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

Tomasz Blanc
INRIA Rocquencourt

Cédric Fournet
Microsoft Research

Andrew D. Gordon
Microsoft Research

Draft, November 2003

Abstract

We present a new static analysis to help identify security defects in class libraries for runtimes, such as JVMs or the CLR, that rely on stack inspection for access control. Our tool inputs a set of class 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 defects. We describe the tool architecture, various examples of security queries, and a practical implementation that analyses large pre-existing libraries for the CLR. 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.

Contents

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 provide access control in the presence of potentially hostile "semi-trusted code", extensible systems such as the Java Virtual Machine (JVM) and the Common Language Runtime (CLR) provide sophisticated security mechanisms, including a stack inspection mechanism that can determine the run-time rights of each piece of code as a function of the stack [11, 5, 16]. Rights 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 rights.

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 architecture, implementation, and formalisation of a new static analysis tool to address these problems. Our tool analyses the use of runtime permissions in the CLR, with its existing mechanisms and libraries, but its principles seem applicable in other settings, such as the JVM. A significant novelty compared to previous program analyses is the need to check permissions for an open system, where some code is unknown at analysis-time. Section 2 reviews the CLR, surveys some of its security mechanisms, and introduces a running example using stack inspection.

The first stage of the tool, described in Section 3, 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. The call graph includes nodes for the unknown code as well as the input libraries, with multiple nodes for each piece of code that can be executed with different run-time permissions. Apart from these novelties, the construction is simple in principle—if not in detail, as our implementation handles the full CLR instruction set.

The second stage of the tool, described in Section 4, consists of evaluating queries on the graph to detect possible security defects, such as the reachability of dangerous methods, or the existence of unusual, possibly erroneous, paths in the graph. We explain a range of security properties that can be (partially) statically checked, and present experimental results from analysing substantial existing libraries such as *mscorlib*.

To provide a precise setting to describe our call graph construction, Section 5 defines a new formal model of stack inspection within the CLR. Our model, BIL-SEC, is a variation of Baby IL [12], 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 [20, 10].

We formalise our control flow analysis for BIL-SEC in Section 6, and state a correctness theorem. Suppose we have a call graph for a particular trusted library, and

then consider an arbitrary choice of partially trusted code to be loaded at runtime. Our main result states that, if there can be a sequence of calls starting from the partially trusted code and ending with a particular trusted method, then there is a corresponding path in the call graph. Hence, a query showing there is no such path in the call graph implies no dynamically loaded code can reach a particular trusted method.

2 Stack-Based Access Control (Review)

2.1 The CLR and its Intermediate Language

The CLR is a memory-managed, object-oriented computing platform [5]. An *assembly* is its unit of code deployment, typically a single file, containing a package of metadata plus actual implementation code. Metadata includes details of the class hierarchy, as well as security-related information such as a digital signature as evidence of origin, or 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, including 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.

2.2 Permissions and Stack Inspection

Code access rights are expressed using *permissions*, data represented by objects of particular classes at runtime. 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 and 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, with 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 **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 **assert** P to add P to the dynamic permissions, provided that P is one of its static permissions. By asserting P, the trusted code takes responsibility for any subsequent demands for P, until the completion of the current method. Such privilege elevations are dangerous, and deserve careful review.

2.3 An Example in BIL-SEC

To illustrate these access control features and the potential for code defects, we give three example classes below. 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*. The class *BadFile* is untrusted code; its static permissions do not include *FilePermission*, but nonetheless it can inherit from the public class *CFile*.

These classes also introduce BIL-SEC, the formal model of CLR access control we develop in Sections 5 and 6. The BIL-SEC syntax is very similar to standard IL assembler; a minor difference is that BIL-SEC has primitive instructions for **demand** and **assert** whilst in IL these are method calls. For the sake of brevity, we omit the source code, but we expect the stack-based assembler semantics should be fairly clear.

The three methods exposed by *File* guard access to the private *Win32*::*Delete* primitive by raising a security exception if called without *FilePermission*—in case of *Delete* and *Backup* directly via demands; in case of *Cleanup* indirectly via the call to *Delete*.

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

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

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 field *tempfile*. The untrusted method *BadFile* :: *DeleteAny* exploits this error. 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 means the same as private, except that derived classes 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.

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 link-demands and inheritance-demands, 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 errors that occur in practice, and our tool for exploring them.

3 Generating a Permission-Sensitive Call Graph

Access control in the CLR is an implicit, global safety invariant; its correctness may be compromised by programming errors disseminated through a large body of code. Empirically, permissions are asserted or demanded in relatively few methods appearing in libraries likely to be designed and tested independently, and most of those methods use permissions in a simple and uniform manner—for instance, the usage of permissions rarely depends on the data flow, and objects representing permissions at run-time are typically either constructed just before the demand or assert, or read from a constant static field for the class. During the development of libraries for the CLR, a large proportion of security defects involves permissions, but these defects often fall into a few simple patterns. (We suspect this is partly due to programmers confronting the security model of the CLR for the first time.)

This suggests that a coarse, specific analysis of code can help reviewing and improving the usage of permissions across libraries. This section describes the first stage of our analysis, a call graph construction, while Section 4 describes queries on the graph.

3.1 A Permission-Sensitive Call-Graph Analysis

In IL, virtual calls (and interface calls) induce a dependency between the control- and data-flows through the dynamic types of objects. Indeed, source-language compilers targeting IL emit mostly virtual calls, relying on the JIT compiler to optimise them at load-time. Hence, a type-based, class-hierarchy analysis is too coarse for our purpose. Our analysis is a refinement of the type system for the CLR, and sometimes falls back to type safety, for instance when loading from an array of boxed values.

