JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 5, No. 3, May–June 2006

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

Jan Smans, Bart Jacobs, Frank Piessens
Department of Computer Science, Katholieke Universiteit Leuven, Belgium

Stack inspection-based sandboxing originated as a security mechanism for safely executing partially trusted code. Today, it is widely used for the more general purpose of supporting the principle of least privilege in component-based software development. In this more general setting, the permissions required by a component to run properly, or the permissions needed by other components to successfully call methods in a given component are conceptually part of the interface specification of the component. Hence, correct documentation of this part of the interface is essential.
In this paper, we propose formal component and method contracts for stack inspection-based sandboxing, and we show that formal verification of these contracts is feasible with state-of-the-art program verification tools. Our contracts are significantly more expressive than existing type systems for stack inspection-based sandboxing.
We describe our solution in the context of the sandboxing mechanism in the .NET Framework, called Code Access Security. Our system relies on the Spec# programming language and its accompanying static verification tool.

1 INTRODUCTION

Stack inspection-based sandboxing originated as a security mechanism for limiting the amount of damage that could be done by dynamically loaded untrusted code. It was designed to support configurable sandboxes for Java applets [1]. Today, it is widely used for the more general purpose of supporting the principle of least privilege in component-based software development. Both C# and Java support composition of applications from components (assemblies for C#, .jar files for Java) where each of these components can be given different access permissions. In scenarios where an application is composed of third-party components coming from different sources, this support for the principle of least privilege has significant security benefits, including increased assurance that bugs in some of the components will not lead to critical security violations.

However, in this setting, the permissions required by a component to run properly, or the permissions needed by other components to successfully call methods in a given component are conceptually part of the interface specification of the component, and correct documentation of this part of the interface is essential. Neither C# nor Java provide adequate support for the specification of these security properties of components. This lack of specification can lead to unexpected security exceptions

Cite this document as follows: Jan Smans, Bart Jacobs, Frank Piessens: *Static Verification of Code Access Security Policy Compliance of .NET Applications*, in Journal of Object Technology, vol. 5, no. 3, May–June 2006, pages 35–58,
http://www.jot.fm/issues/issues_2006_4/article2

at run time [2].

In this paper, we propose formal component and method contracts for stack inspection-based sandboxing, and we show that formal verification of these contracts is feasible with state-of-the-art program verification tools. Our contracts are significantly more expressive than existing type systems for stack inspection-based sandboxing [3].

We will describe our solution in the context of C# and the Microsoft .NET Common Language Runtime, because our implementation builds on the Spec# programming system [4]. But the same approach would be feasible in Java.

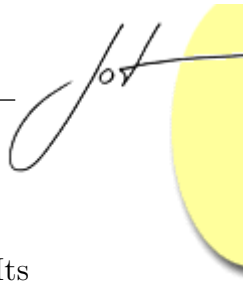
The rest of this paper is structured as follows. In Section 2 we briefly review the Spec# Programming System, Code Access Security, and the security-passing style transformation. In Section 3, we point out the problem we wish to solve in this paper. Next, we present our proposed solution in detail (Section 4) and discuss its advantages and disadvantages (Section 5). Finally, we compare with related work and conclude.

2 BACKGROUND

The Spec# Programming System

The Spec# Programming System [4] consists of three parts: an object-oriented language called Spec#, a compiler, and a program verifier. The language Spec# is an extension of C#. It extends C# with non-null types (for a type T, the corresponding non-null type is denoted by $T!$), checked exceptions, and constructs for writing specifications, such as object invariants and pre- and postconditions 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 Spec# Program Verifier (sometimes referred to as 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. Essentially, the verifier computes a number of first-order logic verification conditions based on an axiomatic semantics of Spec#, and then feeds those verification conditions to a fully automatic theorem prover. The prover can then (1) prove those conditions, showing that implementations are consistent with specifications, or (2) come up with a counterexample showing that there is an error, or (3) give up and provide no information about the correctness of the program (i.e. the verification is incomplete).



Code Access Security

Code Access Security is the sandboxing mechanism of the .NET Framework. Its goal is to limit the access code has to protected resources and operations such that partially trusted code can be executed securely.

Code access rights are defined by means of *permissions*. A permission is a first-class object that represents a right to access certain protected resources or operations. For instance, a *FileIOPermission* object represents the right to perform certain operations (open, create, read, write,...) on certain files.

Permissions are assigned to components (assemblies in C#) 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 a set of permissions. The resulting permission set for a given component is called the *static permission set*.

At run-time, every thread maintains an associated *dynamic permission set* or *security context* that represents the actual access rights that the thread has at this point in its execution. In the CLR, the dynamic permission set is not represented explicitly, but is computed by stack inspection whenever necessary: 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.

