Static Verification of Code Access Security Policy Compliance of .NET Applications

Jan Smans
Dept. of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
jans@cs.kuleuven.ac.be

Bart Jacobs
Dept. of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
bartj@cs.kuleuven.ac.be

Frank Piessens
Dept. of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
frank@cs.kuleuven.ac.be

ABSTRACT

The base class library of the .NET Framework makes extensive use of the Code Access Security system to ensure that partially trusted code can be executed securely. Imperative or declarative permission demands indicate where permission checks have to be performed at run time to make sure partially trusted code does not exceed the permissions granted to it in the security policy.

In this paper we propose expressive method contracts for specifying required security permissions, and a modular static verification technique for Code Access Security based on these method contracts. If a program verifies, it will never fail a run time check for permissions, and hence these run time checks can be omitted.

Advantages of our approach include improved run time performance, and improved and checkable documentation for security requirements. Our system builds on the Spec# programming language and its accompanying static verification tool.

Keywords

static verification, code access security, stack inspection, Spec#

1. INTRODUCTION

Nowadays, most software is created by combining components from various sources. Some programs can even be extended at run time with new components. For example, by extending a media player with a new codec, additional content can be displayed. However, not all parts of a composed program are necessarily equally trusted. For instance, a codec, embedded in a media player, may not be trusted to create network connections while the player itself does have that permission. Nonetheless, all parts, whether they are trusted or not, share the same process space, i.e. memory, processor etc.

To allow execution of heterogeneous programs (i.e. programs composed from parts with different permissions), the Common Language Runtime (CLR) and the Java Virtual Machine (JVM) offer a fine-grained access control mechanism called stack inspection [Gon02a, Fou02a]. The CLR uses the term Code Access Security (CAS) to refer to the stack inspection machinery. A trusted library can rely on this mechanism to protect the resources it encapsulates. The basic idea is to prevent unauthorized access to resources by guarding every sensitive operation by an access control check. This check determines whether the requested operation is allowed by inspecting (every frame on) the call stack. The Base Class Library makes extensive use of CAS to protect access to files, network resources, and so forth.

While stack inspection has proven its usefulness in the past, it also has a number of shortcomings [Wal00a, Aba03a, Pot01a]. First of all, run time checking is used to enforce the security policy. These run time checks can incur a substantial performance overhead. Secondly, since access control checks are part of the implementation of library code, and since such checks are scattered throughout the implementation, it is hard to understand what is actually enforced. This is an issue for the developers of the library code: it is hard to validate that no access checks have been omitted, and that a

.NET Technologies'2005 *conference proceedings,*
ISBN 80-903100-4-4

consistent security policy is enforced [Bes04a]. It is also an issue for developers of client code that calls the library: they will have to rely on informal documentation to infer what permissions their code will actually need to run properly [Kov02a]. Moreover, the risk that documentation becomes stale as library code evolves is real.

In this paper, we propose formal method contracts specifying the CAS related behavior of methods, and we propose a modular static verification technique. For a library developer, successful static verification of a library method ensures that the implementation respects the method contract. Hence, the method contract can be seen as an improved and checkable documentation for possible security exceptions. For the developer of client code, successful static verification of a program (under an assumed minimal permission set for the client code) ensures that no run time check for permissions will ever fail. Successful static verification by the CLR at load time (under the actual permission set for the client code) proves that it is safe to turn off run time checks.

Our system builds on the Spec# programming language (itself an extension of C#) [Bar04a] and its accompanying static verification tool.

The rest of this paper is structured as follows: in section 2 we briefly review the mechanism of Code Access Security, and the Spec# programming system. In section 3 we discuss the abovementioned problems of CAS in more detail, and we define the goal of this paper. Next, we present our proposed solution in detail (section 4), and discuss its advantages and disadvantages (section 5). Finally, we compare with related work and conclude.

2. BACKGROUND

Code Access Security

Code Access Security (CAS) defines code access rights by means of *permissions*. A permission is a first-class object that represents a right to access certain resources. A `FileIOPermission` object for instance represents the right to perform certain operations (read, write, ...) on certain files. Permission objects actually represent *sets* of more primitive permissions, and it is always possible to take the union or intersection of two permission objects of the same type. A PermissionSet object groups permissions of different types.

