particular (dangerous) method in trusted code, then there is a corresponding path in the graph. Hence, a query showing there is no such path implies no dynamically loaded code can reach the method in question.

The paper is organised as follows. Section 2 reviews the CLR and surveys some of its security mechanisms, including run-time permissions. Section 3 introduces the main ideas of our analysis using a running example in BIL-SEC with a typical defect, and its call graph. Section 4 defines BIL-SEC. Section 5 formalises our call graph analysis and states our correctness result. Section 6 surveys our implementation for the full CLR. Section 7 describes queries and experimental results on libraries. Section 9 closes with a discussion of related works and some conclusions. An appendix contains auxiliary definitions. An online version has sample code in C$^\sharp$ and detailed proofs [4].

2. Stack-Based Access Control (Review)

The CLR and its Intermediate Language. The CLR is a memory-managed, typed, object-oriented platform [6]. An *assembly* is its unit of code deployment, typically a single file, containing metadata plus actual implementation code. Metadata includes details of the class hierarchy, as well as security-related information such as digital signatures as evidence of origin, and constraints on the security policy for that assembly. Implementation code is predominantly expressed in an intermediate language (IL) obtained by compiling from a range of programming languages; as usual, an advantage of targeting a tool at an intermediate language is that its analysis applies to software written in any one or a mixture of the source languages. More importantly, we cannot assume that untrusted assemblies comply with any rule that is not checked at the IL level: some security concerns may be invisible in high-level languages, and only appear at the level of IL.

The CLR allows the controlled interaction of a set of dynamically loaded, partially trusted assemblies, that share resources such as the stack and heap, as well as access to fully trusted system libraries, all running within the same operating system process. To control access to these resources, the CLR depends on a range of security mechanisms [16], including type safety and access modifiers, as well as stack inspection. The CLR has a fairly standard class-based type system, with modifiers (*private*, *protected*, etc) controlling the visibility of fields, methods, and other class members. An assembly's metadata and code are checked for type safety and conformance to access modifiers during loading and JIT compilation.

Permissions and Stack Inspection. Code access rights are represented at runtime using objects of particular classes, named *permissions*. Access to each sensitive resource is associated with a particular permission. Permissions can have a complex structure; for instance, an object of class *FileIOPermission* describes access to files, using a combination of flags (read, write,...) and file path expressions.

When an assembly is loaded into the CLR, its access rights are determined by its metadata and the current security policy. The resulting *static permissions*, or S, are associated with all code from that assembly. These static permissions give an upper bound on the permissions that the code can actually use. Factors affecting the static permissions include the assembly's apparent origin (such as the Internet, the intranet, or the local disc), any digital signatures, and metadata requests to be granted or denied particular permissions. The security policy is configurable for each CLR installation, the default being to grant most permissions to code written by the user, and only very few permissions to downloaded code.

During execution, the *dynamic permissions*, or D, default to being the least privileges of all callers on the stack, that is, the intersection of their static permissions. To guard access to some sensitive resource associated with a particular permission P, trusted code evaluates $\mathbf{demand}\ P$, to tell whether P is present in the dynamic permissions. This succeeds if the permission is in the static permissions of the immediate caller and moreover in the static permissions of each caller on the stack. In some harmless situations, such as writing a temporary file, this default stack inspection is too restrictive. To override the default, trusted code evaluates $\mathbf{assert}\ P$ to add P to the dynamic permissions, provided that P appears in the static permissions for this code. By asserting P, the trusted code takes responsibility for any demands for P, until the completion of the current method. Such privilege elevations are dangerous, and deserve careful review.

This brief tour of stack inspection omits many details, including declarative security attributes and useful refinements of the security model—such as variants of demands, known as $\mathbf{linkdemand}$s and $\mathbf{inheritancedemand}$s, that check for a permission in the static permissions of a caller or a subclass, respectively, when code is loaded into the system, instead of every time it is executed. Still, we are now in a position to discuss defects that occur in practice, and our tool for exploring them.

3. Permission-Sensitive Call Graphs, By Example

Access control in the CLR relies on an implicit, global safety invariant; its correctness may be compromised by errors scattered through a large body of code. In fact, during the development of libraries for the CLR, numerous security defects involve permissions, but these defects often fall into a few simple patterns. (This may be partly due to programmers confronting stack inspection for the first time.) Moreover, permission usage is largely data independent. Typically, the permission objects are either constructed just before a demand or assert, or read from a constant static field for the class.

Altogether, this suggests that a permission-specific, large-scale analysis of code can be useful in reviewing and improving the usage of permissions across libraries. Our analysis consists of constructing and querying a call graph given trusted library code as input. Original features of our call graphs include: (1) sensitivity to the dynamic permissions available at each call, and (2) nodes corresponding to unknown semi-trusted code, as well as nodes for the known input libraries. Next, we list some library code that includes a security defect of the sort our analysis is aimed at, and show the corresponding call graph.

All code listings in this paper are in BIL-SEC, which we define formally in Section 4. Its syntax is very similar to the standard IL stack-based assembly language; a minor difference is that BIL-SEC has primitive instructions for **demand** and **assert** whilst in IL these are method calls.

An Example in BIL-SEC. We have devised some simple classes to illustrate access control in the CLR and a typical code defect. The class *File* and its subclass *CFile* are trusted library code; their static permissions include *FilePermission*, which guards the private file-deletion primitive *Win32::Delete*.

```
// in a trusted library
public class File {
    public void Delete(string s){
        demand FilePermission
        newobj Win32::.ctor()
        ldarg 1
        callvirt void Win32::Delete(string)
    }
    public void Backup(string s){
        demand FilePermission
        ...
    }
    protected string tempfile
    protected void Cleanup(){
        ldarg 0
        (ldarg 0) ldfld string File::tempfile
        callvirt void File::Delete(string)
    }
}
```

```
// in another naive, trusted library
public class CFile : File {
    protected void Commit(){
        ....
        assert FilePermission
        ldarg 0
        ldc.string "backupfile"
        callvirt void File::Backup(string)
        ...
        ldarg 0
        callvirt void File::Cleanup()
    }
}
```

The three methods exposed by *File* guard access to the private *Win32::Delete* method by demanding *FilePermission*—directly in case of *Delete* and *Backup*; indirectly in case of *Cleanup* via the call to *Delete*.

Judging correctly that calling *File::Backup* on "backupfile" is harmless, whatever the calling context, the author of method *CFile::Commit* asserts *FilePermission* to prevent any security exception. By mistake, this amplification of the dynamic permissions carries over to the subsequent call of *File::Cleanup*, which is not harmless, since it deletes the file in the field *tempfile*.

The method *BadFile::DeleteAny* of the semi-trusted class *BadFile* exploits this defect. Its static permissions do not include *FilePermission*, but nonetheless it can inherit from the public class *CFile*.

```
// in some hostile, semi-trusted code
public class BadFile : CFile {
    public void DeleteAny(string s){
        // Assign s to tempfile field
        (ldarg 0) (ldarg 1) stfld string File::tempfile
        // Delete the file s
        (ldarg 0) callvirt void CFile::Commit()
    }
}
```

By inheriting from *CFile* it gains access to the protected members *tempfile* and *Cleanup*, and by calling *CFile::Commit* it gains access to *FilePermission* and can delete any file. (The *protected* modifier is the same as *private* except that derived classes still have access.) This example shows an attack on protected members via inheritance, showing that security analyses need to be sensitive to the class hierarchy. The same exploit would work without inheritance if *tempfile* and *Cleanup* were public.

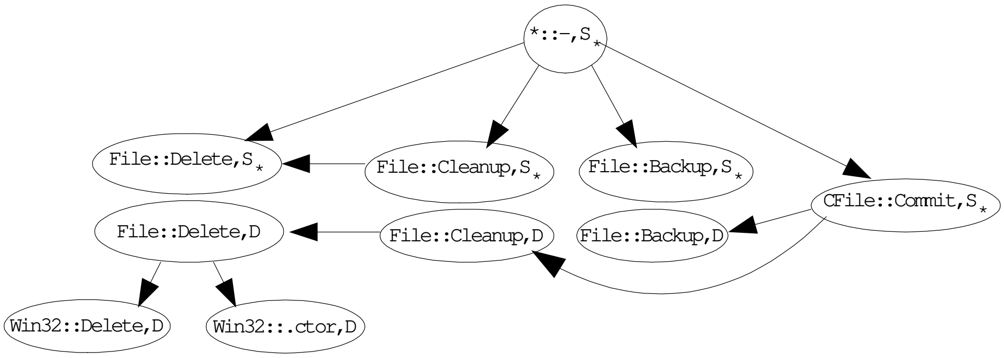

Figure 1: Call graph for example libraries ($D = S_\star \uplus \{\textit{FilePermission}\}$)

Call Graph for the Example. Our goal is to identify anomalous or defective control flows in libraries, and in particular to identify potential paths stretching from semi-trusted code to dangerous operations, such as *Win32::Delete* in our example. Given a collection of trusted library code, and knowing the permissions granted to semi-trusted code, our tool constructs a permission-sensitive call graph, which summarises all control flows from arbitrary semi-trusted code into the library code.