- Calling *Demand* on a permission object p, checks whether p is in the dynamic permission set. This operation initiates a stack inspection: all frames on the stack (from top to bottom) are checked for permissions p. If a frame is encountered that doesn't have this permission in its static permission set, a *SecurityException* is thrown. Otherwise, *Demand* simply returns normally, without any side-effects. This method is used by libraries to guard sensitive operations from being accessed by unprivileged code.

- Calling *Assert* on a permission object p, marks the current (call) stack frame as "privileged for permission p". If such a frame is encountered during stack inspection for permission p, *Demand* returns normally, without checking any frames deeper down the call stack. Thus, asserting a permission effectively makes the thread's dynamic permission set grow. This operation is used by highly trusted code to allow less trusted code to access some resource in a well-defined, secure way.

An analysis we performed of the Rotor BCL [5], a partial, shared source implementation of the Base Class Library, has shown that other operations on permission objects, such as *Deny* and *PermitOnly*, occur only rarely. Therefore, we omit them in this paper.

Next to the imperative syntax discussed above, Code Access Security also supports a declarative security syntax. This declarative syntax uses .NET attributes to

place security information within an assembly's metadata. For example, by placing an attribute on a member, a programmer can instruct the run-time system to invoke *Demand* on a (statically) given permission before executing the member.

Security-passing Style Transformation

Wallach's security-passing style (SPS) transformation [6] can be used to define the semantics of Code Access Security (and of stack inspection-based sandboxing in general) by translating the primitive operations, such as *Demand* and *Assert*, to the host language. The original program using *Demand* and *Assert* is transformed in the following way (see also Figure 1):

$$
\begin{array}{rcl}
SPS(m(x_1,...,x_n)\ \{Body\}) & \equiv & m(x_1,...,x_n,\ s)\ \{ \\
 & & \quad s := s \cap static\_permissions; \\
 & & \quad SPS(Body) \\
 & & \} \\
SPS(o.m(a_1,...,a_n)) & \equiv & o.m(a_1,...,a_n,\ s) \\
SPS(p.Demand();) & \equiv & \textbf{if}\ (p \notin s)\ \{ \\
 & & \quad \textbf{throw new}\ SecurityException(); \\
 & & \} \\
SPS(p.Assert();) & \equiv & \textbf{if}\ (p \notin static\_permissions)\ \{ \\
 & & \quad \textbf{throw new}\ SecurityException(); \\
 & & \} \\
 & & s := s \cup \{p\};
\end{array}
$$

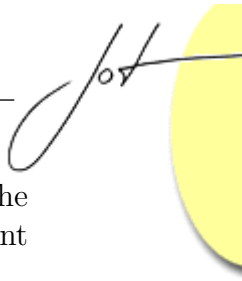
Figure 1: The security-passing style transformation

1. An explicit representation of the security context is chosen. For our purposes, the security context is best represented as a set of permissions.

2. Each method gets an additional parameter representing the caller's dynamic permission set. Every method invocation is updated correspondingly.

3. At the start of each method body, code is added that updates the current security context to be the intersection of the security context of the caller and the static permission set of the callee.

4. The primitive $p.Demand()$ is transformed into code that checks whether p is in the current security context and throws a *SecurityException* if it is not.



5. The primitive $p.Assert()$ is transformed into code that (1) checks if p is in the static permission set of the code calling $Assert()$, and (2) updates the current security context by adding p.

In this paper, we use this transformation to define the semantics of Code Access Security. The advantage of this approach is that these semantics are defined entirely in terms of the semantics of the host programming language. As a consequence, we can use the existing Spec# Program Verifier without it having to be aware of stack inspection. Examples of an SPS transformation are provided later on in the paper.

3 PROBLEM STATEMENT

Problems of Sandboxing based on Stack Inspection

While stack inspection-based sandboxing is a usable and essential part of the security infrastructure offered by some state-of-the-art platforms (including .NET and Java), it has a number of well-known shortcomings. These can be summarized as follows:

1. Security operations, such as *Demand* and *Assert*, are typically part of the implementation of a component and as such, their effect is not visible in the interface of the component (method signatures etc): the (informal) documentation has to specify under what circumstances *SecurityException*s will be thrown. Writing and maintaining precise documentation is error-prone.

   In the .NET Framework, declarative security *Demand*s partly deal with this problem. However, these declarative *Demand*s do not have the same expressive power as imperative *Demand*s, and our analysis of the Rotor BCL has shown that approximately 60% of all *Demand*s are imperative.

2. Not only are security operations part of the implementation, they are scattered throughout the BCL. Our analysis of the Rotor BCL found 183 *Demand*s scattered across 40 classes. This makes it very hard to see what policy the stack inspection mechanism actually enforces.