Permissions are assigned to assemblies based on *evidence*. Examples of evidence include: location where the assembly was downloaded from, or the code publisher that digitally signed the assembly. The *security policy* is a configurable function that maps evidence to permission sets. The resulting permission set for a given assembly is called the *static permission set*. In this paper, we assume static permission sets can be approximated sufficiently, so we don't elaborate on evidence and the security policy evaluation process. In particular, when verifying client code for which the static permission set is not yet known, we will rely on a CAS assembly level attribute that the developer of client code can set to indicate the minimal static permission set his code needs to run properly.

The CLR maintains for every thread an associated *dynamic permission set* that represents the actual access rights that the thread has at this point in its execution. The dynamic permission set is not represented explicitly in the CLR, but is computed by stack inspection: it defaults to the intersection of the static permission sets of all code that is currently on the call stack, but trusted library code can influence the stack inspection process as discussed below.

Library code can control access to protected resources by means of the following operations on permission objects:

- Calling `Demand` on a permission object p checks if p is in the dynamic permission set. This operation initiates a stack walk: all frames on the stack (from top to bottom) are checked for permission p. If a frame is encountered that doesn't have permission p in its static permission set, a `SecurityException` is thrown. Otherwise, `Demand` just terminates normally without any side-effects. This method is used by library code to guard sensitive operations from being accessed by semi-trusted code.
- When calling `Assert` on a permission object p, the current stack frame is marked privileged for permission p. If such a frame is encountered during stack inspection for permission p, Demand returns normally. Hence, asserting a permission makes the dynamic permission set grow. Asserting a permission is used by highly trusted code to allow less trusted code to access some resource in a well-defined, secure way.

Our analysis of the Rotor BCL, a partial, shared-source implementation of the BCL [Stu03a], has shown that other operations on permission objects, such as `Deny` and `PermitOnly`, occur only rarely. Therefore, we do not consider them in this paper.

Operations on permissions can be done imperatively: they are just method calls on objects. However, the Code Access Security system also supports a limited form of *declarative* operations on permission objects:

an attribute can be placed on a method to indicate that a specific operation on a specific permission must be performed before execution of the method. Declarative CAS can be seen as a first step towards making the CAS behavior of a method more explicit. In their current form, declarative demands have limited expressive power: permissions that depend on the state of the program cannot be demanded in a declarative fashion. For example, to demand `FileIOPermission` for a path that was given as a parameter to the method, one must resort to imperative demands.

The CAS system has numerous other features such as link demands and inheritance demands that we do not discuss here. We refer the reader to [Fre03a] for full details.

Spec#/Boogie

The Spec# Programming System [Bar04a] consists of three parts: an object-oriented language called Spec#, a compiler, and a program verifier, called Boogie. The language Spec# is an extension of C#. It extends C# with non-null types, checked exceptions, and constructs for writing specifications, such as object invariants and pre- and post-conditions for methods. Our proposed system builds on Spec#'s support for writing specifications.

The Spec# compiler emits run-time checks for these specifications, and adds specification information as metadata to the generated assembly. The static verifier, Boogie, takes such an assembly with specification metadata, and statically verifies the consistency between the implementation and the specification. The verification is sound, but not complete.

3. PROBLEM STATEMENT

Problems with CAS

While Code Access Security is a usable and essential part of the .NET security infrastructure, it has a number of well-known shortcomings. These can be summarized as follows:

1. Code Access Security is implemented using dynamic checks, which can have a substantial impact on performance. Moreover, being based on stack inspection, Code Access Security can hinder optimizations that affect the execution stack.
2. Security checks are typically part of the implementation of a method and as such, their effect is not visible in the signature of the method: the (informal) documentation has to specify under what circumstances security exceptions will be thrown. Writing and maintaining precise documentation is error-prone.
   While declarative security demands partly deal with this problem, they do not have the same expressive power as imperative demands, and our analysis of the Rotor BCL shows that approximately 60% of all demands are imperative demands.
3. Not only are security checks part of the implementation, they are scattered throughout the BCL. Our analysis of the Rotor BCL found 183 demands scattered across 40 classes. This makes it very hard to understand what the Code Access Security system actually enforces.
4. Finally, stack inspection tries to protect against luring attacks, where partially trusted code uses trusted but naive code to accomplish an attack. But stack inspection only addresses luring attacks based on method calls from semi-trusted to trusted code, and does not deal with other potential interactions, such as the reliance on results from semi-trusted code, or exceptions thrown from such code.

Many researchers have recognized these shortcomings of sandboxing based on stack inspection, and have proposed partial solutions [Pot01a, Aba03a,Wal00a, Fou02a, Bes04a]. We refer to the related work section for a detailed discussion.