For instance, Figure 1 shows the call graph corresponding to the two example classes *File* and *CFile*. Let $S_\star$ be the set of static permissions granted to semi-trusted code. We assume *FilePermission* $\notin S_\star$. Each node is a pair (M, D) where M refers to a method implementation, such as *File::Delete*, and D is the set of dynamic permissions with which M is called. The distinguished method $\star$::$-$, which appears paired with $S_\star$ as the root node, represents the unknown semi-trusted code. Each edge represents a possible call from one method to another.

The first three edges from the root node summarise calls from semi-trusted code into the *File* library. These calls are with dynamic permissions $S_\star$. There is no edge from $(\textit{File::Delete}, S_\star)$ to *Win32::Delete* because the demand for *FilePermission* always fails with dynamic permissions $S_\star$. The remaining edge from the root node represents calls from semi-trusted code to the method *CFile::Commit*, which immediately asserts *FilePermission*. So all the edges from this node are to nodes with dynamic permissions $D = S_\star \cup \{\textit{FilePermission}\}$. In particular, there is a path from $(\textit{CFile::Commit}, S_\star)$ to $(\textit{File::Delete}, D)$, from which an edge leads to *Win32::Delete*, since the demand for *FilePermission* succeeds against D.

The graph shows semi-trusted code cannot trigger calls to *Win32::Delete* when given access only to *File*, but can, if given access also to *CFile*. Given the graph, we can easily write queries to detect such suspicious paths from semi-trusted code to critical methods. These paths may or may not be harmful, but since the graph is an over-approximation, we can safely limit code review to the methods on these paths.

Outline of the Analysis. Our analysis is sensitive to dynamic permissions and many details of the security model, but is otherwise quite coarse. In the terminology of control flow analyses, this amounts to a particular choice of context-sensitivity. Whereas, for instance, a standard n-CFA [12] would keep track of n frames on the stack, we effectively keep track of a summary of the whole stack that suffices to evaluate demands. (Of course, we would benefit from any additional context-sensitivity in the call graph, as long as the analysis terminates.) Alternatively, our analysis can be seen as abstracting a security-passing style implementation of stack inspection [24], where dynamic permissions are systematically computed and passed as an extra parameter to every method, instead of being extracted lazily from the stack.

For the purpose of our analysis, the *abstract value* of a runtime variable is a set of types, an upper bound on the types of all runtime values that may flow to that variable. It is insufficient simply to track the static types of variables, since there is a profound dependency of control-flow on data-flow induced by virtual and interface calls. We use abstract values to track which types may flow to each call site. Still, our analysis is a refinement of the type system for the CLR, and sometimes falls back on type safety, for instance when loading from an array of boxed values.

We include the special symbolic class, $\star$, in the domain of abstract values, to stand for all classes that may be defined in unknown semi-trusted code. We add edges to the class hierarchy so that $\star$ is a subtype of known trusted classes, according to the rules of inheritance. In our example, $\star$ is a subtype of both the (unsealed) public classes *File* and *CFile*, and hence represents an unknown class such as *BadFile*.

To construct the call graph, for every method reachable from semi-trusted code with some dynamic permissions, we construct a distinct node. To each formal argument, method result, local variable, static variable, field, and entry on the stack that has a boxed type, we associate a variable whose abstract value collects the dynamic types that may flow to it. From the code of the program, we obtain constraints (for instance inclusions) between these variables, which we solve by an iterative method. During this iteration, the abstract values of variables may increase, triggering generation of further nodes with different dynamic permissions. Since the abstract domains are finite, the iteration always terminates.

The remainder of the paper divides into formal and informal parts. Sections 4 and 5 formalise the ideas of this section in terms of BIL-SEC. Sections 6 and 7 describe our implementation for the CLR.

4. Modelling Stack-Based Access Control

Our formal model, BIL-SEC, derives from BIL [11], a fragment of IL focusing on its main object-oriented features. To obtain BIL-SEC, we add static and dynamic permissions, plus **demand** and **assert** instructions, and omit features—such as structures and pointers into the stack—unrelated to stack inspection. BIL and BIL-SEC are still Turing complete.

All code runs in the context of an execution environment that defines the available classes and methods, their implementations, and additional data such as types and permissions. We begin our formal model with finite sets of all defined class, field, method and permission names. In BIL-SEC, unlike IL, permissions are atomic constants.

Classes, Fields, Methods, Permissions:

$c, d \in \mathit{Class}$	class name
$\mathbf{System.Object} \in \mathit{Class}$	root of hierarchy
$f \in \mathit{Field}$	field name
$\ell \in \mathit{Meth}$	method name
$P \in \mathit{Permission}$	permission name
$\mathcal{P}(\mathit{Permission}) = \mathit{PermissionSet}$	permission set
$S, D \in \mathit{PermissionSet}$	

There are three kinds of data type: void, integer, and reference (for pointers to heap-allocated objects). Types are the basis for the syntax of method and constructor signatures, and references. For simplicity, each class has exactly one constructor, whose parameters are simply the initial values of all of the fields of the class.

Types, Signatures and References:

$A, B \in \mathit{Type} ::= \mathbf{void} \mid \mathbf{int32} \mid \mathbf{class}\ c$	type: void, integer, or reference
$\mathit{sig} \in \mathit{Sig} ::= B\ \ell(A_1, \ldots, A_n)$	method signature
$\mathit{ksig} \in \mathit{Ksig} ::= \mathbf{void}\ \mathbf{.ctor}(A_1, \ldots, A_n)$	constructor signature
$M ::= c{::}\mathit{sig}$	method reference
$K ::= c{::}\mathit{ksig}$	constructor reference
$c{::}\mathit{sig} \triangleq B\ c{::}\ell(A_1, \ldots, A_n)$ where $\mathit{sig} = B\ \ell(A_1, \ldots, A_n)$	
$c{::}\mathit{ksig} \triangleq \mathbf{void}\ c{::}\mathbf{.ctor}(A_1, \ldots, A_n)$ where $\mathit{ksig} = \mathbf{void}\ \mathbf{.ctor}(A_1, \ldots, A_n)$	

We can now specify an *execution environment* as given by an inheritance relation $\mathit{inherits}$, plus three total functions specifying the fields, methods, and static permissions of each class. We assume all method bodies are well-typed. Appendix A details the evaluation rules of BIL-SEC and our (standard) assumptions on $\mathit{inherits}$. The extended version of this paper also contains the typing rules.

Execution Environment:
$(\mathit{inherits}, \mathit{fields}, \mathit{methods}, \mathit{statics})$

$\mathit{inherits} \subseteq \mathit{Class} \times \mathit{Class}$	class hierarchy
$\mathit{fields} \in \mathit{Class} \rightarrow (\mathit{Field} \stackrel{\text{fin}}{\rightarrow} \mathit{Type})$	fields of a class
$\mathit{methods} \in \mathit{Class} \rightarrow (\mathit{Sig} \stackrel{\text{fin}}{\rightarrow} \mathit{Class} \times \mathit{Body})$	methods of a class
$\mathit{statics} \in \mathit{Class} \rightarrow \mathit{PermissionSet}$	static permissions

The function $\mathit{fields}(c)$ returns a partial map from field names to their types. The domain of the map consists of the fields actually defined for the class c. The function $\mathit{methods}(c)$ returns a partial map from signatures sig to method implementations. The domain of the map consists of the signatures actually defined for the class c. Its range provides, for each defined method $c{::}\mathit{sig}$, the superclass d that implements the method and the method body b. We make the implementation class explicit because it determines the static permissions attached to the method body b. The function $\mathit{statics}(c)$ gives the static permissions associated with class c.

In BIL-SEC, like BIL, we specify method bodies using a postfix applicative syntax, that closely corresponds to the syntax of IL assembler. The following syntax is a subset of BIL, apart from the new instructions **assert** and **demand**. These operations are not present in IL as instructions, but exist in system libraries as native methods that access internal runtime data structures. Our **demand** instruction is a conditional with two branches, but in IL is a method call whose failure triggers a security exception. An omitted **else** branch, as in the example in Section 3, simply returns **void**.

Applicative Expressions for Method Bodies:

$i4$	32 bit signed integer
$a, b \in \mathit{Body} ::=$	method body
$\mathbf{ldc.i4}\ i4$	load integer
$a\ b$	run a then run b
$\mathbf{assert}\ P\ a$	assert P then run a

demand P a **else** b	demand P then run a, else run b
ldarg j	load method argument j if $j > 0$ or self if $j = 0$
a **starg** j	store result of a into argument $j > 0$
$a_1 \ \cdots \ a_n$ **newobj** K	create new object with fields $a_1, \ldots, a_n$
$a_0\ a_1 \ \cdots \ a_n$ **callvirt** M	call M on object a_0 with arguments $a_1, \ldots, a_n$
a **ldfld** A $c::f$	load field f of type A from a of class c
a b **stfld** B $c::f$	store b of type B into field f of a in class c

The typing rules and big-step imperative operational semantics for BIL are easily adapted to accommodate stack inspection. Appendix A gives the detailed definitions. The new operational semantics takes parameters S and D to track the static and dynamic permissions of the code being evaluated. The new rules for **demand** and **assert** correspond closely to the informal semantics in Section 2.