Since the dynamic permissions depend on the current call stack, a natural approach would be to rely on a general-purpose, context-sensitive, inter-method call graph analysis for the CLR. However, such an analysis would have to be precise enough to simulate the effect of tracing arbitrary, semi-trusted calls on its resulting graph, to reflect precisely dynamic operations on permissions, and to scale up to large libraries.

Instead, we develop a specific analysis that is sensitive to dynamic permissions and many details of the security model, and is otherwise quite coarse. In terms of control flow analyses, this amounts to a particular choice of context-sensitivity. Whereas, for instance, a standard n-CFA [13] would keep track of n frames on the stack, we effectively keep track of a representation of the whole stack that can be used only 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 an abstraction of a security-passing-style implementation of stack-inspection [23], where dynamic permissions are systematically computed and passed as an extra parameter to every method, instead of being extracted lazily from the stack.

3.2 Informal Description of the Algorithm

To every method reachable with some dynamic permissions, we associate a distinct node. The node contains variables for the formal arguments and returned values, plus the current state of the intra-method analysis. Nodes are created on demand, hence only visited method implementations are represented at the end of the analysis.

The abstract values we compute statically are sets of runtime types: to each method formal argument, method result, local variable, static variable, field, and entry on the stack that has a boxed type, we associate a variable collecting the dynamic types that may flow to it. Operations in IL are implemented by constraint generation and propagation.

Intra-method IL code analysis involves the symbolic execution of each code block, using a typed symbolic stack. Each block of code is executed at most once for each dynamic-permissions context D, and translated into a series of constraints. The analysis builds the control flow between these blocks, and connect them using constraints on their entry- and exit-stacks.

In addition to basic constraints for inclusion, equality, and primitive operations such as dynamic type-casts, we use dynamic constraints for virtual calls and for security actions. 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. For example, the constraint for a **demand** has a parameter variable for the permissions being demanded, and conditionally triggers the analysis of the code guarded by the **demand** when this variable is updated. Our constraint solver is rather simple, and keeps selecting unsatisfied constraints until a fixpoint is detected. (Since there is a finite number of nodes, and a finite number of operations in every method body, the process always terminates.) At that point, the

graph provides a sound approximation of reachability and dynamic permissions for every piece of code.

Experimentally, the total runtime of the analysis for large libraries is (just) reasonable: the graph generation converges in a few hours when dealing with core libraries for the CLR (2,687 types yielding 27,276 visited methods).

3.3 Simulating Arbitrary Unknown Code

After loading all assemblies, and in order to generate our domain for values, we create additional class references in a special namespace that stands for all classes that may be defined in unknown code. To this end, we assume that the class hierarchy is upward-closed for classes and interfaces and, for each known class, simulate the rules of inheritance to complete the 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. On the other hand, we do not further instantiate these unknown classes; their implementation may sometimes either inherit or override a particular method, but this case analysis is performed as part of method resolution as we propagate virtual calls.

We initially create a node for each method that is callable by unknown code, with dynamic permissions set to the static permissions of unknown code. We take into account the semantics of the CLR, including scoping rules, access modifiers, inheritance rules, and declarative security actions such as link-demand and inheritance-demand. Unknown code may only operate on objects accessible at runtime, for instance using a public constructor, obtained as a parameter in a call-back, 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 virtual calls).

3.4 Representing Permissions and their Operations

Runtime permissions have a complex, dynamic structure in the CLR More abstractly, 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 declarative security (including any constructor parameters) and is locally inferred in code using an ad hoc, local dataflow analysis.
- A coarse-grain representation—either *NoPermission* or *SomePermissions* for a given permission class—represents dynamic permission contexts D. (Most security queries involve one or two permissions classes, so we adapt our representation accordingly to speed up the analysis.)

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 the metadata and configuration files.

During the analysis, we intercept calls to the security API, such as to assert and demand permissions, and also check for declarative attributes, and transform them into abstract security actions. (In BIL-SEC, these correspond to synthetic instructions **assert** and **demand**.) In the libraries we analyse, the permission parameters for asserts and demands are objects with simple data-flows, so we adopt an ad hoc optimisation: during the analysis, we assume that the variables representing these objects

are assigned a single value, then eventually check this assumption, and would report an unsafe approximation if a more complex dataflow occurred.

3.5 Limitations

We follow a pragmatic approach, in order to scale up to the CLR standard libraries, with very coarse approximations for rare features. We assume that native code and unverifiable IL code preserve runtime type-safety. We do not deal with reflection and some operations on delegates. 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.

4 Querying the Graph

The tool helps to review the security of the code, rather than return a binary answer. Indeed, permission classes define a data structure, rather than an access control policy, which is often left implicit. We run queries on the call-graph to try to extract a global view of their usage of permissions for access control.

At its core, the call graph provides a description of code paths: for each runtime stack, we have a corresponding path in the graph. 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 occurring in the graph from the attacker 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 the shortest (symbolic) paths from the attacker to *Win32Native.DeleteFile* with no security action at all, with a single assert followed by a single demand on some *FileIOPermission*, with a single demand on *IsolatedStoragePermission*, and so on, depending on the libraries. 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.

4.1 Finding the Purpose of Asserts and Demands

Since they affect functionality and performance, as well as security, each dynamic action on permissions should have a clear goal, which we attempt to read 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. Similarly, we check that every $\mathbf{demand}$ is fallible, and try to identify at least one protected operation.

The method calls that may trigger security exceptions depend on the security configuration, for instance on the static permissions granted to libraries; this information needs to be documented, but is hard to extract from the source. The tool provides a systematic way to collect all fallible $\mathbf{demand}$s. For instance, we detected several cases of $\mathbf{demand}$s positioned before a conditional access to a protected method; such errors

may lead to spurious, sometimes undocumented security exceptions as the **demand** fails before the condition.