This paper builds on these existing solutions and on the Spec# specification and verification infrastructure to propose a new solution that addresses (at least in part) the first three disadvantages identified above. In the discussion section, we also briefly indicate how our approach could be extended to deal also with the last disadvantage.

Goal

Our goal is to define method contracts for CAS that support modular static verification of an assembly with a known static permission set.

Figure 1 concretizes this goal in the form of a tool called *casverify*. To verify an assembly (i.e. verify whether it could ever throw a `Security-Exception`) for a given set of static permissions, we input that assembly, together with the specifications of all referenced assemblies, to *casverify*. The tool then determines whether execution of the given assembly could ever cause a demand to fail.

Note that we use the term Spec#$_{perm}$ to indicate that the input consists of assemblies annotated with the permission-preconditions proposed in this paper.

Our tool *casverify* is sound, but incomplete. In order to be useful, it requires method contracts and hence introduces annotation overhead.

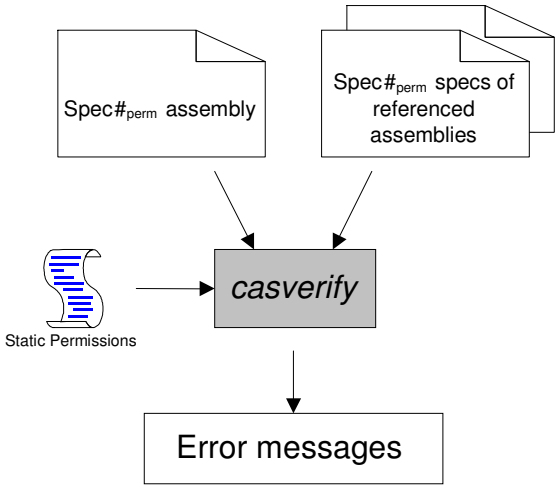

Figure 1*: casverify*

We envision three use cases:

Library developers must invest the effort to write precise method contracts. These contracts can be seen as a formal kind of documentation. A successful static verification ensures that the documentation is correct, in the sense that any method in the library assembly will never throw any security exceptions if it is called with a dynamic permission set that respects the preconditions.

Developers of client code need not invest the effort of writing precise method contracts. We assume they just specify the requested minimum permission set for each assembly, using assembly level declarative security attributes. Each method in the assembly then gets a (overly conservative) precondition that requires this declared minimum permission set. If client assemblies can be statically verified under these method contracts, one can be sure that no security exceptions will be thrown at run time.

At assembly load time, the CLR can input an assembly (together with its corresponding static permissions and referenced assemblies) to *casverify* to determine whether it is safe to turn off run time checking for that assembly.

4. APPROACH

To verify an assembly for a given set of static permissions, we first input that assembly, together with the specifications of all referenced assemblies, to a program transformer. This program transformer implements a transformation similar to Wallach's Security-passing Style (SPS) transformation [Wal00a]. The output of this transformation is a Spec# assembly (plus corresponding specifications for referenced methods) that can be verified by Boogie. If Boogie can show that the transformed assembly is correct, the original assembly will never raise a `SecurityException` when executed with the given static permissions (or more). Figure 2 shows how all this translates to an implementation for *casverify*.

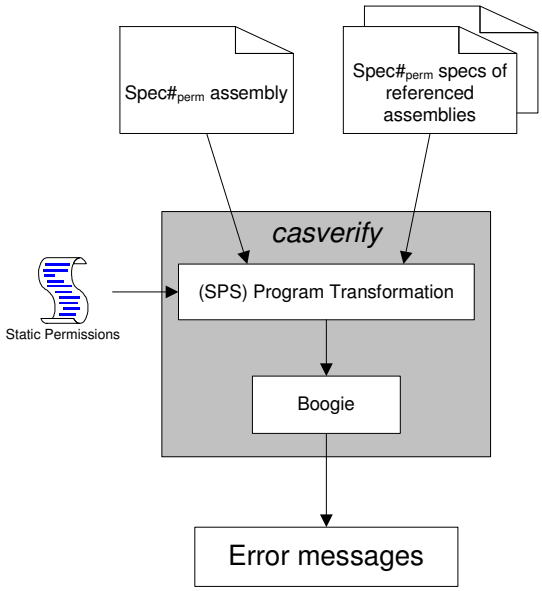

Figure 2: Implementation of *casverify*

In this section we first illustrate the basic idea behind our approach using a very simple example. Secondly, we show how to extend this idea towards more complex scenarios.