Suppose M is a method reference and C is a set of classes. Let "M is reachable from C" mean intuitively that by creating a new object of class $c \in C$ in an empty store, and calling its method ℓ, there is an evaluation during which a virtual call resolves to the particular method implementation M. Method reachability is formalised in Appendix A, and is the subject of a theorem concerning the flow analysis for BIL-SEC, presented next.

5. Modelling a Permission-Sensitive Analysis

This section describes a permission-sensitive analysis in the formal setting of BIL-SEC; this formal analysis is considerably simpler than the one described in Section 6 for the full IL, yet captures many of the main ideas. We state a soundness result (Theorem 1): if a trusted node is unreachable from any untrusted node in the flow graph, then in fact the corresponding trusted method is unreachable from any untrusted code.

Environments with Two Levels of Trust. To represent code with different levels of trust, we partition *Class* in the execution environment into trusted classes (libraries, local code that are available at analysis time) and untrusted classes (applets, plug-ins that are unknown at analysis time). Given this partition, we refine our definition of environments to separate trusted code and untrusted code. Our analysis will depend only on the trusted code.

We model the outcome of evaluating the static security policy and access modifiers (such as **public**, **private**, **virtual**, etc) on trusted libraries by three sets: *Public* (methods callable from untrusted classes), *Virtual* (methods overridable in untrusted classes), and $S_\star$, the static permissions assigned to untrusted classes.

Partially-Trusted Environments:
$\mathcal{E} = (E, \mathit{Trusted}, S_\star, \mathit{Public}, \mathit{Virtual})$

$\mathit{Trusted} \subseteq \mathit{Class}$	trusted classes
$\mathit{Untrusted} \triangleq \mathit{Class} \setminus \mathit{Trusted}$	untrusted classes
$S_\star \subseteq \mathit{PermissionSet}$	permissions for untrusted code
$\mathit{Methods} \triangleq \{c::sig \mid \mathit{methods}(c)(sig) = d, b\}$	all defined method references
$\mathit{Public} \subseteq \mathit{Methods}$	callable by untrusted code
$\mathit{Virtual} \subseteq \mathit{Methods}$	overridable by untrusted code

For each $c, d \in \mathit{Class}$ such that d *inherits* c and $c::sig \in \mathit{Methods}$, we have:
(1) Trust decreases with inheritance:
$d \in \mathit{Trusted} \Rightarrow c \in \mathit{Trusted}$
(2) *Public* is invariant by inheritance:
$d::sig \in \mathit{Public} \Leftrightarrow c::sig \in \mathit{Public}$
(3) *Virtual* decreases with inheritance:
$d::sig \in \mathit{Virtual} \Rightarrow c::sig \in \mathit{Virtual}$

For each $c \in \mathit{Class}$ and $d \in \mathit{Untrusted}$ such that $\mathit{methods}(c)(sig) = (d, b)$, we have:
(4) $\mathit{statics}(d) = S_\star$
(5) $M \in \mathit{Public}$ for every **callvirt** M occurring in b
(6) $d::sig \in \mathit{Virtual}$

Let $E_{|\,\mathit{Trusted}}$ and $\mathcal{E}_{|\,\mathit{Trusted}}$ be obtained from E and $\mathcal{E}$ by restricting the domains of *Public*, *Virtual*, *inherits*, *fields*, *methods*, *statics* from *Class* to *Trusted*:

$$\mathcal{E}_{|\,\mathit{Trusted}} \triangleq (E_{|\,\mathit{Trusted}}, \mathit{Trusted}, S_\star, \mathit{Public}_{|\,\mathit{Trusted}}, \mathit{Virtual}_{|\,\mathit{Trusted}})$$

$$E_{|\,\mathit{Trusted}} \triangleq (\mathit{inherits}_{|\,\mathit{Trusted}}, \mathit{fields}_{|\,\mathit{Trusted}}, \mathit{methods}_{|\,\mathit{Trusted}}, \mathit{statics}_{|\,\mathit{Trusted}})$$

We require that both E and $E_{|\,\mathit{Trusted}}$ are valid execution environments.

Abstraction of Types. Our analysis associates each body with an abstract value, the set of possible dynamic types of its result. The analysis depends only on $\mathcal{E}_{|\,\mathit{Trusted}}$, and is independent of the untrusted classes in $\mathcal{E}$, understood to be known only after the analysis. To track unknown, untrusted classes during the analysis, we introduce a new reference type **class** $\star$, and include it in the set of abstract types. In some circumstances, for instance when considering arguments of a *Public* method, the only safe assumption to be made about a symbolic value is that it is well-typed. Hence, we introduce a type-safe abstraction $sub^\sharp(A^\sharp)$ to define all the potential abstract types of a result, according to its type. As every class is inheritable in BIL-SEC, **class** $\star$ is present in $sub^\sharp$(**class** c) for any trusted c.

Abstract Types $Type^\sharp$**:**

$$A^\sharp, B^\sharp \in Type^\sharp ::= \mathbf{void} \mid \mathbf{int32} \mid \mathbf{class}\ \star \mid \mathbf{class}\ c \quad (c \in Trusted)$$

Type-Safe Abstraction $sub^\sharp(A) \subseteq Type^\sharp$**:**

$$sub^\sharp(\mathbf{void}) = \{\mathbf{void}\}$$
$$sub^\sharp(\mathbf{int32}) = \{\mathbf{int32}\}$$
$$sub^\sharp(\mathbf{class}\ c) = \{\mathbf{class}\ d \mid d \in Trusted \wedge d\ inherits\ c\} \cup \{\mathbf{class}\ \star\}$$

Constraints and their Generation. Next, we define the syntax of nodes, symbolic values, and constraints used in our analysis. A trusted node α of the graph is a pair (M, D) where M is a method implementation and D is a set of dynamic permissions with which it is reachable. There may be multiple nodes for the same method but with different dynamic permissions. A symbolic value t represents the values that flow as arguments and results to and from nodes. The syntax includes symbolic variables λ, references to an argument $\alpha.i$ ($\alpha.0$ corresponds to the caller-object) or the result $\alpha.result$ of a node, and sets of abstract types $\{A_1^\sharp, \ldots, A_n^\sharp\}$. A constraint C on the graph is a conjunction built from a set constraint primitive $t \subseteq t'$ and a special primitive $VCALL$ to represent virtual call resolution. We define the semantics of constraints later in this section.

Trusted Nodes, Symbolic Values, Constraints:

$\alpha, \beta ::= (M, D)$	trusted node
$t ::= \lambda \mid \alpha.result \mid \alpha.i \mid \{A_1^\sharp, \ldots, A_n^\sharp\}$	symbolic value
$C ::=$	constraint
$\mathbf{T}$	true
$C \wedge C'$	conjunction
$t \subseteq t'$	inclusion
$VCALL(\alpha, t_0, t_1, \ldots, t_n, \lambda)$	virtual call

We generate constraints in an operational style: a derivation $b \Rightarrow_{D'}^{\alpha} t \mid C$ means that the expression b at node α, with current dynamic permissions D' returns a symbolic value t subject to the constraints C. Informally, t represents the set of types of all possible values returned by b when it is executed in α with dynamic permissions D'. The analysis of **demand** instructions is sensitive to the current dynamic permissions D', which because of prior **assert**s may not equal the dynamic permissions D associated with the current node α.

The constraints generated by the following rules are predicates on the abstract values that may flow as method arguments and results, and on which nodes are reachable. We have stipulated when defining a well-formed execution environment that all method bodies are well-typed. Hence, the rules below assume—and do not attempt to enforce—that bodies are well-typed.