3. Stack inspection-based sandboxing is implemented using dynamic checks, which can have a substantial impact on performance. Moreover, as it relies on the presence of activation records on the call stack, this security mechanism can hinder optimizations that affect the stack.

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 untrusted code, or exceptions thrown from such code.

Many researches have recognized these shortcomings of sandboxing based on stack inspection, and have proposed partial solutions [3, 10, 6, 14, 12]. We discuss these in Section 6.

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 Section 6, we also briefly indicate how our solution could be extended to deal with the last disadvantage.

4 PROPOSED SOLUTION

In this paper, we define a formal and expressive specification formalism for the .NET Code Access Security system, together with an approach for verifying whether a component complies with these specifications.

To verify a component C (i.e. verify whether it could ever throw a *SecurityException*), we need (1) C's implementation, (2) C's specification and (3) the specifications of all referenced components. Our verification then determines whether execution of the given component could ever cause a *Demand* to fail, given an environment that respects specifications. Our approach is sound but incomplete. In

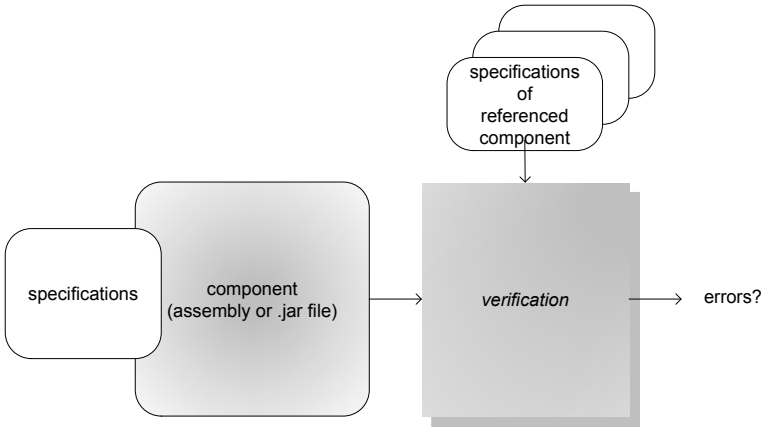


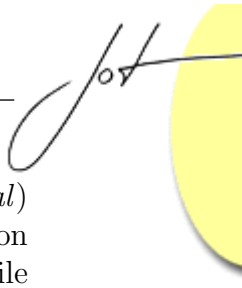
Figure 2: Overview of our solution

order to be useful, it requires developers to write specifications and hence introduces annotation overhead.

Component and method contracts

Two types of contracts will be used to specify the "sandboxing behavior" (i.e. behavior w.r.t. CAS) of components.

For the first type of contracts, *component-level contracts*, we rely on .NET's existing assembly-level attributes for requesting permissions. Using these attributes,



a component developer can define a *minimal permission set* (via *RequestMinimal*) and an *optional permission set* (via *RequestOptional*). The minimal permission set defines a lower bound on the component's run-time static permissions, while the optional permission set declares additional permissions that, if granted to the component, enable additional functionality. A component will never need more permissions than those declared in the minimal and the optional permission sets. For example, a computer game can operate only when given the permission to open windows (minimal permission set). But when the game also has permission to access the file system (an optional permission), it can store high scores. In the .NET Framework these assembly-level attributes are only used at load-time to check whether the statically assigned permission set is valid according to these attributes. Our approach however relies on them for providing compile-time guarantees about the absence of unexpected run time *SecurityException*s. Since the .NET syntax for writing these assembly-level attributes is rather verbose, we will use slightly simpler syntax in our examples.

The second type of contract, *method-level contracts*, are basically Spec# preconditions that specify on a per-method basis what dynamic permission sets are expected by the method. If callers adhere to this specification, the method will not throw any *SecurityException*s. For example, a *Connect* method that loads a given URI may specify as a precondition that the security context must include *FileIOPermission* if the URI points to a file and *SocketPermission* if the URI points to a network resource. Note that these method-level contracts are essentially plain Spec# preconditions where an additional variable s, referring to the caller's dynamic permission set, can be mentioned.

Writing component-level contracts does not incur substantial overhead, and we believe it is reasonable to assume that every component will be specified with such a contract. Method-level contracts on the other hand incur a substantial annotation overhead, and it is probably not realistic to assume that all components will be fully annotated with method contracts. However, for client code (i.e. code that is not intended to be called from other components) it is easy to derive from the component contract safe default method contracts: every method simply requires that the security context includes the minimal permission set. For library code (i.e. code that is intended to be called from other components), the annotation overhead is probably acceptable: it is merely a precise documentation of what clients of the code need to know about the sandboxing-related behavior of each library method.