The Basic Idea

To keep our explanation as clear and simple as possible, we make some assumptions about the programs we consider in this subsection. First of all, we assume that only one permission type is used, namely `XPermission`. An assembly either has this permission or has no permission at all. Secondly, we do not consider permissions that take parameters, so `XPermission` objects have no parameters.

To be able to prove that for a given policy no permission demand will ever fail in a certain assembly, we require each of its methods and all referenced methods to be annotated using preconditions specifying the minimal required dynamic permission set of the method's callers. For libraries, we expect developers to write these annotations; for client code, these preconditions correspond to the requested minimum permission set.

A method execution may (directly or indirectly) raise a `SecurityException` if its caller violates a permission-precondition[1], i.e. if the dynamic permission set of its caller does not include the minimal dynamic permission set specified in the precondition. In order to prove that no method in a certain assembly will ever throw such an exception, we have to show that 1) no method implementation violates a callee's permission-precondition and that 2) each method's permission-precondition is sufficiently strong to make every demand in its body succeed.

In a Spec# program, the dynamic permission set is not represented explicitly in the CLR in a separate data structure, but is computed by stack inspection. However, to be able to mention it in our specifications, we assume every method has access to a variable `s`[2] that represents the dynamic permission set of its caller. Because we assumed that the programs we are verifying use only one permission type, namely `XPermission`, it suffices to give `s` the type bool. `s` is true if and only if the dynamic permission set includes `XPermission`.

```
class LibraryClass{
 void DoSensitive(int level)
   requires s==true;
 {
   new XPermission().Demand();
   //do sensitive operation
 }
 void SafeDoSensitive()
   requires true;
 {
   new XPermission().Assert();
   DoSensitive(2);
 }
}
```

Figure 3: Class LibraryClass

Consider the class `LibraryClass` of Figure 3. This class contains two methods: `DoSensitive` and `SafeDoSensitive`. The former method performs a sensitive operation after demanding `XPermission`. The sensitivity of the operation depends on the parameter `level`: if `level` is large, the operation becomes more "dangerous". The latter method, `SafeDoSensitive`, allows any code, even code that doesn't have `XPermission` in its static permission set, to perform the sensitive operation, but only for `level` equal to two. We assume that `LibraryClass` is part of a trusted library and that the static permission set of that library contains `XPermission`. The developer of that class has annotated the method `DoSensitive` with a precondition, specifying that it should only be called when `s` is true. In other words, the developer specified that the dynamic permission set of callers of `DoSensitive` should contain `XPermission`. Note that giving `XPermission` to a piece of code, allows it to perform the sensitive operation for any value of level. `SafeDoSensitive` has no real precondition: it can be called by any code, in any context.

```
SPS(m(a1,…,an){Body}) ≡                    (1)
        m(a1,…,an,bool s){
          s = s && StaticPerm();
          SPS(Body)
        }
SPS(o.m(x1,…,xn);) ≡                       (2)
        o.m(x1,…,xn,s);
SPS(p.Demand();) ≡                         (3)
        assert s;
SPS(p.Assert();) ≡                         (4)
        assert StaticPerm();
        s = true;
```

Figure 4: (SPS) program transformation

Next, we discuss the SPS program transformation. Operations that modify the call stack, such as method calls and permission assertions, also (potentially) modify the dynamic permission set. For example, when `XPermission` is successfully asserted, `s` becomes `true`. To make these modifications explicit, the SPS program transformation inserts additional operations to update `s`. Figure 4 shows what transformations have to be applied to each part of the program[3]. Note that the transformed program is used only for static verification; the original program is executed. Furthermore, note that this transformation can be entirely automated and that no user interaction is required. When reading the transformation rules, keep in mind the difference between Assert() (i.e. calling the Assert() method on a permission object), and `assert` (the assertion of a boolean invariant that the static verifier will have to prove). For instance, rule (3) says that at a program point where a Demand() is done, the verifier should prove that `s` is true (i.e. `XPermission` is in the dynamic permission set).

[1] From now on, we will use the term *permission-precondition* to refer to any precondition that constrains the caller's dynamic permission set.

[2] This variable is only needed for verification purposes and is not present at run-time.