Constraint Generation for Method Bodies: $b \Rightarrow_D^\alpha t \mid C$**:**

(Gen **ldc**)

$$\frac{}{\mathbf{ldc.i4}\ i4 \Rightarrow_D^\alpha \{\mathbf{int32}\} \mid \mathbf{T}}$$

(Gen **ldarg**)

$$\frac{}{\mathbf{ldarg}\ j \Rightarrow_D^\alpha \alpha.j \mid \mathbf{T}}$$

(Gen **starg**)

$$\frac{a \Rightarrow_D^\alpha t_a \mid C_a}{a\ \mathbf{starg}\ j \Rightarrow_D^\alpha \{\mathbf{void}\} \mid C_a \wedge t_a \subseteq \alpha.j}$$

(Gen Seq)

$$\frac{a \Rightarrow_D^\alpha t_a \mid C_a \qquad b \Rightarrow_D^\alpha t_b \mid C_b}{(a\ b) \Rightarrow_D^\alpha t_b \mid C_a \wedge C_b}$$

(Gen **assert**)

$$\frac{b \Rightarrow_{D \cup (P \cap statics(c))}^\alpha t \mid C \qquad \alpha = (c{::}sig, D')}{\mathbf{assert}\ P\ b \Rightarrow_D^\alpha t \mid C}$$

(Gen **demand**)

$$\frac{a_{P \in D} \Rightarrow_D^\alpha t \mid C}{\mathbf{demand}\ P\ a_{true}\ \mathbf{else}\ a_{false} \Rightarrow_D^\alpha t \mid C}$$

(Gen **ldfld**)

$$\frac{a \Rightarrow_D^\alpha t_a \mid C_a}{a\ \mathbf{ldfld}\ A\ c{::}f \Rightarrow_D^\alpha sub^\sharp(A) \mid C_a}$$

(Gen **stfld**)

$$\frac{a \Rightarrow_D^\alpha t_a \mid C_a \qquad b \Rightarrow_D^\alpha t_b \mid C_b}{a\ b\ \mathbf{stfld}\ A\ c{::}f \Rightarrow_D^\alpha \{\mathbf{void}\} \mid C_a \wedge C_b}$$

(Gen **newobj**)

$$\frac{(a_i \Rightarrow_D^\alpha t_i \mid C_i)^{i \in 1..n}}{a_1\ \cdots\ a_n\ \mathbf{newobj}\ c{::}ksig \Rightarrow_D^\alpha \{\mathbf{class}\ c\} \mid C_1 \wedge \cdots \wedge C_n}$$

(Gen **callvirt**)

$$\frac{(a_i \Rightarrow_D^\alpha t_i \mid C_i)^{i \in 0..n} \qquad \lambda \text{ fresh} \qquad \beta = (M, D)}{a_0\ a_1\ \cdots\ a_n\ \mathbf{callvirt}\ M \Rightarrow_D^\alpha \lambda \mid C_0 \wedge \cdots \wedge C_n \wedge VCALL(\beta, t_0, \ldots, t_n, \lambda)}$$

The rule (Gen **callvirt**) introduces a fresh variable, λ, to represent the result of each virtual call in a method body. Consider a method body b implemented in c, that is, $methods(c)(sig) = (c, b)$ for some implementation node $\alpha = (c{::}sig, D)$. We assume that the identity of the fresh variable introduced in the derivation $b \Rightarrow_D^\alpha t \mid C$ for a particular **callvirt** instruction is a function of the node α and the position of the **callvirt** within the method body. Hence, if there are two derivations $b \Rightarrow_D^{(c::sig,D)} t \mid C$ and $b \Rightarrow_{D'}^{(c::sig,D')} t' \mid C'$, the two variables for a particular **callvirt** are equal just if $D = D'$.

Constraint Satisfaction and Flows. The table below represents the outcome of our analysis by a *flow*, a structure

$(\mathcal{N},\mathcal{U},\mathcal{M})$. The finite sets $\mathcal{N}$ and $\mathcal{U}$ represent all reachable nodes. The valuation function $\mathcal{M}$ fixes an abstract value $\mathcal{M}(t) \subseteq \mathit{Type}^\sharp$ for each symbolic value t. The predicate $(\mathcal{N},\mathcal{U},\mathcal{M}) \models C$ means that the structure $(\mathcal{N},\mathcal{U},\mathcal{M})$ satisfies the constraint C.

Control Flow Associated with $\mathcal{E}_{|\,Trusted}$: $(\mathcal{N},\mathcal{U},\mathcal{M})$

A *control flow* is a triple $(\mathcal{N},\mathcal{U},\mathcal{M})$ where

- $\mathcal{N} = \{\alpha_1, \ldots, \alpha_n\}$ is a set of trusted nodes.
- $\mathcal{U} = \{(\star::-, D_1), \ldots, (\star::-, D_m)\}$ is a set of untrusted nodes, with $D_i \subseteq S_\star$ for $i = 1..m$.
- $\mathcal{M}$ maps values t to sets of abstract types $\mathcal{M}(t) \subseteq \mathit{Type}^\sharp$, with $\mathcal{M}(\{A_1^\sharp, \ldots, A_n^\sharp\}) \triangleq \{A_1^\sharp, \ldots, A_n^\sharp\}$.

We let $\mathit{callee}(c::\mathit{sig}, D) \triangleq (d::\mathit{sig}, \mathit{statics}(d) \cap D)$ where $(d, b) = \mathit{methods}(c)(\mathit{sig})$.

We define a predicate $(\mathcal{N},\mathcal{U},\mathcal{M}) \models C$ by induction on C:

- $(\mathcal{N},\mathcal{U},\mathcal{M}) \models \mathbf{T}$.
- $(\mathcal{N},\mathcal{U},\mathcal{M}) \models t \subseteq t'$ when $\mathcal{M}(t) \subseteq \mathcal{M}(t')$.
- $(\mathcal{N},\mathcal{U},\mathcal{M}) \models C \wedge C'$ when $(\mathcal{N},\mathcal{U},\mathcal{M}) \models C$ and $(\mathcal{N},\mathcal{U},\mathcal{M}) \models C'$.
- $(\mathcal{N},\mathcal{U},\mathcal{M}) \models \mathit{VCALL}((c::\mathit{sig}, D), t_0, \ldots, t_n, \lambda)$ with $\mathit{sig} = B\ \ell(A_1, \ldots, A_n)$ when, for all $d \in Trusted$, we have:
  1. If **class** $d \in \mathcal{M}(t_0)$ with $\alpha = \mathit{callee}(d::\mathit{sig}, D)$, then $\alpha \in \mathcal{N}$ and $(\mathcal{N},\mathcal{U},\mathcal{M}) \models \bigwedge_{i \in 0..n} t_i \subseteq \alpha.i \wedge \alpha.\mathit{result} \subseteq \lambda$.
  2. If **class** $\star \in \mathcal{M}(t_0)$, then
     (a) if d *inherits* c with $\alpha = \mathit{callee}(d::\mathit{sig}, D)$, then $\alpha \in \mathcal{N}$ and $(\mathcal{N},\mathcal{U},\mathcal{M}) \models \bigwedge_{i \in 0..n} t_i \subseteq \alpha.i \wedge \alpha.\mathit{result} \subseteq \lambda$.
     (b) if $c::\mathit{sig} \in \mathit{Virtual}$, then $(\star::-, D \cap S_\star) \in \mathcal{U}$ and $(\mathcal{N},\mathcal{U},\mathcal{M}) \models \mathit{sub}^\sharp(B) \subseteq \lambda$.

A *correct flow* is a flow $(\mathcal{N},\mathcal{U},\mathcal{M})$ such that:

1. $(\star::-, S_\star) \in \mathcal{U}$.
2. If $(\star::-, D) \in \mathcal{U}$ and $D \subseteq D' \subseteq S_\star$, then $(\star::-, D') \in \mathcal{U}$.
3. For all $(\star::-, D) \in \mathcal{U}$ and $c::\mathit{sig} \in \mathit{Public}$ with $\mathit{sig} = B\ \ell(A_1, \ldots, A_n)$), let $\alpha = \mathit{callee}(c::\mathit{sig}, D)$ and $A_0 = \mathbf{class}\ c$. We have $\alpha \in \mathcal{N}$ and, for $i \in 0..n$, $(\mathcal{N},\mathcal{U},\mathcal{M}) \models \mathit{sub}^\sharp(A_i) \subseteq \alpha.i$
4. For all $\alpha = (c::\mathit{sig}, D) \in \mathcal{N}$ with $\mathit{methods}(c)(\mathit{sig}) = (c, b)$ and $b \Rightarrow_D^\alpha t \mid C$, we have $(\mathcal{N},\mathcal{U},\mathcal{M}) \models t \subseteq \alpha.\mathit{result} \wedge C$.

In the satisfaction rule for VCALL, the set $\mathcal{M}(t_0)$ ranges over the (abstract) dynamic types for the target object. Condition 1 deals with trusted types: d ranges over trusted classes and *callee* yields the nodes α corresponding to their implementation of the method. These nodes must be analysed and meet constraints generated for the arguments and results of the call. Condition 2 deals with untrusted types, if any. Untrusted classes can inherit implementations from trusted classes d, and the corresponding nodes α must be analysed, with the same constraints as above (Condition 2a). Besides, if the method is virtual, untrusted classes can also provide their own implementations; the node $(\star::-, D \cap S_\star)$ must be analysed, and a constraint reflects that these implementations can return any type-safe value (Condition 2b).

Finally, we define a flow to be *correct* to mean that all abstract untrusted code (Points 1 and 2) and all public methods which may be called with any type-safe value (Point 3) are part of the analysis, and that all constraints generated for the code of each node in $\mathcal{N}$ are satisfied (Point 4).