4.2 Checking Uniformity: Towards Access-Control Policy Extraction

For a given protected operation, we would expect security checks present on control paths to implement the same (implicit) access control discipline. Hence, 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 evaluate the usage of each explicit security action in known code. The output of the tool provides new rules of the form "A permission with these parameters always protects this operation", and also provides counter-examples for each input statement. The model can be interpreted as a "security map" for the library. 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. It is also useful to detect changes in the (implicit) security model as new libraries are added, or as libraries are updated.

4.3 Link-demands, and Other Optimisations

In some libraries, link-demands are routinely used as cheaper, weaker alternative to **demand**s. In effect, only the immediate caller is checked, instead of the call stack. To validate this usage, for each link-demand in the code, we check that the corresponding dynamic permissions would suffice to pass a demand for the same permission.

Queries on the call graph can be used to determine whether ordinary, interprocedural code transformations such as code inlining or tail-call eliminations are correct. More specifically, they can also check the relevance and correctness of several permission-specific performance optimisations. For example, to reduce the runtime cost of stack walks in the CLR, one should preferably use declarative permissions, hoist security actions out of loops, or even demands, then immediately assert a permission to reduce the length of stack walks for the callee. Similarly, we can immediately detect useless **demand**s, or suggest the relocation of some **demand**s to avoid paths with redundant checks.

4.4 Additional Flows

Stack-inspection automatically keeps track of nested calls, but ignores more complex control flows (call-backs, exceptions), and any data flows. Once we have identified parts of the graph protected by permissions, we can use queries that check for local, well-known problems with these flows, such as the escape of private mutable data. As an example, we implemented a query that reports (potential) call-backs from libraries to unknown code. Although we observe a large proportion of virtual calls in libraries that might call back unknown code (from 5% to 10%), only a few of them occur in code that executes with elevated dynamic permission, and most of them use the same static method references, so their manual review turns out to be feasible and interesting. These call-backs 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. We refer to [1] for examples and discussion of this error pattern.

4.5 Implementation and Experimental Results

Our implementation is written in Objective Caml, and relies on the AbstractIL libraries [22] to read and manipulate IL assemblies. It offers a command-line interface for queries, typically evaluated against pre-generated call-graphs. We tried most of these queries on the core libraries for the CLR, *mscorlib.dll* plus *system.dll*. The queries are motivated by common bugs in the development process. We also collected small, synthetic examples that illustrate the usefulness of each query.

5 BIL-SEC: A Model of Access Control in the CLR

Our formal model, BIL-SEC, derives from BIL [12], 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-SEC is 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.

Classes, Fields, Methods, Permissions:

$c \in \mathit{Class}$	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
$S, D \in \mathcal{P}(\mathit{Permission}) = \mathit{PermissionSet}$	permission sets

There are three kinds of data type: void, integer, and reference (for pointers to heap-allocated objects). We associate with the syntax for types a subtype relation $A <: B$ given in Appendix B. generated from a binary relation $c \ \mathit{inherits}\ d$ that indicates that class c is a subclass of the superclass d. We assume that the relation $c \ \mathit{inherits}\ d$ is transitive, and therefore so too is the relation $A <: B$.

Types:

$A, B \in \mathit{Type} ::=$	type
void	nothing
int32	32 bit signed integer
class c	reference

Types are the basis for the syntax of method and constructor signatures, and references, given next. 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 for Methods and Constructors:

$\mathit{sig} \in \mathit{Sig} ::= B\ \ell(A_1, \ldots, A_n)$	method signature
$\mathit{ksig} \in \mathit{Ksig} ::= \mathbf{void}\ .\mathbf{ctor}(A_1, \ldots, A_n)$	constructor signature
$M ::= c::\mathit{sig}$	method reference

$K ::= c::ksig$	constructor reference
$c::sig \triangleq B\ c::\ell(A_1, \ldots, A_n)$ where $sig = B\ \ell(A_1, \ldots, A_n)$	
$c::ksig \triangleq \mathbf{void}\ c::.\mathbf{ctor}(A_1, \ldots, A_n)$ where $ksig = \mathbf{void}\ .\mathbf{ctor}(A_1, \ldots, A_n)$	

We can now specify an *execution environment* as given by an inheritance relation $inherits$, plus three total functions specifying the fields, methods, and static permissions of each class. We assume all method bodies are well-typed. We assume the following axioms:

Execution Environment: $(inherits, fields, methods, statics)$

$inherits \subseteq Class \times Class$	class hierarchy
$fields \in Class \rightarrow (Field \stackrel{\text{fin}}{\rightarrow} Type)$	fields of a class
$methods \in Class \rightarrow (Sig \stackrel{\text{fin}}{\rightarrow} Class \times Body)$	methods of a class
$statics \in Class \rightarrow PermissionSet$	static permissions
$c\ inherits\ c$	(Hi Refl)
$c\ inherits\ c' \wedge c'\ inherits\ c'' \Rightarrow c\ inherits\ c''$	(Hi Trans)
$c\ inherits\ c' \wedge c'\ inherits\ c \Rightarrow c = c'$	(Hi Antisymm)
$c\ inherits\ \mathbf{System.Object}$	(Hi Root)
$c\ inherits\ c' \wedge c\ inherits\ c'' \Rightarrow$ $c'\ inherits\ c'' \vee c''\ inherits\ c'$	(Hi Single)
$c\ inherits\ d \wedge f \in dom(fields(d)) \Rightarrow$ $f \in dom(fields(c)) \wedge$ $fields(c)(f) = fields(d)(f)$	(Hi $fields$)
$c\ inherits\ d \Rightarrow$ $dom(methods(d)) \subseteq dom(methods(c))$	(Hi $methods$)
$methods(c)(sig) = (d, b) \Rightarrow$ $c\ inherits\ d \wedge methods(d)(sig) = (d, b)$	(Hi Meth Impl)

The function $fields(c)$ 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 $methods(c)$ 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 $c::sig$, 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 $statics(c)$ gives the static permissions associated with class c. It models the outcome of the security policy that assigns rights to code when it is loaded.