[3] Note that the SPS-transformation shown in Figure 4 could be applied to IL-code to make it language independent.

```
class LibraryClass{
 void DoSensitive(int level, bool s)
   requires s == true;
 {
   s = s && StaticPerm();
   assert s;
   //do sensitive operation
}
 void SafeDoSensitive(bool s)
 {
   s = s && StaticPerm();
   assert StaticPerm();
   s = true;
   DoSensitive(2, s);
}
 static bool StaticPerm()
   ensures result == true;
 {
   return true;
 }
}
```

Figure 5: LibraryClass after transformation

Figure 5 shows the result of the program transformation for `LibraryClass`. During verification, we assume that the policy assigns `XPermission` to this class. This is encoded via the method `StaticPerm`: this method returns `true` if the static permission set of its class contains `XPermission`; otherwise, it returns `false`.

```
[assembly:PermissionSetAttribute(
RequestMinimum, Name = "Execution")]
class ClientClass{
 LibraryClass! t;
 void m1()
 {
   t.DoSensitive(5);
 }
 void m2()
 {
   t.SafeDoSensitive();
 }
}
```

Figure 6: Class ClientClass

Using a static program verifier, such as Boogie, we can verify `LibraryClass`. Boogie checks (among others) that preconditions hold at every call-site and that every assert-statement will succeed at run time. If we can prove the correctness of the transformed class, we know that using the original class under a dynamic permission set that satisfies the precondition will never result in a `SecurityException`. In other words, clients can provably rely on the formal method contract. If, for instance, the developer would leave out the precondition on the `DoSensitive()` method, verification would fail.

After having verified the correctness of `LibraryClass`, we can write a client for it. The class `ClientClass` of Figure 6 is a client of `LibraryClass`: it calls methods of the library in its implementation.

For client code, we cannot (always) expect developers to write permission-preconditions. We assume they just specify the requested minimum permission set for each assembly, using assembly level declarative security attributes. Each method in the assembly then gets an (overly conservative) precondition that requires this declared minimum permission set. The `PermissionSetAttribute` for `ClientClass` indicates that the developer expects that its code can potentially be executed without any static permission (except for the permission to execute, which we ignore for this example). So, for `ClientClass` methods, permission-preconditions default to `true` (i.e. no conditions on `s`). Therefore, anyone can call `ClientClass`'s methods without needing to hold `XPermission`.

```
class ClientClass{
 LibraryClass! t;
 void m1(bool s)
   requires true;
 {
   s = s && StaticPerm();
   t.DoSensitive(5, s);
 }
 void m2(bool s)
   requires true;
 {
   s = s && StaticPerm();
   t.SafeDoSensitive(s);
 }
 static bool StaticPerm()
   ensures result == false;
 {
   return false;
 }
}
```

Figure 7: ClientClass after transformation

After (automatically) adding preconditions, the program transformation described in Figure 4 is applied to `ClientClass`. The result of this transformation is shown in Figure 7. Note that `StaticPerm` returns false this time because the static permission set of `ClientClass` does not contain `XPermission`.

The transformed program and the specification of `LibraryClass` (a referenced assembly) are then "fed" to Boogie:

- The static verifier detects that `m1` violates the precondition of `DoSensitive`. This indicates a `SecurityException` might be thrown as part of the execution of `m1` (where a method execution includes nested method executions).
- The static verifier proves that `m2` will never raise a `SecurityException` because it does not violate a precondition or assert.

Extending the Basic Idea

In the previous section we discussed the basic ideas behind our approach. However, we considered only programs using a single, atomic permission. In this section we show how programs using multiple, parameterized permissions can be verified.

```
SPS(m(a1,…,an){Body}) ≡                    (1')
        m(a1,…,an, PermissionSet! s){
          s = s.Intersect(StaticPerm());
          SPS(Body)
        }
SPS(o.m(x1,…,xn);) ≡                       (2')
        o.m(x1,…,xn, s.Copy());
SPS(p.Demand();) ≡                         (3')
        assert SPS(allows(s,p));
SPS(p.Assert();) ≡                         (4')
        assert SPS(allows(StaticPerm(),p);
        s = s.AddPermission(p);
SPS(allows(s,p)) ≡                         (5)
        p.IsSubsetOf(
           s.GetPermission(p.GetType()));
```

Figure 8: (SPS) program transformation- revised

When considering programs using multiple permissions, a dynamic permission set can no longer be represented by a Boolean variable. Instead we will represent dynamic permission sets by objects of the class `PermissionSet`[4]. This modification makes the rules for program transformation a bit more complex: instead of manipulating simple boolean variables, we now have to interact with dynamic permission sets by means of `PermissionSet` methods (see Figure 8).

[4] The class `PermissionSet` used in this paper differs slightly from the one in the BCL in order to make it more amenable to static verification. The details of the differences are irrelevant for this paper, and hence are not discussed.