Intuitively, a correct flow provides an upper bound on all possible control paths through a trusted library composed with any untrusted code. The following theorem formalises this intuition. An extended version of the paper contains the proof [4].

Theorem 1 (Runtime Reachability) *Let $\mathcal{E}$ be a partially-trusted environment. Let $(\mathcal{N},\mathcal{U},\mathcal{M})$ be a correct flow for $\mathcal{E}_{|\,Trusted}$. If $M \in \mathit{Methods}_{|\,Trusted}$ is reachable from Untrusted, then $(M, D) \in \mathcal{N}$ for some D.*

In order to benefit from the theorem, we can effectively compute the least correct control flow by fixpoint iteration. The existence and computability of a fixpoint follows from standard results of constraint solving stating that a (finite) set of monotonic constraints defined over a finite lattice admits a least solution which can be computed by fixpoint iteration (see for instance [18]). For a given $\mathcal{E}_{|\,Trusted}$, we use a lattice of control flows obtained as the product of the lattice of sets of trusted nodes, the lattice of sets of untrusted nodes and the lattice of valuation functions ordered pointwise.

Example Flows. To illustrate our definitions and the theorem, we provide correct flows for the example libraries *File* and *CFile* given in Section 3. We assume the following: $\mathit{statics}(\mathit{File}) = \mathit{statics}(\mathit{CFile}) = \mathit{statics}(\mathit{Win32}) = S_t$; $\mathit{Win32::Delete}, \mathit{Win32::.ctor} \notin \mathit{Public}$; $\{\mathit{FilePermission}\} \cap S_\star = \varnothing$; and that no method of $File$, $CFile$, $Win32$ is in $\mathit{Virtual}$. We let $D = S_\star \cup (\{\mathit{FilePermission}\} \cap S_t)$.

Analysis for $Trusted = \{File\}$

The minimal correct flow $(\mathcal{N}_1, \mathcal{U}_1, \mathcal{M}_1)$ has four nodes

$$\begin{aligned} \mathcal{N}_1 &= \left\{ \begin{array}{l} (File::Delete, S_\star), (File::Backup, S_\star), \\ (File::Cleanup, S_\star) \end{array} \right\} \\ \mathcal{U}_1 &= \{(\star::-, S_\star)\} \end{aligned}$$

Taking for instance $\alpha = (File::Delete, S_\star)$, we have:

$$\begin{array}{rcl} \mathcal{M}_1(\alpha.0) & = & \{File, \textbf{class}\ \star\} \\ \mathcal{M}_1(\alpha.1) & = & \{\textbf{string}\} \\ \mathcal{M}_1(\alpha.result) & = & \{\textbf{void}\} \end{array}$$

Analysis for $Trusted = \{File, CFile\}$

The minimal correct flow $(\mathcal{N}_2, \mathcal{U}_2, \mathcal{M}_2)$ has ten nodes

$$\mathcal{N}_2 = \mathcal{N}_1 \cup \left\{ \begin{array}{l} (CFile::Commit, S_\star), (File::Delete, D), \\ (File::Backup, D), (File::Cleanup, D), \\ (Win32::.ctor, D), (Win32::Delete, D) \end{array} \right\}$$

$$\mathcal{U}_2 = \mathcal{U}_1 = \{(\star::-, S_\star)\}$$

This flow corresponds to Figure 1. As a corollary of Theorem 1, we obtain that there is no path from untrusted code to *Win32::Delete* using only *File*, but that there is potentially such a path if additionally *CFile* is present.

6. Implementation

We survey the design and implementation of our tool. This section describes the refinement of the analysis from BIL-SEC to the CLR. Section 7 shows how to apply the analysis to identify typical security defects and discusses experimental results.

Our implementation is written in Objective Caml. It relies on the AbstractIL toolkit to read and manipulate typed IL assemblies [23]. It has a simple command-line interface for interactive queries, which can be evaluated against precomputed call graphs for large libraries. Including various parsers for configuration files and permissions, the source code has 16Klocs. To the best of our knowledge, this is the first global control flow analysis for the CLR that deals with virtual calls and inheritance.

The CLR is considerably more complex than BIL-SEC. The main practical difficulties for generating the graph stem from the size of the standard libraries (providing thousands of classes to unknown code) and the need to give a precise account of the numerous features of the CLR related to security. As a side benefit, we found several ambiguities and defects in the process of reflecting the semantics of these features.

Known and Unknown Code. Starting from the target input libraries, we recursively load any assembly mentioned in a type reference. Hence, as in BIL-SEC, the known classes and interfaces do not statically depend on unknown code.

For each known class, we then simulate the rules of inheritance to complete the class hierarchy. This completion is necessary to accurately simulate the resolution of virtual calls whenever dynamic types declared in unknown code may flow to call sites in known code. In BIL-SEC, we use a single symbolic class, $\star$, to simulate all untrusted subclasses. In the implementation, we consider access modifiers and **inheritancedemand**s, and use a distinct symbolic class to simulate the subclasses of every trusted class. We do not further instantiate these unknown classes; since unknown implementations may either inherit or override a given method, we consider both cases during method resolution as we propagate virtual calls.

To begin with, we create an initial node, $(\star::-, S_\star)$ in BIL-SEC, and we simulate all the possible calls to methods directly callable by unknown code. We take into account the semantics of the CLR, including scoping rules, access modifiers, inheritance rules, and declarative security actions such as **linkdemand** and **inheritancedemand**. Unknown code may only operate on objects accessible at runtime, for instance using a public constructor, obtained as a parameter in a callback, or reading a protected field in a superclass. Accordingly, a variable represents all values currently available to unknown code, and is used to simulate its operations (including its virtual calls).

Representing Permissions and their Operations. Runtime permissions have a complex structure, so we rely on two different approximations, with different trade-offs between precision and complexity:

- A fine-grain representation reflects most of the details available in security metadata, including constant parameters, and is also locally inferred in code using an auxiliary dataflow analysis. (This dataflow is simple in practice: permission values used in asserts and demands are typically newly-constructed objects or objects read from constant fields; the remaining cases are handled in an ad hoc manner.)

  The domain for these permissions is a nested product of lattices for independent boolean flags, for multilevel permissions, and for permissions with string parameters. This domain is built from a structural description of the twenty or so permission classes appearing in standard libraries. We lose precision for string parameters, using for instance a single abstract value for representing "read access for some specific (unknown) file".

- A potentially coarser representation is used for the permission contexts D in the global analysis. To obtain maximal precision, the analysis can be carried out using the fine-grained representation. However, to trade precision for efficiency, we have devised flexible abstractions. The coarsest one computes over a domain of two values *UntrustedCodePermissions* and *AllPermissions*. Since most security queries involve one or two permission classes, we can also adapt this representation to precisely keep track of these target permissions, and abstract away all other permissions.

Even the computation of static permissions requires some care to reflect the semantics of the CLR; it involves several parsers to extract the security policy from metadata and configuration files. During the analysis, we intercept calls to permission libraries, such as *System.Security.CodeAccessPermission::Demand*, detect declarative security attributes, and transform them into abstract security actions. (In BIL-SEC, these correspond to the synthetic instructions **assert** and **demand**.) In addition, we support additional security actions such as *RevertAssert*, *RevertAll*, *Deny*, and *PermitOnly*.

Constraint Generation and Resolution. Our analysis interleaves the generation and resolution of constraints, until a fixpoint is reached.

Intra-method IL constraint generation is essentially a refinement of the type checking algorithm, with type inference for the (symbolic) stack. Each block of code is executed at most once for each reachable value of D, its effective dynamic permissions, and yields a set of constraints. The analysis also builds the local control flow between these blocks, and connects them using additional constraints on their entry- and exit-stacks. The effect of security actions is immediately simulated during code analysis: the outcome of a demand is determined by comparing the demanded permissions to the dynamic permissions D of the method; the rest of the block is analysed only if this comparison potentially succeeds. In contrast with BIL-SEC, however, an assert does not immediately affect D—the asserted permission is taken into account to compute D for any call within the scope of the assert.

Our constraints consist of inclusions, equalities, primitive operations such as boxing and run-time type checks, and dynamic constraints for virtual calls. Constraint resolution may update (or merge) variables. In addition, dynamic constraints may trigger the analysis of additional blocks of code, leading to the generation of additional constraints. Our constraint solver is rather simple, and keeps selecting and propagating unsatisfied constraints, until a fixpoint is reached. The runtime and memory requirements for analysing standard libraries required careful performance optimisations on internal data structures. The resulting graph provides a sound approximation of reachability and dynamic permissions for known code—since nodes are created on demand, only method implementations that may be reachable from unknown code are represented, at their dynamic permission contexts.