Note that the component-level contract is a contract between a component and its deployer, while the method-level contract is a contract between the component and its callers. The deployer of a component should make sure the static permissions of a component are a superset of the minimal permission set requested in the component-level contract. When calling methods, clients should abide by the method-level contract of the callee, i.e. at each call-site, the dynamic permission set should satisfy the method-level contract of the callee.

Verification of Sandboxing Contracts

Successful verification of a component in our approach guarantees that the component will not throw any *SecurityException*s when (1) the minimal permission set described by the component-level contract is a subset of the static permissions that are assigned to it by the security policy, (2) it is only called under dynamic permission sets that satisfy its method-level contracts and (3) all external methods that are called by the verified component respect their contracts[1]. Informally, successful verification means "if the environment behaves according to its specification, the component will not throw any *SecurityException*s". For the verification of a component, we need (1) the component itself, (2) its component-level and method-level contracts and (3) the method-level contracts of all referenced components. Verification then proceeds in two steps:

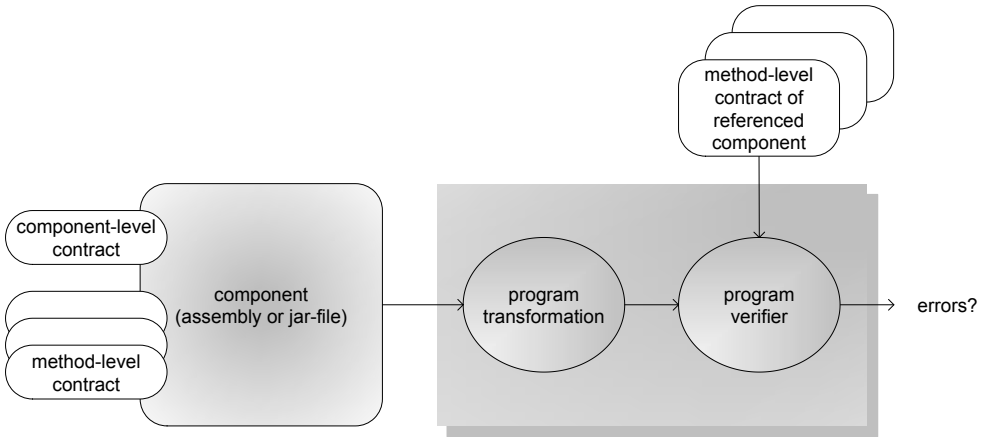


Figure 3: Detailed overview of our solution

1. The verification tool applies a security-passing style transformation to the component. As a result of this transformation, all sandboxing primitives are transformed into expressions of the host language (in our case Spec#): execution of the transformed component on a CAS-unaware VM would be equivalent to executing the original component on a CAS-aware VM. Thus, the result of the SPS-transformation is ordinary Spec# code (i.e. code without any calls to security operations such as *Demand* or *Assert*).

2. In the second step the transformed component is input to the Spec# Program Verifier. The latter need not be able to reason about any sandboxing primitives, since these were all removed by the transformation. If verification succeeds, the semantics of the SPS-transformation guarantees that in the original component *Demand*s will not fail (if the environment behaves appropriately) and as a consequence, that it will not throw any unexpected *SecurityException*s.

[1] How we deal with optional permission sets is discussed later in this section.

Illustrating the Basic Idea

This first example is meant to show the basic idea behind our approach. To keep it as clear and simple as possible, we make some assumptions about the programs we consider. First of all, we assume only one permission type is used, namely *WebPermission*. Secondly, we do not consider permissions that take parameters, so *WebPermission* objects have no parameters. Under these assumptions, a set of permissions is either the empty set or the singleton containing only *WebPermission*. Moreover, we do not consider optional permission sets for this example. Our example involves two components: a library component and a client component.

The Library Component

The library component offers a class called *Connector*, which is shown in Figure 4. The component-level contract (first line of 4) specifies that it requires *WebPermission* in its minimal permission set, i.e. the component cannot operate without having *WebPermission* in its static permission set.

```
[Minimum := {WebPermission}]
class Connector {
  Stream Connect(String url)
    requires WebPermission ∈ s;
  {

    new WebPermission().Demand();
    //create the connection
  }

  Stream ConnectToTrusted()
    requires true;
  {

    new WebPermission().Assert();

    return Connect(''t.com'');
  }
}
```

Figure 4: *Connector* before SPS

```
class Connector {
  Stream Connect(String url, bool s)
    requires WebPermission ∈ s;
  {
    s := s ∩ StaticPermSet();
    assert WebPermission ∈ s;
    //create the connection
  }

  Stream ConnectToTrusted(bool s)
    requires true;
  {
    s := s ∩ StaticPermSet();
    assert WebPermission ∈ StaticPermSet();
    s := s ∪ {WebPermission};
    return Connect(''t.com'', s);
  }
}
```

Figure 5: *Connector* after SPS

The class offers two methods: *Connect* and *ConnectToTrusted*. The former method offers clients the possibility to connect to arbitrary URLs but guards this

(sensitive) operation by an access control check. The latter method, *ConnectToTrusted*, allows any client to make connections, but only to the trusted site `t.com`. Both methods specify method-level contracts. The contract of *Connect* specifies that the caller's dynamic permission set (denoted by the variable s) should contain *WebPermission* in order to prevent *SecurityException*s. The contract of *ConnectToTrusted* requires **true**, which means that this method can be called by any client in any context.