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 2, 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
$\mathbf{demand}\ P\ a\ \mathbf{else}\ b$	demand P then run a, else run b
$\mathbf{ldarg}\ j$	load method argument $j > 0$ or self if $j = 0$
$a\ \mathbf{starg}\ j$	store result of a into argument $j > 0$
$a_1\ \cdots\ a_n\ \mathbf{newobj}\ K$	create new object with fields $a_1, \ldots, a_n$
$a_0\ a_1\ \cdots\ a_n\ \mathbf{callvirt}\ M$	call M on object a_0 with arguments $a_1, \ldots, a_n$
$a\ \mathbf{ldfld}\ A\ c::f$	load field f of type A from a of class c
$a\ b\ \mathbf{stfld}\ B\ c::f$	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 B 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 for some $c \in C$ and expression $\mathbf{newobj}\ c::\mathbf{void}\ .\mathbf{ctor}()\ \mathbf{callvirt}\ c::\mathbf{void}\ \ell()$ (create a new object of class c then call its method ℓ) there is an evaluation in an empty store during which a virtual call resolves to the particular method implementation M. This intuitive notion is formalised in Appendix B. Method reachability is the subject of a theorem concerning the flow analysis for BIL-SEC, presented next.

6 A Permission-Sensitive Call-Graph Analysis for BIL-SEC

This section describes a permission-sensitive analysis in the formal setting of BIL-SEC; this formal analysis is considerably simpler than the IL implementation reported in Section 3 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.

6.1 Partially-Trusted Environments

To represent code with different levels of trust, we partition Class in the execution environment into trusted classes (libraries, local code) and untrusted classes (applets, plug-ins). Given this partition, we refine our definition of environments to separate trusted code and untrusted code. Our result 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), $\mathit{Virtual}$ (methods overrideable 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

$Methods \triangleq \{c{::}sig \mid methods(c)(sig) = d, b\}$	all defined method references
$Public \subseteq Methods$	callable by untrusted code
$Virtual \subseteq Methods$	overrideable by untrusted code

For all d *inherits* c, and any $c{::}sig \in Methods$,
(0) $d \in Trusted \Rightarrow c \in Trusted$ — Trust decreases with inheritance
(1) $d{::}sig \in Public \Leftrightarrow c{::}sig \in Public$ — $Public$ is invariant by inheritance
(2) $d{::}sig \in Virtual \Rightarrow c{::}sig \in Virtual$ — $Virtual$ decreases with inheritance

For each $c, d \in Untrusted$ such that $methods(c)(sig) = (d, b)$, we have:
(1) $statics(d) = S_\star$
(2) $M \in Public$ for every $\mathbf{callvirt}\ M$ occurring in b
(3) $d{::}sig \in Virtual$

Let $\mathcal{E}_{|\,Trusted}$ be obtained from $\mathcal{E}$ by restricting its domain from $Class$ to $Trusted$:

$$\mathcal{E}_{|\,Trusted} \triangleq (E_{|\,Trusted}, Trusted, S_\star, Public_{|\,Trusted}, Virtual_{|\,Trusted})$$
$$E_{|\,Trusted} \triangleq (inherits_{|\,Trusted}, fields_{|\,Trusted}, methods_{|\,Trusted}, statics_{|\,Trusted})$$

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

6.2 Abstraction of Types

Our analysis associates each body with a symbolic value which denotes the set of possible dynamic types of its result. This approximation is more precise than the static types obtained by typing. The analysis depends only on the trusted classes in $\mathcal{E}_{|\,Trusted}$, 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 $\mathbf{class}\ \star$, 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, $\mathbf{class}\ \star$ is present in $sub^\sharp(\mathbf{class}\ c)$ for any trusted c.

Abstract Types $Type^\sharp$, Type-Safe Abstraction $sub^\sharp(A) \in Type^\sharp$:

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

6.3 Constraints and their Generation

Next, we define the syntax of nodes, symbolic values, and constraints used in our analysis. A 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 $\alpha.i$ 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.

Nodes, Symbolic Values, Constraints:

$\alpha, \beta ::= (M, D)$	node
$t ::=$	symbolic value
$\quad \lambda$	result variable
$\quad \alpha.result$	node result
$\quad \alpha.i$	node parameter
$\quad \{A_1^\sharp, \ldots, A_n^\sharp\}$	constant set of abstract types
$C ::=$	constraint
$\quad \mathbf{T}$	true
$\quad C \wedge C'$	conjunction
$\quad t \subseteq t'$	subset inclusion
$\quad VCALL(\alpha, t_0, t_1, \ldots, t_n, \lambda)$	virtual call constraint

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 runtime 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 associated with the current node α.

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 with $methods(c)(sig) = (c, b)$ 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 $D = D'$.

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.