We illustrate the extended approach using the trusted library method `ReadUri` of Figure 9. This method creates a stream to read from a given universal resource identifier (uri). Firstly, notice that the parameter uri determines which permissions are required: if the uri refers to a file, we need permission to access the file system; if it refers to a website, we need permission to access the web. Using preconditions, we can clearly state this in the interface of the method. Secondly, our approach supports permissions with parameters, given their precise specification.

```
public Stream ReadUri(Uri! uri)
  requires uri.Scheme == "file" ==>
           allows(s, newFileIOPermission(
             uri.AbsolutePath));
  requires uri.Scheme == "http" ==>
          allows(s,
            newWebPermission(uri.Host));
 {
   String p = uri.AbsolutePath;
   String h = uri.Host;
   Stream stream = null;
   if(uri.Scheme == "file"){
     stream =  File.Open(p);
   }
   if(uri.Scheme == "http"){
     new WebPermission(h).Demand();
     new SocketPermission(h,80).Assert();
     Socket socket = new Socket(h, 80);
     stream = new NetworkStream(socket);
   }
   return stream;
}
```

Figure 9: Method ReadUri

In general, to verify a method, the verifier needs a precise specification of `PermissionSet` and of all involved permissions, in particular the constructor and the methods `Equals`, `Intersect`, `Union` and `IsSubsetOf` need to be carefully specified for each permission type. In the appendices we give detailed specifications for `PermissionSet` and for a permission class. Furthermore, we show what `ReadUri` looks like after program transformation in appendix C.

5. DISCUSSION AND FUTURE WORK

Our system partially addresses the first three disadvantages of CAS discussed in section 3.

If static verification of an assembly succeeds, run time checks can be turned off, improving performance.

By making security requirements explicit as preconditions, formal documentation for the CAS related behavior of methods is provided, and if the method verifies, one can be sure that the documentation is correct in the sense that if the client security context satisfies the precondition, there will definitely be no security exceptions.

The declarative nature of the preconditions makes it easier to understand what a library actually enforces: one does not need to look at the implementation to understand the security requirements of a method.

Hence we believe the proposed system is valuable as it stands. Still, we envisage a number of adaptations and extensions that have not yet been explored completely, and will be the subject of future work.

Supporting history based access control

To deal with the fourth disadvantage listed in section 3, our system could be adapted to verify history based access control [Aba03a] instead of standard stack inspection. To support history based access control, the SPS transformation needs small changes, and methods do not only need preconditions on the security context, but also postconditions: every method might potentially influence the dynamic permission set even after it has returned. It is not clear to us yet whether this additional annotation overhead would be workable in practice.

Trading off annotation overhead for precision

Our system supports a tradeoff in annotation overhead versus precision of the analysis. A library developer has to annotate methods with preconditions, but the weakest precondition that guarantees that no security exceptions will be thrown can be complex to write and will in general not be computable automatically.

By writing stronger but simpler preconditions soundness is maintained, but some valid programs might be rejected. Finding the right balance between complexity of annotations and precision of the analysis can only be done by building up practical experience.

Reducing annotation overhead by inferring preconditions

While computing the weakest precondition that ensures no security exceptions will be thrown is infeasible in general, in many cases it is actually quite easy.

An analysis of the use of CAS in the Rotor BCL shows that most occurrences of permission demands are instances of the following pattern: a method validates parameters, creates an appropriate permission possibly based on method parameters, demands that permission and subsequently asserts sufficient permissions to make sure the rest of the method will not throw further security exceptions. For methods that follow this pattern, inferring an appropriate precondition automatically is fairly easy. In particular, if the demand is specified declaratively (40% of the demands of the Rotor BCL are declarative), inferring the corresponding precondition is trivial. So there is hope that annotation overhead can be kept small.

The hardest cases are probably methods that do not themselves demand or assert permissions, but instead call other methods that do so.

A full assessment of the feasibility of inferring preconditions is future work.

6. RELATED WORK

Static analysis of stack inspection has been discussed extensively in the literature.

Pottier, Skalka and Smith [Pot01a] developed a security typing system and showed that in a type-safe program, no demand ever fails at run-time. Our preconditions are more expressive, and consequently less conservative, than their typing system. As opposed to Pottier, our analysis is path-sensitive. For instance, for

```
if(i+j != j+i){
   new DnsPermission().Demand();
}
```

Pottier requires `DnsPermission` to be in the dynamic permission set before execution of the example, whereas we do not.