Limitations. Pragmatically, to scale up to large libraries, we make coarse approximations for features that seldom occur. Although we cover all instructions, we do not deal with certain primitive features, such as reflection and some operations on delegates. We assume that calls to native code and unverifiable IL code preserve runtime type-safety. Finally, the analysis is useful for security only when unknown code has few permissions, so we assume that unknown code never gains privileges to emit new IL code or bypass type checking.

7. From the Call Graph to Security Defects

Permission-sensitive call graphs provide non-trivial, useful information to aid review of the security of libraries: for each potential call stack, we have a corresponding path in the graph. Still, interpreting the raw results of the analysis is delicate, and often requires human judgement. (Indeed, permission classes define a data structure, rather than a high-level access control policy, which is usually implicit [9, 1].)

We run queries on the call graph to extract a global view of the usage of permissions for access control. The queries are motivated by typical error patterns observed in the development of libraries for the CLR. (See also [4] for a collection of small, synthetic examples in C$^\sharp$ that illustrate these patterns.) In contrast with other works [14], we do not rely on a formal logic for expressing classes of queries.

Reachability Queries. The call graph relates all method implementations that may be called at runtime. In particular, for each path in the graph, we can collect the sequence of security actions (**demand**, **assert**, **linkdemand**,…) performed along this path. For any identified, privileged operation located in the code (such as a native call to a system library) that is reachable in the graph, the tool reports a collection of short, "exemplary" paths from unknown code to the privileged operation. Each such path represents a (possibly infinite) equivalence class of code paths at runtime, for a notion of equivalence that initially relates paths with the same interleaving of security actions, and that can be refined to investigate unexpected cases.

For example, we may report minimal paths with no security actions from unknown code to system calls analogous to *Win32::Delete* in the example of Section 3. More broadly, we may report a minimal path for each potential sequence of security actions leading to this system call. For instance, for file deletion, we have paths with a single demand on some *FileIOPermission*, with a single demand on *IsolatedStoragePermission* followed by an assert on *FileIOPermission*, and so on. In practice, even for large libraries, we observe a small number of different cases, due to the relatively small number of dynamic security actions, so in many cases all identified classes can be reviewed by hand.

This information is useful when adding new trusted code. By comparing the old graph with the new one, we can observe methods that have become reachable. In our example, the addition of the naive, trusted library causes the appearance of native methods *Win32::.ctor* and *Win32::Delete*. It indicates that a security invariant may have been broken. Indeed, these two methods are reachable because of the $\mathbf{assert}$ in *CFile::Commit*.

Finding a Purpose to Security Actions. Since they affect functionality and performance, as well as security, each dynamic action on permissions should have a clear goal, which we attempt to infer from the call graph. For every reachable $\mathbf{assert}\ P$, we check that there is a node where the $\mathbf{assert}$ affects the dynamic permissions (that is, with dynamic permissions D and possibly $P \notin D$), and explore paths starting from the $\mathbf{assert}$ with $P \in D$ to identify at least one sensitive operation protected by the assert, and otherwise flag the $\mathbf{assert}$ for review. (In our example, we check that the $\mathbf{assert}$ in *CFile::Commit* enables the $\mathbf{demand}$ in *File::Delete*.) We also check that every $\mathbf{demand}$ is fallible, and try to find at least one protected operation.

Link-demands, and Other Optimisations. A common performance optimisation is to substitute $\mathbf{linkdemand}$s for $\mathbf{demand}$s, in order to avoid the run-time cost of stack inspection. Since only the immediate caller's permissions are now checked, this transformation is potentially unsafe. Accordingly, for each $\mathbf{linkdemand}$ in the code, our tool verifies whether the corresponding dynamic permissions would suffice to pass a $\mathbf{demand}$ for the same permission, and otherwise reports additional paths from unknown code to the protected method. (In our running example, one may substitute a $\mathbf{linkdemand}$ for the $\mathbf{demand}$ in *File::Delete*. This creates a dangerous path from unknown code to *Win32::Delete* via *File::Cleanup*, which is reported by this query.)

Similarly, we can use the call graph to determine whether ordinary, interprocedural code transformations such as code inlining or tail-call eliminations are correct.

Additional Flows. Stack inspection automatically keeps track of nested calls, but ignores more complex control flows (callbacks, exceptions), and any data flows. (See [1] for a discussion of this issue, including problematic programming examples in C$^\sharp$.) Once we have identified parts of the graph protected by permissions, we can use queries that check for local, common risks with these flows, such as the escape of private mutable data.

As an example, we implemented a query that reports (potential) callbacks from libraries to unknown code. Although we observe a large proportion of virtual calls in libraries that might call back to unknown code (from 5% to 10%), only a few of them occur in code that executes with elevated dynamic permission, and most of those call the same method references, so their manual review turns out to be feasible and interesting. These callbacks may still be safe, since dynamic permissions are lowered during the call, but there is a risk if the caller neglects to validate the result, or any shared mutable data. See [1] for examples and discussion of this error pattern.

Checking Uniformity: Towards Policy Extraction. For a given protected operation, security checks present on control paths should implement the same (implicit) access control discipline. Conversely, if all paths except those through a new library demand a particular permission, this one path should be flagged as a risk. We implemented a simple model extraction and refinement tool, which enables us to detect sensitive operations, and to systematically assess every security action. Although our current model is not expressive enough to capture the usage of all permissions, it suffices to restrict the scope of reviews to complex or unusual patterns.

Experimental Results for the CLR libraries. We tried our tool against the standard libraries of the CLR. As an example in the .NET Framework v1.1, the construction of the call graph for *System.Windows.Forms.dll* involves seven additional assemblies, including the core libraries *mscorlib.dll* and *System.dll*, for a total of 4,283 trusted types (including interfaces) and 10,080 methods directly callable from unknown code loaded with the *Internet* set of permissions. (This set contains the few permissions assigned by default to downloaded applets.) The completion generates 987 additional types for unknown code. On a machine with a Pentium M 1.6 GHz processor, the construction takes 40 minutes and 850MB, and involves 2,161,660 constraints between 338,341 variables. The code uses 25 permission classes. It reaches 742 demands and 403 asserts. The resulting graph has 43,817 trusted nodes and 410,759 edges.

We list a few kinds of defects encountered as we tested our tool on libraries: Two calls to the same sensitive method in different libraries are guarded by demands with different permission parameters. The scope of an assert or a demand is too large; for example, we found conditionals interleaved between a Demand and its sensitive operation, leading to unnecessary (and undocumented) security exceptions. In a few cases, such as *System.IO.Directory::GetCurrentDirectory*, the demanded permission depends on the result of a sensitive call; these cases need some careful review, to check (for instance) that all control flows that leak the result of the call are effectively guarded. More commonly, we found discrepancies between the documentation and the potential demands in nested calls. Operations on permissions in these libraries have been carefully reviewed by hand, at a considerable cost, so we expect to find more defects as we apply our tool to new libraries.

8. Related Work

There is by now a large literature on stack inspection, so for the sake of brevity this section only discusses related work on static analyses of stack inspection, rather than research primarily focused on its design and implementation [24], its limitations and formal semantics [9, 7], or on alternative mechanisms [17, 8, 1].

Pottier, Skalka, and Smith [21] develop a type system for a λ-calculus with stack inspection that statically ensures that, in any well-typed term, no demand fails.

Banerjee and Naumann [2] give an analysis for a Java-like language equipped with stack inspection to determine whether two classes with the same interface are representation independent, that is, if a difference in their private data representations is detectable by any other component. Nitta, Takata, and Seki [20] analyse the complexity of deciding whether a whole program satisfies a security property.

Jensen, Le Métayer, and Thorn [14] introduce a graph model for programs with stack inspection. They can verify whether all reachable stacks satisfy a formula expressed in linear temporal logic. Based on the same model, Besson *et al.* [5] infer a weakest precondition that ensures that a security violation cannot occur in a library abstracted by its call graph. However, they do not explain how to obtain a graph that safely approximates unknown code.

Koved, Pistoia, and Kershenbaum [15] provide an algorithm and an implementation to analyse permissions for Java. Their analysis is context-sensitive, flow-sensitive and they also use data-flow on permission objects to improve precision. In contrast with the present work, it aims at determining a set of permissions that are required to run a given program.

Bartoletti, Degano, and Ferrari [3] provide an analysis that inputs a control flow graph for a program and calculates a safe approximation of dynamic permissions at each program point. In contrast with the present work, their analysis is only partly open, and does not account for virtual calls towards unknown code.

In general, control-flow analyses have been thoroughly studied, and provide a useful framework for developing more specific static analyses such as ours. For instance, Grove and Chambers detail general algorithms and datastructures to build a context-sensitive call graph [12]. In their analysis to assess test coverage for libraries, Rountev *et al.* [22] model the set of unknown environments by generating a single, most-general program. Their representation of unknown code accounts for unknown callers, but not for unknown subclasses, which is important in our setting to detect potential callbacks.