To verify the correctness of *Connector* versus its specification, we first apply an SPS transformation. The result of this transformation is shown in Figure 5.

- An extra parameter s, representing the caller's dynamic permission set, is added to every method declaration and is inserted at every call site. We use booleans to represent these sets: **false** represents {}, **true** represents {*WebPermission*}.
- A per-component, static method *StaticPermSet* is introduced (omitted from Figure 5). The postcondition of *StaticPermSet* specifies that it returns a permission set which is equal to or larger than the minimal permission set as described by the component-level contract. For this example, the contract of *StaticPermSet* specifies it returns a superset of {*WebPermission*}, because the minimal permission set contains *WebPermission*.
- At the beginning of every method the dynamic permission set is updated to reflect the addition of a new activation record to the call stack. The new dynamic permission set is the intersection of the old dynamic permission set and the set of static permissions assigned to the callee.

  Note that we do not have to revert the dynamic permission set to its former value when the method returns, since only a copy of the dynamic permission set is passed on at each call site.
- Every *p.Demand* is transformed into a Spec# **assert** statement[2]. Every **assert** statement implies a proof obligation for the program verifier which we invoke on the transformed program. In this case the verifier has to prove that s will always contain *WebPermission*.
- Each *p.Assert* is transformed into two operations. First of all, because *Assert* can only be applied to a permission p when p is in the caller's (i.e. the caller of *Assert*) static permission set, we add a proof obligation for the program verifier: the static permission set should contain *WebPermission* (this can easily be proven since *StaticPermSet* returns a superset of {*WebPermission*}). Secondly, *Assert* adds the given permission to the dynamic permission set, so s is updated accordingly.

[2]Keep in mind the difference between *Assert* and **assert**. *Assert* is a method which can be invoked on permission objects, while **assert** is a Spec# program statement, declaring that some property always holds at a certain point during execution and implying a proof obligation for the program verifier.

After applying the transformation, we input the transformed program to the Spec# program verifier. This program verifier checks (amongst other things) that preconditions hold at every call site and that every **assert** statement will succeed at run time. If the transformed program verifies, the original program will not throw any *SecurityException*s when executed within an environment that complies with all given specifications.

The Client Component

The client component contains a class called *ConnectorClient*, shown in Figure 6. *ConnectorClient* is a client of *Connector* because the former class calls methods of the latter. The component-level contract for the client component specifies that it can operate without having any static permissions. Note that we have intentionally chosen a faulty example to show how verification can detect errors in components. *ConnectorClient* offers two methods: *Spy* and *SendDataToTrusted*. The former

```
[Minimum := {}]
class ConnectorClient {
  Connector! c;
  void Spy()

  {

    Stream st := c.Connect(``s.com'');
    //send confidential data
  }

  void SendDataToTrusted()

  {

    Stream st := c.ConnectToTrusted();
    //send data to trusted host
  }
}
```

Figure 6: client before SPS

```
class ConnectorClient {
  Connector! c;
  void Spy(bool s)
    requires {} ⊆ s;
  {
    s := s ∩ StaticPermSet();
    Stream st := c.Connect(``s.com'', s);
    //send confidential data
  }

  void SendDataToTrusted(bool s)
    requires {} ⊆ s;
  {
    s := s ∩ StaticPermSet();
    Stream st := c.ConnectToTrusted(s);
    //send data to trusted host
  }
}
```

Figure 7: client after SPS

method tries to send confidential data to a spyware server (something we wish to prevent and detect statically), while the latter sends data to `t.com`, a trusted host. Only trusted code should be able to send (possibly confidential) data to arbitrary hosts. Observe that both methods make use of a *Connector* object.

For client components, we cannot (always) expect developers to write method-level contracts. Instead, we simply infer (overly conservative) method contracts from the component-level contract. More specifically, each method gets a precondition requiring the caller's dynamic permission set to include the declared minimum permission set. Because the minimum permission set for the client component of our example is the empty set, the precondition requires the caller's dynamic permission set to be at least the empty set, which is equivalent to saying that the precondition requires nothing.