Generating Constraints from 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 Seq)

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

(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 Assert)
$$\frac{b \Rightarrow^{\alpha}_{D \cup (P \cap statics(c))} t \mid C \quad \alpha = (c{::}sig, D')}{\mathbf{assert}\ P\ b \Rightarrow^{\alpha}_{D} t \mid C}$$

(Gen **ldarg**)
$$\frac{}{\mathbf{ldarg}\ j \Rightarrow^{\alpha}_{D} \alpha.j \mid \mathbf{T}}$$

(Gen **starg**)
$$\frac{a \Rightarrow^{\alpha}_{D} t_a \mid C_a}{a\ \mathbf{starg}\ j \Rightarrow^{\alpha}_{D} \{\mathbf{void}\} \mid C_a \wedge t_a \subseteq \alpha.j}$$

(Gen **ldfld**)
$$\frac{a \Rightarrow^{\alpha}_{D} t_a \mid C_a}{a\ \mathbf{ldfld}\ A\ c{::}f \Rightarrow^{\alpha}_{D} sub^{\sharp}(A) \mid C_a}$$

(Gen **stfld**)
$$\frac{a \Rightarrow^{\alpha}_{D} t_a \mid C_a \quad b \Rightarrow^{\alpha}_{D} t_b \mid C_b}{a\ b\ \mathbf{stfld}\ A\ c{::}f \Rightarrow^{\alpha}_{D} \{\mathbf{void}\} \mid C_a \wedge C_b}$$

(Gen **newobj**)
$$\frac{a_i \Rightarrow^{\alpha}_{D} t_i \mid C_i \quad \forall i \in 1..n}{a_1\ \cdots\ a_n\ \mathbf{newobj}\ c{::}ksig \Rightarrow^{\alpha}_{D} \{\mathbf{class}\ c\} \mid C_1 \wedge \cdots \wedge C_n}$$

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

6.4 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. In the clause for $VCALL$, we consider each possible type in the caller-object's valuation $\mathcal{M}(t_0)$. The corresponding implementation (given by $callee$) has to be analysed. So, the corresponding node belongs to $\mathcal{N}$. In addition, assignments for arguments and results must be consistent. When an untrusted type belongs to $\mathcal{M}(t_0)$, some further approximation is needed if an untrusted implementation may be called. Finally, we define a flow to be *sound* to mean that it has sufficient nodes to satisfy the constraints generated by all the reachable code, and in particular all untrusted abstract implementations (points 1 and 2, below) and all public methods (point 4).

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

$\mathcal{N}$ is $\{\alpha_1, \ldots, \alpha_n\}$, the set of trusted nodes.
$\mathcal{U}$ is $\{(\star{::}-, D_1), \ldots, (\star{::}-, D_m)\}$ with $D_i \subseteq S_\star$, the set of untrusted nodes.
$\mathcal{M}$ maps every t to $\mathcal{M}(t) \subseteq \mathit{Type}^{\sharp}$, with $\mathcal{M}(\{A^{\sharp}_1, \ldots, A^{\sharp}_n\}) = \{A^{\sharp}_1, \ldots, A^{\sharp}_n\}$.

Let $callee(c{::}sig, D) \triangleq (d{::}sig, statics(d) \cap D)$ where $(d, b) = methods(c)(sig)$.
We define a predicate $(\mathcal{N}, \mathcal{U}, \mathcal{M}) \models C$ by induction on the size of C.

- $(\mathcal{N}, \mathcal{U}, \mathcal{M}) \models \mathbf{T}$ always
- $(\mathcal{N}, \mathcal{U}, \mathcal{M}) \models t \subseteq t'$ iff $\mathcal{M}(t) \subseteq \mathcal{M}(t')$
- $(\mathcal{N}, \mathcal{U}, \mathcal{M}) \models C \wedge C'$ iff $(\mathcal{N}, \mathcal{U}, \mathcal{M}) \models C$ and $(\mathcal{N}, \mathcal{U}, \mathcal{M}) \models C'$.
- $(\mathcal{N}, \mathcal{U}, \mathcal{M}) \models VCALL((c{::}sig, D), t_0, \ldots, t_n, \lambda)$ iff:

1. For all types $\mathbf{class}\ d \in \mathcal{M}(t_0)$:
   - let $\alpha = \mathit{callee}(d{::}\mathit{sig}, D)$ in
     $\alpha \in \mathcal{N} \wedge (\mathcal{N},\mathcal{U},\mathcal{M}) \models \bigwedge_{i\in 0..n} t_i \subseteq \alpha.i \wedge \alpha.\mathit{result} \subseteq \lambda$
2. If $\mathbf{class}\ \star \in \mathcal{M}(t_0)$ then, for all $d \in \mathit{Trusted}$ with $d\ \mathit{inherits}\ c$,
   - let $\alpha = \mathit{callee}(d{::}\mathit{sig}, D)$ in
     $\alpha \in \mathcal{N} \wedge (\mathcal{N},\mathcal{U},\mathcal{M}) \models \bigwedge_{i\in 0..n} t_i \subseteq \alpha.i \wedge \alpha.\mathit{result} \subseteq \lambda$
   - if $d{::}\mathit{sig} \in \mathit{Virtual}$ then
     $(\star{::}-, D \cap S_\star) \in \mathcal{U} \wedge (\mathcal{N},\mathcal{U},\mathcal{M}) \models \mathit{sub}^\sharp(B) \subseteq \lambda$

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 $\alpha = (c{::}\mathit{sig}, D) \in \mathcal{N}$,
   - $\mathit{methods}(c)(\mathit{sig}) = (c, b)$ for some b, and
   - if $b \Rightarrow^\alpha_D t \mid C$ then $(\mathcal{N},\mathcal{U},\mathcal{M}) \models t \subseteq \alpha.\mathit{result} \wedge C$.
4. For all $(\star{::}-, D) \in \mathcal{U}$ and $(c{::}B\ \ell(A_1,\ldots,A_n)) \in \mathit{Public}$,
   let $\alpha = \mathit{callee}((c{::}B\ \ell(A_1,\ldots,A_n)), D)$ and $A_0 = \mathbf{class}\ c$.
   We have $\alpha \in \mathcal{N}$ and, for $i \in 0..n$, $(\mathcal{N},\mathcal{U},\mathcal{M}) \models \mathit{sub}^\sharp(A_i) \subseteq \alpha.i$

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.