9. Conclusions and Future Work

We implemented a control-flow-based analysis for reviewing the security of libraries for the CLR. We also established a correctness result for BIL-SEC, a small but significant subset of IL. To the best of our knowledge, the idea of a permission-sensitive analysis of stack inspection with static representations of unknown, potentially hostile code is new, as is the catalogue of queries in Section 7 to help code reviews for security. Our main theoretical result, Theorem 1, shows our flow analysis can prove the unreachability of a particular sensitive method in the presence of any arbitrary hostile code; we are aware of no such prior results for formal models of stack inspection, although there are some analogous results for unrelated formalisms such as the ambient calculus [19] and, more recently, a model of firewalls for Javacard applets [13].

We are working to improve the performance of our tool, as well as to develop our catalogue of queries. It would be interesting (and hard) to develop an analysis that is more sensitive to other parts of the context, such as allocation points for objects, or that is more precise for some aspects of IL, such as exception handling and concurrency. In any case, we believe our tool can be very helpful for programmers, and especially library writers concerned with the security implications of their code.

Acknowledgements. Andrew Kennedy and Don Syme helped us to model the semantics of the CLR. Toshiyuki Maeda implemented some of the security queries on call graphs. Jean-Jacques Lévy provided comments on a draft.

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *Network and Distributed System Symposium (NDSS'03)*, pages 107–121. Internet Society, February 2003.

[2] A. Banerjee and D. Naumann. Representation independence, confinement, and access control. In *29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 166–277, Jan. 2002.

[3] M. Bartoletti, P. Degano, and G. Ferrari. Static analysis for stack inspection. In *ConCoord: International Workshop on Concurrency and Coordination*, volume 54 of *ENTCS*. Elsevier, 2001.

[4] F. Besson, T. Blanc, C. Fournet, and A. D. Gordon. From stack inspection to access control: A security analysis for libraries. Long version of this paper, available from `http://research.microsoft.com/~fournet/pwp`, 2004.

[5] F. Besson, T. de Grenier de Latour, and T. Jensen. Secure calling contexts for stack inspection. In *4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 76–87, 2002.

[6] D. Box. *Essential .NET Volume I: The Common Language Runtime*. Addison Wesley, 2002.

[7] J. Clements and M. Felleisen. A tail-recursive semantics for stack inspections. In *Programming Languages and Systems (ESOP 2003)*, volume 2618 of *LNCS*, pages 22–37. Springer-Verlag, 2003.

[8] Ú. Erlingsson and F. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255. IEEE Computer Society Press, 2000.

[9] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(3):360–399, 2003. An extended abstract appears in POPL'02.

[10] L. Gong. *Inside Java™ 2 Platform Security*. Addison Wesley, 1999.

[11] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 248–260, Jan. 2001.

[12] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):686–746, nov 2001.

[13] R. R. Hansen. A hardest attacker for leaking references. In D. Schmidt, editor, *Programming Languages and Systems (ESOP 2004)*, volume 2986 of *LNCS*, pages 310–324. Springer-Verlag, March 2004.

[14] T. Jensen, D. L. Metayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 89–103. IEEE Computer Society Press, 1999.

[15] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In *17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 359–372, 2002.

[16] S. Lange, B. LaMacchia, M. Lyons, R. Martin, B. Pratt, and G. Singleton. *.NET Framework Security*. Addison Wesley, 2002.

[17] A. C. Myers. JFlow: Practical, mostly-static information flow control. In *26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 228–241, 1999.

[18] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer, 1999.

[19] F. Nielson, H. R. Nielson, R. R. Hansen, and J. G. Jensen. Validating firewalls in mobile ambients. In *Concurrency Theory (Concur'99)*, volume 1664 of *LNCS*, pages 463–477. Springer-Verlag, 1999.

[20] N. Nitta, Y. Takata, and H. Seki. An efficient security verification method for programs with stack inspection. In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 68–77, 2001.

[21] F. Pottier, C. Skalka, and S. Smith. A systematic approach to access control. In *Programming Languages and Systems (ESOP 2001)*, volume 2028 of *LNCS*, pages 30–45. Springer-Verlag, 2001.

[22] A. Rountev, A. Milanova, and B. G. Ryder. Fragment class analysis for testing of polymorphism in java software. In *25th international conference on Software engineering*, pages 210–220. IEEE Computer Society, 2003.

[23] D. Syme. ILX: Extending the .NET common IL for functional language interoperability, Sept. 2001. `http://research.microsoft.com/projects/ilx/`.

[24] D. S. Wallach, A. W. Appel, and E. W. Felten. SAFKASI: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, 2000.

A. BIL-SEC: Additional Definitions

Subtype Relation: $A <: B$

(Sub Refl) $$\frac{}{A <: A}$$

(Sub Class) $$\frac{c\ \mathit{inherits}\ c'}{\mathbf{class}\ c <: \mathbf{class}\ c'}$$

Axioms for $\mathit{inherits}$

$c\ \mathit{inherits}\ c$	(Hi Refl)
$c\ \mathit{inherits}\ c' \wedge c'\ \mathit{inherits}\ c'' \Rightarrow c\ \mathit{inherits}\ c''$	(Hi Trans)
$c\ \mathit{inherits}\ c' \wedge c'\ \mathit{inherits}\ c \Rightarrow c = c'$	(Hi Antisymm)
$c\ \mathit{inherits}\ \mathbf{System.Object}$	(Hi Root)
$c\ \mathit{inherits}\ d \wedge f \in \mathit{dom}(\mathit{fields}(d)) \Rightarrow f \in \mathit{dom}(\mathit{fields}(c)) \wedge \mathit{fields}(c)(f) = \mathit{fields}(d)(f)$	(Hi *fields*)
$c\ \mathit{inherits}\ d \Rightarrow \mathit{dom}(\mathit{methods}(d)) \subseteq \mathit{dom}(\mathit{methods}(c))$	(Hi *methods*)
$\mathit{methods}(c)(\mathit{sig}) = (d, b) \Rightarrow d\ \mathit{inherits}\ c \wedge \mathit{methods}(d)(\mathit{sig}) = (d, b)$	(Hi Meth Impl)

Evaluating Method Bodies. A *result* is an outcome of evaluating an expression. A result can be void, an integer, or an object reference, a pointer into the heap.

References, Results:

p, q	heap reference	
$u, v \in \mathit{Result} ::=$	result	
$\mathbf{0}$		void
$i4$		integer
p		object reference

A *store* consists of a stack s plus a heap h. A *heap* is a finite map from references to boxed objects, which takes the form $c[f_i \mapsto u_i\ {}^{i \in 1..n}]$, where c is the class of the object, $f_1, \ldots, f_n$ are its field names, and $u_1, \ldots, u_n$ are the contents of the fields. A *stack* consists of a sequence of frames, each of which represents a method invocation. A *frame* $.\mathbf{args}(u_0, \ldots, u_n)$ consists of u_0, a reference to self, plus the arguments $u_1, \ldots, u_n$. (There are no local variables, but note that arguments are mutable.)

Memory Model:

$o ::= c[f_i \mapsto u_i\ {}^{i \in 1..n}]$	boxed object
$h ::= p_i \mapsto o_i\ {}^{i \in 1..n}$	heap
$\mathit{fr} ::= .\mathbf{args}(u_0, \ldots, u_n)$	frame: vector of arguments
$s ::= \mathit{fr}_1 \cdots \mathit{fr}_n$	stack (grows left to right)
$\sigma ::= (h, s)$	store

Our operational semantics appeals to the following functions for accessing and mutating the store, in particular, the heap component. (In future work, we intend to include in BIL-SEC the stack pointers of BIL, in which case these functions would need to access and mutate the stack as well as the heap.)