Again, to verify *ConnectorClient*, we first apply an SPS transformation to it. The result of this transformation is shown in Figure 7. Note that for the client component the postcondition of *StaticPermSet* states that the method can return any superset of the empty set (i.e. any permission set). After applying the transformation, we input the resulting program to the Spec# Program Verifier. Its output is as follows:

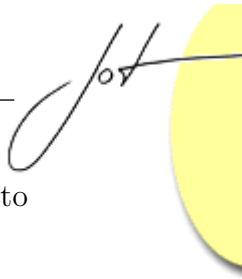
- The verifier detects that *Spy* violates the precondition of *Connect*. This indicates that a *SecurityException* may be thrown by *Spy*. The component-level contract is inconsistent with the implementation of *Spy*.
- The static verifier proves that *SendDataToTrusted* will never throw a *SecurityException* because it does not violate a precondition or **assert**.

Optional Permission Sets and the *Test* Operation

Besides a minimal required permission set, a component-level contract may also define an optional permission set. In doing so, the component declares that it does not require any optional permissions in order to execute, but it can offer additional functionality when granted these additional permissions.

In order to benefit from additional (optional) permissions, components should be able to check at run time which access rights they (or more precisely, their threads) currently have. In other words, they should be able to query the dynamic permission set to verify whether it contains a certain permission. In the existing Code Access Security system, *SecurityException*s are considered to be normal exceptions. Hence, applications typically check for optional permissions by tentatively performing the sensitive operation within a try-catch-block. The solution we propose however considers *SecurityException*s to be program errors, so tentatively performing an operation that may fail is ruled out. As such, our approach is not entirely backward compatible because existing programs that test for optional permissions by tentatively performing the operation are considered invalid in our approach.

To enable components to query the dynamic permission set in our solution, we propose introducing a new security operation called *Test*. Invoking *Test* on a permission p returns **true** when p is part of the dynamic permission set; otherwise it returns **false**. A sensitive operation which requires some optional permission p, will then typically be placed within an **if** statement with condition $p.Test()$. The



introduction of a new security operation also implies adding an additional rule to the SPS transformation:

`p.Test()` $\equiv$ `p` $\in$ `s`

As a sanity check for *Test*, we could verify that only permissions included in the union of the minimal and optional permission set are tested.

We illustrate the use of optional permission sets and the Test-operation by means of another example. Consider a new library component containing a single class called *ConnectorExt* (see Figure 8), which is an extension of the *ConnectorLibrary* of Figure 4. The component-level contract indicates that it can execute without

```
[Minimum := {}]
[Optional = {WebPermission}]
class ConnectorExt : Connector
{
  Stream TryConnect(String url)


  {

    if (new WebPermission().Test()) {
      return base.Connect(url);
    }
    return null;
  }
}
```

Figure 8: *LibraryExt* before SPS

```
class ConnectorExt : Connector
{
  Stream TryConnect(String url,
                    bool s)
    requires {} ⊆ s;
  {
    s := s ∩ StaticPermSet();
    if (WebPermission ∈ s) {
      return base.Connect(url, s);
    }
    return null;
  }
}
```

Figure 9: *LibraryExt* after SPS

having any permissions (minimal permission set), but that it may be able to offer additional functionality when given the *WebPermission* (optional permission set).

ConnectorExt extends its superclass with a single method called *TryConnect*. If sufficient permissions are present in the dynamic permission, this method will return a connection to the given URL; otherwise, it will return **null**. For checking whether the required *WebPermission* is present, the implementation relies on the *Test* primitive. Verification of this component will succeed, because the verifier can assume the dynamic permission set contains *WebPermission* in the body of the **if** statement.

A Full Example

For programs using a single, atomic permission, it suffices to represent permission sets by means of boolean variables. However, when considering programs with multiple, parameterized permissions, (1) a more expressive encoding for permission sets has to be chosen and (2) a precise specification of each parameterized permission is needed, in order for the program verifier to be able to reason about them. Our approach is as follows:

- The .NET Framework offers a class *PermissionSet* whose objects represent sets of permissions. We will rely on a slightly modified version of *PermissionSet* (see Figure 12) to represent static and dynamic permission sets.
- Every permission class in the .NET Framework must implement interface *IPermission* (see Figure 11). In order for static verification to be possible, the methods *IsSubsetOf*, *Union* and *Intersect* defined in this interface and the method *Equals* inherited from *Object*, must be precisely specified for each permission type. As for *PermissionSet*, we rely on a slightly modified definition of .NET's *IPermission* (see Figure 11).