A second difference is that [Pot01a] considers permissions to be atomic: a piece of code either has the permission (PermissionState.Unrestricted), or does not have the permission at all (PermissionState.None). For some types of permissions, such as FileIOPermission, this is too restrictive. Our approach can handle parameterized permissions. For instance, consider the following example:

```
new FileIOPermission("/tmp");
```

Our approach allows client code that only has permission to access to the temporary directory, to call methods containing this statement. Atomic-permission approaches would reject such programs.

However, the increased expressiveness of our approach comes at a price: [Pot01a] can algorithmically infer the type of each method, while we require programmers to write preconditions. Moreover, to benefit from the path sensitivity of our

approach, one potentially needs specification and verification of the functional correctness of code on the path to a permission demand. For now, we reduce the annotation overhead by using sensible defaults. In the future, we hope to find a way to automatically infer or safely approximate these preconditions.

In [Bes04a], Besson, Blanc, Fournet and Gordon propose a technique for analyzing the security of libraries for systems that rely on stack inspection for access control. Their tool generates a permission-sensitive call graph, given a library and a description of the permissions granted to unknown client code. This graph can then be queried to detect anomalous or defective control flow in the library.

Bartoletti, Degano and Ferrari [Bar01a] use safe approximations of the permissions granted/denied to code at run time to reduce some of the overhead due to stack inspection. Their analysis requires the entire program as input; it cannot handle virtual calls to unknown code.

Koved, Pistoia and Kershenbaum [Kov02a] present a technique for computing the set of required access rights at each program point. Their technique uses a context sensitive, flow sensitive, interprocedural data flow analysis. We are currently investigating this technique for automatically inferring the permission-preconditions at each program point. However, because of path insensitivity, this technique is overly conservative.

The program transformation described in this paper is based on the Security-passing Style transformation first proposed by Wallach. In [Wal00a], Wallach explains how the performance of stack inspection can be improved using this transformation.

7. CONCLUSION

This paper proposes a system for static verification of compliance to a Code Access Security policy. It relies on expressive method contracts to specify the dynamic permission set that a method requires the caller to have in order to execute without security exceptions.

The system supports modular verification of methods annotated with such contracts. Verification of such a single method is useful in the context of library development, and ensures consistency of the contract with the implementation of the method, essentially showing that the (formal) documentation of security related behavior of the method is correct.

If all assemblies that make up a program verify, one can be sure there will be no security exceptions, and hence run time stack inspection can be turned off.

8. ACKNOWLEDGMENTS

Bart Jacobs is a Research Assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.-Vlaanderen).

The authors would like to thank Wolfram Schulte for his comments and feedback on a draft of this paper.

We would also like to thank the reviewers for their useful comments and feedback.

9. REFERENCES

[Aba03a] Abadi, M., Fournet, C. Access Control Based on Execution History. NDSS, pp. 6-7, 2003.

[Bar01a] Bartoletti, M., Pierpalo, D. and Ferrari, G. Static Analysis for Stack Inspection. in Elsevier Science B.V., 2001.

[Bar04a] Barnett, M., Leino, K.R.M. and Schulte,W. The Spec# Programming System: An Overview. Microsoft Research, 2004.

[Bes04a] Besson, F., Blanc, T., Fournet, C. and Gordon, A.D. From Stack Inspection to Access Control: A Security Analysis for Libraries. in proc. 17th IEEE Computer Security Foundations Workshop, pp. 61-75, 2004.

[Fou02a] Fournet, C. and Gordon A.D. Stack Inspection: theory and variants. Symposium on Principles of Programming Languages, 2002.

[Fre03a] Freeman, A. and Jones, A. *Programming .NET Security*, O'Reilly 2003.

[Gon02a] Gong, L. Java™ 2Platform Security Architecture. 2002.

[Kov02a] Koved, L., Pistoia, M. and Kershenbaum, A. Access Rights Analysis for Java. 2002.

[Pot01a] Pottier, F., Skalka, C., Smith, S. A Systematic Approach to Static Access Control. in proc. of 10th European Symposium on Programming, pp. 30-45, 2001.

[Stu03a] Stutz, D., Neward, T and Shilling, G. Shared Source CLI. O'Reilly, 2003.

[Wal00a] Wallach, D.S., Appel, A.W. and Felten, E.W. SAFKASI: A Security Mechanism for Language-based Systems. ACM Transactions on S. E. and M. 9, No. 4, 2000.

Appendix A: PermissionSet

Below, we give the specification of the class `PermissionSet`. The definition given below differs slightly from the one given in the BCL:

- `AddPermission` does not modify `this`, but instead creates a new permission set.
- Intersect does not return null when the intersection is empty. Instead it returns an empty permission set.
- `GetPermission` never returns null. If a permission is not present in the set, `GetPermission` returns a permission with `PermissionState.None`.