Auxiliary Partial Functions for Accessing the Heap:

$\mathit{dynClass}(\sigma, p)$	lookup dynamic class of p in store σ
$\mathit{lookup}(\sigma, p.f)$	lookup field $p.f$ in store σ
$\mathit{update}(\sigma, p.f, v')$	update store field σ at $p.f$ with result v'

if $h = p \mapsto c[f_i \mapsto u_i\ ^{i \in 1..n}], h'$ and $j \in 1..n$

$\mathit{dynClass}((h, s), p) = c$

$\mathit{lookup}((h, s), p.f_j) = u_j$

$\mathit{update}((h, s), p.f_j, v') =$

$\quad ((p \mapsto c[f_j \mapsto v', f_i \mapsto u_i\ ^{i \in (1..n)-\{j\}}], h'), s)$

As in Fournet and Gordon's formulation of stack inspection [9], evaluation of an expression depends on two permission sets, the static permissions S, and the dynamic permissions D, with $D \subseteq S$. The static permissions are those associated with the current method, and the dynamic permissions are those effectively available. We formalise evaluation by a judgement of the following form:[1]

Evaluation Judgement:

$\sigma \vdash b \leadsto^S_D v \cdot \sigma'$	given σ and dynamic permissions D, body b with static permissions S returns v, leaving σ'

Evaluation Rules for Control Flow:

(Eval $\mathbf{ldc}$)

$$\frac{}{\sigma \vdash \mathbf{ldc.i4}\ \mathit{i4} \leadsto^S_D \mathit{i4} \cdot \sigma}$$

(Eval Seq)

$$\frac{\sigma \vdash a \leadsto^S_D u \cdot \sigma' \quad \sigma' \vdash b \leadsto^S_D v \cdot \sigma''}{\sigma \vdash a\, b \leadsto^S_D v \cdot \sigma''}$$

(Eval $\mathbf{demand}$)

$$\frac{\sigma \vdash a_{P \in D} \leadsto^S_D v \cdot \sigma'}{\sigma \vdash \mathbf{demand}\ P\ a_{true}\ \mathbf{else}\ a_{false} \leadsto^S_D v \cdot \sigma'}$$

(Eval $\mathbf{assert}$)

$$\frac{\sigma \vdash a \leadsto^S_{D \cup (\{P\} \cap S)} v \cdot \sigma'}{\sigma \vdash \mathbf{assert}\ P\ a \leadsto^S_D v \cdot \sigma'}$$

The expression $\mathbf{ldc.i4}\ \mathit{i4}$ evaluates to the integer $\mathit{i4}$. The expression $a\ b$ evaluates a to a result, expected to be void. The result of the whole expression is then the result of evaluating b. The expression $\mathbf{demand}\ P\ a_{true}\ \mathbf{else}\ a_{false}$ evaluates either a_{true} or a_{false}, depending on whether P is one of the dynamic permissions. The expression $\mathbf{assert}\,P\,a$ intersects $\{P\}$ with the static permissions, adds the outcome to the dynamic permissions, and evaluates a.

Evaluation Rules for Arguments:

(Eval $\mathbf{ldarg}$)

$$\frac{\sigma = (h, s.\mathbf{args}(u_0, \ldots, u_n)) \quad j \in 0..n}{\sigma \vdash \mathbf{ldarg}\ j \leadsto^S_D u_j \cdot \sigma}$$

(Eval $\mathbf{starg}$)

$$\frac{\sigma \vdash a \leadsto^S_D u'_j \cdot (h, s.\mathbf{args}(u_0, \ldots, u_j, \ldots, u_n)) \quad j \in 0..n}{\sigma \vdash a\ \mathbf{starg}\ j \leadsto^S_D \mathbf{0} \cdot (h, s.\mathbf{args}(u_0, \ldots, u'_j, \ldots, u_n))}$$

The expression $\mathbf{ldarg}\,j$ returns argument j of the current stack frame. The expression $a\,\mathbf{starg}\,i$ evaluates a, stores the result in argument i in the current stack frame, then returns void.

Evaluation Rules for Objects:

(Eval $\mathbf{newobj}$)

$$\frac{\begin{array}{l}\left(\sigma_i \vdash a_i \leadsto^S_D v_i \cdot \sigma_{i+1}\right)^{i \in 1..n} \\ \sigma_{n+1} = (h, s) \quad \mathit{fields}(c) = f_i \mapsto A_i\ ^{i \in 1..n} \\ p \notin \mathit{dom}(h) \quad h' = h, p \mapsto c[f_i \mapsto v_i\ ^{i \in 1..n}]\end{array}}{\sigma_1 \vdash a_1\ \cdots\ a_n\ \mathbf{newobj}\ c{::}\mathit{ksig} \leadsto^S_D p \cdot (h', s)}$$

(Eval $\mathbf{callvirt}$)

$$\frac{\begin{array}{l}\left(\sigma_i \vdash a_i \leadsto^S_D v_i \cdot \sigma_{i+1}\right)^{i \in 0..n} \\ \sigma_{n+1} = (h, s) \quad c' = \mathit{dynClass}(\sigma_{n+1}, v_0) \\ \mathit{methods}(c')(\mathit{sig}) = d, b \quad S_d = \mathit{statics}(d) \\ (h, s.\mathbf{args}(v_0, v_1, \ldots, v_n)) \vdash b \leadsto^{S_d}_{D \cap S_d} v' \cdot (h', s'\,\mathit{fr}')\end{array}}{\sigma_0 \vdash a_0\ a_1\ \cdots\ a_n\ \mathbf{callvirt}\ c{::}\mathit{sig} \leadsto^S_D v' \cdot (h', s')}$$

(Eval $\mathbf{ldfld}$)

$$\frac{\sigma \vdash a \leadsto^S_D p \cdot \sigma'}{\sigma \vdash a\ \mathbf{ldfld}\ A\ c{::}f \leadsto^S_D \mathit{lookup}(\sigma', p.f) \cdot \sigma'}$$

(Eval $\mathbf{stfld}$)

$$\frac{\sigma \vdash a \leadsto^S_D p \cdot \sigma' \quad \sigma' \vdash b \leadsto^S_D v \cdot \sigma''}{\sigma \vdash a\, b\ \mathbf{stfld}\ A\ c{::}f \leadsto^S_D \mathbf{0} \cdot \mathit{update}(\sigma'', p.f, v)}$$

The expression $a_1\ \cdots\ a_n\ \mathbf{newobj}\ K$, where K is the constructor for a class c, heap allocates an object whose fields contain the results of evaluating $a_1, \ldots, a_n$, and returns the new reference.

The expression $a_0\, a_1\ \cdots\ a_n\,\mathbf{callvirt}\,M$, where M refers to $B\ \ell(A_1, \ldots, A_n)$ in class c, evaluates a_0 to a reference to a boxed object of class c' (expected to inherit from c), retrieves the implementation superclass d and method body for signature $B\ \ell(A_1, \ldots, A_n)$ in dynamic class c', and returns the result of evaluating this method body in a new

1 In contrast with BIL and our implementation, our model BIL-SEC currently does not contain operations for parameters passed by reference to an entry on the stack (parameter keywords *out* and *ref* in C♯), so we don't need to mutate the stack s in depth during evaluation. Hence, we could simplify the evaluation judgement by passing only the heap and the top frame (h, fr) instead of the heap plus the stack $\sigma = (h, s)$.

stack frame whose argument vector consists of the reference to the boxed object (the self pointer) together with the results of $a_1, \ldots, a_n$. The new invocation runs with static permissions equal to $statics(d)$ where d is the implementation superclass, and with the current dynamic permissions adjusted by intersecting with $statics(d)$. The result of this evaluation is the store $(h', s'\,fr')$, where fr' is the final state of the new stack frame. Once evaluation of the method is complete, the stack is popped, to leave (h', s') as the final store. The expression a **ldfld** $A\ c{::}f$ evaluates a to an object reference, then returns field f of this object. The expression $a\,b$ **stfld** $A\,c{::}f$ evaluates a to a reference to an object, updates its field f with the result of evaluating b, and returns void.

Reachability. For a given execution environment, we define a notion of dynamic method reachability. Our main result concerns unreachability of sensitive methods.

Reachability

To every evaluation $\sigma \vdash b \rightsquigarrow^S_D v \cdot \sigma'$, we associate the (unique) derivation tree obtained from the evaluation rules. To each instance in the tree of the rule:

(Eval $\mathbf{callvirt}$)

$$\frac{\begin{array}{l}\left(\sigma_i \vdash a_i \rightsquigarrow^S_D v_i \cdot \sigma_{i+1}\right)^{i \in 0..n} \\ \sigma_{n+1} = (h, s) \quad c' = dynClass(\sigma_{n+1}, v_0) \\ methods(c')(sig) = d, b \quad S_d = statics(d) \\ (h, s.\mathbf{args}(v_0, v_1, \ldots, v_n)) \vdash b \rightsquigarrow^{S_d}_{D \cap S_d} v' \cdot (h', s'\,fr')\end{array}}{\sigma_0 \vdash a_0\ a_1\ \cdots\ a_n\ \mathbf{callvirt}\ c{::}sig \rightsquigarrow^S_D v' \cdot (h', s')}$$

we associate the label $M \triangleq d{::}sig$. The evaluation $\sigma \vdash b \rightsquigarrow^S_D v \cdot \sigma'$ reaches M when M is a label in its derivation tree.

M is *reachable* from $C \subseteq Class$ when an evaluation

$$(\epsilon, \epsilon) \vdash \begin{array}{l}\mathbf{newobj\ void}\ c{::}.\mathbf{ctor}() \\ \mathbf{callvirt\ void}\ c{::}l()\end{array} \rightsquigarrow^S_S v \cdot \sigma'$$

reaches M for some v, σ', and $c \in C$ with $methods(c)(\mathbf{void}\ l()) = c, b$ and $statics(c) = S$.

By labelling with $d{::}sig$, we mark the method implementations whose bodies are actually evaluated. A method implementation is reachable from C if there exists a body in C that directly or indirectly evaluates this implementation. The intent is to characterise the code that an attacker could trigger.

We refer to the long version of this paper for the adaptation of the typing rules and type-safety theorem of [11].