Figure 10 shows the SPS transformation for our approach, defined in terms of methods on *IPermission* and *PermissionSet* objects.

$$
\begin{array}{rcl}
SPS(m(x_1,...,x_n)\{Body\}) & = & m(x_1,...,x_n,\ s)\ \{ \\
 & & \quad s := s.Intersect(\ StaticPermSet()\ ); \\
 & & \quad SPS(Body) \\
 & & \} \\
SPS(o.m(a_1,...,a_n)) & = & o.m(a_1,...,a_n,\ s.Copy()) \\
SPS(p.Demand();) & = & \textbf{assert}\ s.Contains(p); \\
SPS(p.Assert();) & = & \textbf{assert}\ StaticPermSet().Contains(p); \\
 & & s := s.AddPermission(p); \\
SPS(p.Test()) & = & s.Contains(p) \\
 & ... &
\end{array}
$$

Figure 10: The security-passing style transformation, defined in terms of *IPermission* and *PermissionSet* methods

In the remainder of this section, we define two library components: *ToasterServices.dll* and *ToasterUtil.dll*. The former component contains a class *Toaster*, offering toasting services to clients and a permission class *ToasterPermission*, which is used by *Toaster* to prevent its sensitive operations from being used by untrusted components. The latter component, *ToasterUtil.dll*, is a client of the former.

Note that the example used in this section is based on one described in [7].

```
interface IPermission {
  [Pure] bool IsSubsetOf(IPermission! other)
    requires other.GetType() = GetType();

  [Pure] IPermission! Intersect(IPermission! other)
    requires other.GetType() = GetType();
    ensures result.GetType() = GetType();

  [Pure] IPermission! Union(IPermission! other)
    requires other.GetType() = GetType();
    ensures result.GetType() = GetType();
}
```

Figure 11: The interface *IPermission* (partial)

```
class PermissionSet {
  [Pure] bool ContainsPermissionOf(Type! t);

  [Pure] IPermission! GetPermission(Type! t);
    requires this.ContainsPermissionOf(t);
    ensures result.GetType() = t;

  [Pure] bool IsSubsetOf(PermissionSet! other);
    ensures result =
        forall{Type! t; this.ContainsPermissionOf(t) ⟹
        other.ContainsPermissionOf(t) ∧
        this.GetPermission(t).Equals(other.GetPermission(t))};

}
```

Figure 12: The interface of class *PermissionSet* (partial)

ToasterPermission

Figure 13 shows (part of[3]) the interface of class *ToasterPermission*. Its objects represent the right to perform certain operations on toasters. Conceptually, every

[3]The specifications shown in this paper are only partial and are just meant to give a flavor of what the full specifications look like. We refer to our website [9] for the full specifications.

ToasterPermission has two flags (boolean parameters): *information* and *eject*. The former flag indicates whether the permission entails the right to retrieve information about toasters, while the latter determines whether it allows for the ejection of bread. In addition, it also has a numeric parameter, *toastLevel*, ranging from *Uncooked* to *Burnt* and indicating the maximum level of toastedness at which bread may be ejected. Notice how the precise behavior of the constructor and the methods *IsSubsetOf*, *Intersect*, *Union* and *Equals* has been documented by means of Spec#-postconditions. In addition, *IsSubsetOf*, *Intersect* and *Union* also inherit the specifications defined at the level of *IPermission*.

```
class ToasterPermission : IPermission{
  [Pure] bool Information();
  [Pure] bool Eject();
  [Pure] Level ToastLevel();

  [Pure] ToasterPermission(bool information, bool eject, Level level)
    ensures this.Information() = information;
    ensures this.Eject() = eject;
    ensures this.ToastLevel() = level;

  [Pure] bool IsSubsetOf(IPermission! other)
    ensures result =
        (Information() ⟹ ((ToasterPermission)other).Information() ∧
        Eject() ⟹ ((ToasterPermission)other).Eject() ∧
        ToastLevel() ≤ ((ToasterPermission)other).ToastLevel());
}
```

Figure 13: The class *ToasterPermission* (partial) defined in *ToasterServices.dll*

Note that it is important for the soundness of our approach that (like for other classes) the implementation of *ToasterPermission* complies with its specification. This is verified as part of verification of *ToasterServices.dll*. In general, before relying on a certain permission type, an administrator should check whether the given permission complies with the intented semantics for permissions.

Toaster

In addition to a permission class, *ToasterServices.dll* also contains the class *Toaster* (see Figure 14). The component-level contract for *ToasterServices.dll* specifies that the library itself should have full access to toasting operations.