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

To illustrate our definitions and the theorem, Appendix A describes correct flows for the trusted libraries *File* and *CFile*. A corollary is that the flow for *File* guarantees there is no path from untrusted code to *Win32*::*Delete*; on the other hand, there is such a path when the analysis includes *CFile*.

7 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 [23], its limitations and formal semantics [10, 6], or on alternative mechanisms [17, 9, 1].

Pottier, Skalka and Smith [20] develop a type system for a λ-calculus with stack inspection that statically ensures that, in any well-typed term, no $\mathbf{demand}$ 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 [19] analyse the complexity

of deciding whether a whole program satisfies a security property. Jensen, Le Métayer and Thorn [14] introduce a framework for formal security properties, including stack inspection.

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 particular method in a closed set of classes. Bartoletti, Degano and Ferrari [3] provide a control flow analysis for optimising stack inspection. They calculate safe approximations of denied or granted permissions at each point of the program. This information is useful to minimise the number of stack frames to inspect.

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 data-structures to build a context-sensitive call graph [13].

8 Conclusions and Future Work

We implemented a new control-flow-based analysis for runtime permissions in the CLR, and 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 attacker code, as described in Section 3, is new, as is the catalogue of queries in Section 4 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 [18].

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.

8.1 Acknowledgements

Toshiyuki Maeda implemented some of the security queries. Frédéric Besson suggested improvements to a draft.

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *10th Annual Network and Distributed System Symposium (NDSS'03)*. 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: Int. Workshop on Concurrency and Coordination*, volume 54 of *ENTCS*. Elsevier, 2001.

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

[5] J. Clements and M. Felleisen. A tail-recursive semantics for stack inspections. In *ESOP 2003*, volume 2618 of *LNCS*, pages 22–37. Springer-Verlag, 1999.

[6] 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.

[7] C. Fournet and A. D. Gordon. Stack inspection: Theory and variants. In *29th ACM Symposium on Principles of Programming Languages (POPL'02)*, pages 307–318, Jan. 2002.

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

[9] 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.

[10] 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.

[11] 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.

[12] L. Koved, M. Pistoia, and A. Kershenbaum. Access rights analysis for Java. In *OOPSLA*, pages 359–372, 2002.

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

[14] 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.

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

[16] 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.

[17] 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.

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

[19] 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 Example Flows

We apply our sensitive-permission call-graph analysis to the example we mentioned in section 2.3. Here are some hypotheses for this analysis:

- $File, CFile, Win32 \in \mathit{Trusted}$
- $\mathit{statics}(File) = \mathit{statics}(CFile) = \mathit{statics}(Win32) = S_t$
- $Win32{::}Delete \notin \mathit{Public}$
- No method of $File$, $CFile$, $Win32$ is in $\mathit{Virtual}$.
- $\{FilePermission\} \cap S_\star = \varnothing$
- $D = S_\star \cup (\{FilePermission\} \cap S_t)$

Adding the naive library (where $CFile$ is defined) offers an involuntary opportunity of using $Win32{::}Delete$ to delete any file. A way of exploiting this opportunity is illustrated by $BadFile{::}DeleteAny$.

In the following, our intent is to show how our analysis allows to detect such opportunities created by an addition of code.

We first analyse the main library, where $File$ is defined. We then add to the analysis the naive trusted library and we compare the results of these analyses.

Analysis of the Main Trusted Library

For $\mathit{Trusted} = \{File\}$, the minimal correct flow is $(\mathcal{N}_1, \mathcal{U}_1, \mathcal{M}_1)$ where:
$\mathcal{N}_1 = \{(File{::}Delete, S_\star) \quad (File{::}Backup, S_\star) \quad (File{::}Cleanup, S_\star)\}$
$\mathcal{U}_1 = \{(\star{::}-, S_\star)\}$
For instance, if we consider $\alpha = (File{::}Delete, S_\star)$, we have:
$\mathcal{M}_1(\alpha.0) = \{File, \mathbf{class}\ \star\}$
$\mathcal{M}_1(\alpha.1) = \{\mathbf{string}\}$
$\mathcal{M}_1(\alpha.\mathit{result}) = \{\mathbf{void}\}$

Analysis of the Main and the Naive Trusted Libraries

For $\mathit{Trusted} = \{File, CFile\}$, the minimal correct flow is $(\mathcal{N}_2, \mathcal{U}_2, \mathcal{M}_2)$ where:
$\mathcal{N}_2 = \mathcal{N}_1 \cup \{(CFile{::}Commit, S_\star) \quad (File{::}Delete, D) \quad (File{::}Backup, D)$
$(File{::}Cleanup, D) \quad (Win32{::}Delete, D)\}$
$\mathcal{U}_2 = \mathcal{U}_1 = \{(\star{::}-, S_\star)\}$

Sketch of the constraints generated by $\beta = (CFile{::}Commit, S_\star)$

$$\frac{a_0\ a_1\ \mathbf{callvirt}\ File{::}Cleanup \Rightarrow_D^\beta t \mid C}{\mathbf{assert}\ FilePermission\ (a_0\ a_1\ \mathbf{callvirt}\ File{::}Cleanup) \Rightarrow_{S_\star}^\beta t \mid C}$$

Sketch of the constraints generated by $\alpha' = (File{::}Delete, D)$

$$\frac{(FilePermission \in D) = true \quad a_0\ a_1\ \mathbf{callvirt}\ Win32{::}Delete \Rightarrow_D^{\alpha'} t \mid C}{\mathbf{demand}\ FilePermission\ (a_0\ a_1\ \mathbf{callvirt}\ Win32{::}Delete)\ (\mathbf{ldarg}\ 1) \Rightarrow_D^{\alpha'} t \mid C}$$

If we consider the constraints generated in $\beta = (CFile{::}Commit, S_\star)$, we see in the sketch above that the call to $File{::}Cleanup$ occurs with dynamic permissions D which implies, according to the correct flow definition, $(File{::}Cleanup, D) \in \mathcal{N}$, which then entails $\alpha' \in \mathcal{N}$.