In Spec#, non-null types (see [Bar04a]) are denoted by `T!` (where `T` is an ordinary reference type).

```
class PermissionSet{

 public IPermission! GetPermission(Type! t)
   ensures result.GetType() == t;

 public PermissionSet! Intersect(PermissionSet! other)
   ensures Forall {Type! t;
                   result.GetPermission(t).Equals(
                   this.GetPermission(t).Intersect(other.GetPermission(t)))
                  };

 public PermissionSet! AddPermission(IPermission! p)
   ensures Forall {Type! t;
                   (t != p.GetType())
                     ==>
                    result.GetPermission(t).Equals(this.GetPermission(t)
                  };
   ensures result.GetPermission(p.GetType()).Equals(
           p.Union(old(GetPermission(p.GetType()))));
}
```

Appendix B: IPermission and SocketPermission

Below, we give the specifications of `IPermission` and of (a simplified version of) `SocketPermission`. The definitions given below differ slightly from the ones given in the BCL:

- Intersect will never return null, not even when the intersection is empty. Instead it will return a permission with `PermissionState.None`.

```
public interface IPermission {
 bool IsSubsetOf(IPermission! other)
   requires other.GetType() == this.GetType();

 IPermission! Intersect(IPermission! other)
   requires other.GetType() == this.GetType();
   ensures result.GetType() == this.GetType();

 IPermission! Union(IPermission! other)
   requires other.GetType() == this.GetType();
   ensures result.GetType() == this.GetType();
}
```

```
public sealed class SocketPermission : IPermission {

 public bool Includes(EndPointPermission p);

 public SocketPermission(PermissionState state)
   ensures state == PermissionState.Unrestricted ==>
           Forall{EndPointPermission! p; Includes(p)};
   ensures state == PermissionState.None ==>
           Forall{EndPointPermission! p; !Includes(p)};

 public SocketPermission(string host, int port)
   ensures Forall{EndPointPermission! p;
           Includes(p) == (p.Host == host && p.Port == port)};

 public bool IsSubsetOf(SocketPermission! other)
   ensures result == Forall{EndPointPermission! p;
           Includes(p) ==> other.Includes(p)};

 public SocketPermission! Intersect(SocketPermission! other)
   ensures Forall{EndPointPermission! p; result.Includes(p) ==
           (this.Includes(p) && other.Includes(p))};

 public SocketPermission! Union(SocketPermission! other)
   ensures Forall{EndPointPermission! p; result.Includes(p) ==
           (this.Includes(p) || other.Includes(p))};

 public bool IsSubsetOf(IPermission! other)
   ensures result == IsSubsetOf((SocketPermission!) other);

 public IPermission! Intersect(IPermission! other)
   ensures result == Intersect((SocketPermission!) other);

 public IPermission! Union(IPermission! other)
   ensures result == Union((SocketPermission!) other);
}
```

Appendix C: ReadUri after (SPS) program transformation

```
class ClassName{

 public Stream ReadUri(Uri! uri, PermissionSet! s)
   requires uri.Scheme == "file" ==>
       new FileIOPermission(uri.AbsolutePath).IsSubsetOf(
        s.GetPermission(new FileIOPermission(uri.AbsolutePath).GetType()));
   requires uri.Scheme == "http" ==>
       new WebPermission(uri.Host).IsSubsetOf(
        s.GetPermission(new WebPermission(uri.Host).GetType()));
 {
   s = s.Intersect(StaticPerm());
   String p = uri.AbsolutePath;
   String h = uri.Host;
   Stream stream = null;
   if(uri.Scheme == "file"){
      stream =  File.Open(p, s.Copy());
   }
   if(uri.Scheme == "http"){
      assert new WebPermission(h).IsSubsetOf(
        s.GetPermission(new WebPermission(h).GetType()));
      assert new SocketPermission(h, 80).IsSubsetOf(
        StaticPerm().GetPermission(new SocketPermission(h, 80).GetType()));
      s = s.AddPermission(new SocketPermission(h, 80));
      Socket socket = new Socket(h, 80, s.Copy());
      stream = new NetworkStream(socket, s.Copy());
   }
   return stream;
 }

 public static PermissionSet StaticPerm()
   //---> for every statically assigned permission p
   ensures p.IsSubsetOf(result.GetPermission(p.GetType()));

}
```