In the generation of constraints α', thanks to larger dynamic permissions (in comparison with the situation in α in the first analysis), the $true$ branch of **demand** is selected, which implies $(Win32{::}Delete, D) \in \mathcal{N}$.

Using Theorem 1, we deduce from the result of the static analysis that $Win32{::}Delete$ is *unreachable* when the naive library is absent. Adding the naive library implies the appearance of the node $(Win32{::}Delete, D)$, which means that $Win32{::}Delete$ is possibly reachable.

B BIL-SEC: Additional Definitions

B.1 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
$\quad \mathbf{0}$	$\quad$ void
$\quad i4$	$\quad$ integer
$\quad p$	$\quad$ 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
$fr ::= .\mathbf{args}(u_0, \ldots, u_n)$	frame: vector of arguments
$s ::= fr_1 \cdots fr_n$	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'

$$
\begin{array}{ll}
\mathit{dynClass}((h,s),p) = c & \text{if } h(p) = c[f_i \mapsto u_i \ ^{i\in 1..n}] \\
\mathit{lookup}((h,s),p.f_j) = v_j & \text{if } h(p) = c[f_i \mapsto u_i \ ^{i\in 1..n}] \text{ and } j \in 1..n \\
\mathit{update}((h,s),p.f_j,v') = ((p \mapsto c[f_j \mapsto v', f_i \mapsto u_i \ ^{i\in(1..n)-\{j\}}], h'), s) & \\
\qquad\qquad \text{if } h = p \mapsto c[f_i \mapsto u_i \ ^{i\in 1..n}], h' \text{ and } j \in 1..n &
\end{array}
$$

As in Fournet and Gordon's formulation of stack inspection [10], 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 formalize evaluation by a judgment of the following form:[1]

Evaluation Judgment:

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

Evaluation Rules for Control Flow:

(Eval **ldc**)

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

(Eval Seq)

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

(Eval Demand)

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

(Eval Assert)

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

The expression $\mathbf{ldc.i4}\ i4$ evaluates to the integer $i4$. 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 **ldarg**)

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

(Eval **starg**)

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

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.

[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 judgment by passing only the heap and the top frame (h, fr) instead of the heap plus the stack $\sigma = (h, s)$.

Evaluation Rules for Objects:

(Eval **newobj**)

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

(Eval **callvirt**)

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

(Eval **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 **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 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 $\mathit{statics}(d)$ where d is the implementation superclass, and with the current dynamic permissions adjusted by intersecting with $\mathit{statics}(d)$. The result of this evaluation is the store $(h', s'\ \mathit{fr}')$, 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 $a\,\mathbf{ldfld}\,A\,c{::}f$ evaluates a to an object reference, then returns field f of this object.

The expression $a\ b\ \mathbf{stfld}\ A\ c{::}f$ evaluates a to a reference to an object, updates its field f with the result of evaluating b, and returns void.

B.2 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 $\sigma \vdash b \rightsquigarrow_D^S v \cdot \sigma'$, we associate the (unique) derivation tree obtained from the rules of Section B.1. To each instance in the tree of the rule:

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

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

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

$$(\epsilon, \epsilon) \vdash (\mathbf{newobj}\ \mathbf{void}\ c{::}.\mathbf{ctor}())\ \mathbf{callvirt}\ \mathbf{void}\ c{::}l() \rightsquigarrow_S^S v \cdot \sigma'$$

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

By labelling with $d{::}\mathit{sig}$, we mark the method implementations whose body 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 characterize the code that an attacker could trigger.

B.3 Typing Method Bodies

The following is a fairly standard object-oriented type system, closely based on the BIL type system, and mostly independent of permissions.

Type Frames and Typing Judgment:

$\mathit{Fr} ::= .\mathbf{args}(A_0, \ldots, A_n)$	frame: types of arguments
$\mathit{Fr} \vdash b : B$	given Fr, body b returns type B

Additional Assumptions:

$$\mathit{methods}(c)(B\ \ell(A_1, \ldots, A_n)) = (d, b)\ \Rightarrow \quad .\mathbf{args}(\mathbf{class}\ c, A_1, \ldots, A_n) \vdash b : B \qquad \text{(Ref } \mathit{methods}\text{)}$$

Typing Rule for Subsumption:

$$\frac{\text{(Body Subsum)} \quad \mathit{Fr} \vdash b : B \quad B <: B'}{\mathit{Fr} \vdash b : B'}$$

Typing Rules for Control Flow:

$$\frac{\text{(Body } \mathbf{ldc}\text{)}}{\mathit{Fr} \vdash \mathbf{ldc.i4}\ i4 : \mathbf{int32}} \qquad \frac{\text{(Body Seq)} \quad \mathit{Fr} \vdash a : \mathbf{void} \quad \mathit{Fr} \vdash b : B}{\mathit{Fr} \vdash a\ b : B}$$

$$\frac{\text{(Body Demand)} \quad \mathit{Fr} \vdash a : B \quad \mathit{Fr} \vdash b : B}{\mathit{Fr} \vdash \mathbf{demand}\ P\ a\ \mathbf{else}\ b : B} \qquad \frac{\text{(Body Assert)} \quad \mathit{Fr} \vdash a : B}{\mathit{Fr} \vdash \mathbf{assert}\ P\ a : B}$$

Typing Rules for Arguments:

(Body **ldarg**)
$$\frac{j \in 0..n}{\mathbf{.args}(A_0, \ldots, A_n) \vdash \mathbf{ldarg}\ j : A_j}$$

(Body **starg**)
$$\frac{\mathbf{.args}(A_0, \ldots, A_n) \vdash a : A_j \quad j \in 0..n}{\mathbf{.args}(A_0, \ldots, A_n) \vdash a\ \mathbf{starg}\ j : \mathbf{void}}$$

Typing Rules for Reference Types:

(Ref **newobj**) (where $\mathit{fields}(c) = f_i \mapsto A_i\ ^{i \in 1..n}$)
$$\frac{\mathit{Fr} \vdash a_i : A_i \quad \forall i \in 1..n}{\mathit{Fr} \vdash a_1\ \cdots\ a_n\ \mathbf{newobj}\ \mathbf{void}\ c::\mathbf{.ctor}(A_1, \ldots, A_n) : \mathbf{class}\ c}$$

(Ref **callvirt**) (where $B\ \ell(A_1, \ldots, A_n) \in \mathit{dom}(\mathit{methods}(c))$)
$$\frac{\mathit{Fr} \vdash a_0 : \mathbf{class}\ c \quad \mathit{Fr} \vdash a_i : A_i \quad \forall i \in 1..n}{\mathit{Fr} \vdash a_0\ a_1\ \cdots\ a_n\ \mathbf{callvirt}\ B\ c::\ell(A_1, \ldots, A_n) : B}$$

(Ref **ldfld**) (where $\mathit{fields}(c) = f_i \mapsto A_i\ ^{i \in 1..n}$)
$$\frac{\mathit{Fr} \vdash a : \mathbf{class}\ c \quad j \in 1..n}{\mathit{Fr} \vdash a\ \mathbf{ldfld}\ A_j\ c::f_j : A_j}$$

(Ref **stfld**) (where $\mathit{fields}(c) = f_i \mapsto A_i\ ^{i \in 1..n}$)
$$\frac{\mathit{Fr} \vdash a : \mathbf{class}\ c \quad \mathit{Fr} \vdash b : A_j \quad j \in 1..n}{\mathit{Fr} \vdash a\ b\ \mathbf{stfld}\ A_j\ c::f_j : \mathbf{void}}$$

B.4 Typing the Memory Model

To validate the type system, we need to assign types to runtime data structures including results, heaps, stacks, and stores. The main difference between the following and the corresponding rules for BIL is the omission of pointers into the stack.

Heap, Stack, and Store Types:

$H ::= p_i \mapsto c_i\ ^{i \in 1..n}$	heap type
$\mathcal{S} ::= \mathit{Fr}_1 \cdots \mathit{Fr}_n$	stack type
$\Sigma ::= (H, \mathcal{S})$	store type

Conformance Judgments:

$H \models u : A$	in Σ, result u has type A
$H \models o : c$	in H, object o has class c
$H \models h$	heap h conforms to H
$H \models \mathit{fr} : \mathit{Fr}$	frame fr conforms to Fr
$\Sigma \models \sigma$	store σ conforms to Σ

Conformance Rules

(Con Object) (where $\mathit{fields}(c) = f_i \mapsto A_i\ ^{i \in 1..n}$)
$$\frac{H \models v_i : A_i \quad \forall i \in 1..n}{H \models c[f_i \mapsto v_i\ ^{i \in 1..n}] : c}$$

(Res Void)
$$\frac{}{H \models \mathbf{0} : \mathbf{void}}$$

(Res Int)
$$\frac{}{H \models i4 : \mathbf{int32}}$$

(Res Ref)
$$\frac{H(p) = c \quad c\ \mathit{inherits}\ c'}{H \models p : \mathbf{class}\ c'}$$

(Con Heap) (where $H = p_i \mapsto c_i \ ^{i \in 1..n}$)

$$\frac{H \models o_i : c_i \quad \forall i \in 1..n}{H \models p_i \mapsto o_i \ ^{i \in 1..n}}$$

(Con Frame)

$$\frac{H \models u_i : A_i \quad \forall i \in 0..n}{H \models .\mathbf{args}(u_0, \ldots, u_n) : .\mathbf{args}(A_0, \ldots, A_n)}$$

(Con Store)

$$\frac{H \models h \quad H \models fr_i : Fr_i \quad \forall i \in 1..n}{(H, Fr_1 \cdots Fr_n) \models (h, fr_1 \cdots fr_n)}$$

Finally, our type safety result is a slight reformulation of type safety for BIL. If a program satisfies the restrictions on type structure and the typing rules for method bodies then its evaluation can lead only to conformant intermediate states. Let $H \leq H'$ mean that $dom(H) \subseteq dom(H')$ and $H(p) = H'(p)$ for all $p \in dom(H)$.

Theorem 2 *If $(H, \mathcal{S}\ Fr) \models \sigma$ and $Fr \vdash b : B$ and $\sigma \vdash b \rightsquigarrow^S_D v \cdot \sigma^\dagger$ then there exists a heap type $H^\dagger$ such that $H \leq H^\dagger$ and $H^\dagger \models v : B$ and $(H^\dagger, \mathcal{S}\ Fr) \models \sigma^\dagger$.*

The proof is an easy adaptation of Gordon and Syme's proof [12].