Because only highly trusted code (i.e. code having the appropriate *ToasterPermissions*) should be allowed to interact with *Toaster* objects, its methods are guarded by an access control check, i.e. the invocation of a *Demand* on a *ToasterPermission* object.

```
[assembly : Minimum := {new ToasterPermission(true, true, Level.Burnt)}]
class Toaster {
  bool ContainsBread()
    requires s.Contains(new ToasterPermission(true, false, Level.Uncooked));
  {
    new ToasterPermission(true, false, Level.Uncooked).Demand();
    ...
  }

  void EjectBread(Level level)
    requires s.Contains(new ToasterPermission(false, true, level));
  {
    new ToasterPermission(false, true, level).Demand();
    ...
  }
}
```

Figure 14: The class *Toaster* (partial) defined in *ToasterServices.dll*

The permissions required by the methods of class *Toaster* in order to complete successfully are apparent from their contracts. For example, the method-level contract of *ContainsBread* clearly states that this method should not be invoked without permission to retrieve toaster information (*ToasterPermission* with the *information* flag set to **true**). Note that the method-level contracts can refer to variables whose value is only determined at run time. For instance, the specification of *EjectBread* mentions the method parameter *level*.

ToasterUtil.dll: a library on top of *ToasterServices.dll*

Finally, we define another library component *ToasterUtil.dll* (a client of *ToasterServices.dll*) with a single class *ToasterClient* (see Figure 15). The component-level contract indicates the static permission set should include full *ToasterPermission* and full *WindowPermission*. Furthermore, it declares that *ToasterUtil.dll* can provide additional functionality (in this case persistent logging) when given *FileIOPermission*.

ToasterClient offers a single method called *PerformOperation*, which performs different actions depending on the value of the parameter *op*. If the requested operation is Display, some information about the toaster is displayed within a window; if the operation is Eject, the bread is ejected from the toaster. Notice that depending on the value of *op*, different dynamic permission sets are required from callers. This is clearly shown in the interface by the conditional preconditions. Using the Test-operation, *PerformOperation* also tries to write log entries whenever possible

```
[assembly : Minimum := {new ToasterPermission(true, true, Level.Burnt),
                        new WindowPermission()}]
[assembly : Optional := {new FileIOPermission()}]
class ToasterClient {
  Toaster! t;

  void PerformOperation(Operation op)
    requires (op = Operation.Display)
        ⟹ s.Contains(new WindowPermission()) ∧
           s.Contains(new ToasterPermission(true, false, Level.Uncooked));
    requires (op = Operation.Eject)
        ⟹ s.Contains(new ToasterPermission(false, true, Level.Burnt));
  {
    if (op = Operation.Display) {
      bool c := toaster.ContainsBread();
      //display info in GUI
    }

    if (op = Operation.Eject) { toaster.Eject(Level.Burnt); }

    if (new FileIOPermission().Test()) { //write a log entry }
  }
}
```

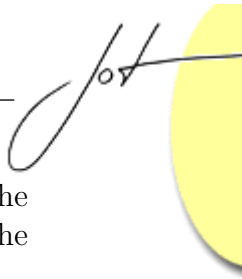
Figure 15: The class *ToasterClient* (partial) defined in *ToasterUtil.dll*

(i.e. whenever *FileIOPermission* is part of the dynamic permission set).

Summary

We introduced two types of sandboxing contracts: *component-level contracts* that specify a minimal and possibly an optional static permission set on the one hand and *method-level contracts* that describe what dynamic permission sets are expected by the implementation of the corresponding method on the other hand.

When a component *complies with its sandboxing contract*, it is guaranteed not to throw any *SecurityException*s, assuming that (1) the minimal permission set described by the component-level contract is a subset of the static permissions that are assigned to it by the security policy, (2) it is only called under dynamic permission sets that satisfy its method-level contracts and (3) all methods that are called by the verified component respect their contracts. Verifying whether a component complies with its sandboxing contract takes two steps. First a security-passing style transformation is applied to the component and secondly this transformed component is



input to the Spec# Program Verifier. If the verifier can prove the correctness of the transformed program, the semantics of the SPS-transformation guarantee that the original component will not throw unexpected *SecurityException*s.

We also proposed a new security operation called *Test*. Using this operation, components can determine whether the dynamic permission set contains a certain permission and adapt their behavior accordingly.

5 DISCUSSION

Experiences with the Spec# Program Verifier

On our website [9], we have a working prototype of our solution and show how some examples (including the ones given in this paper) can be verified.

However, in order to verify some examples, we had to extend the Spec# Program Verifier with (amongst other things) sound support for using pure methods and object creation in specifications as discussed in [8]. On our website, we describe in detail which extensions we made to the static verifier.

The Spec# specification formalism has proven to be very expressive and enabled us to write very precise method level contracts. However, we only verified relatively small examples. Whether verification scales to larger examples, remains an issue for future work.

Applicability to Java

Although our solution has been worked out in the context of the .NET Code Access Security system, we believe it is also applicable to Java stack inspection. The Java machinery for sandboxing differs slightly from the one used in .NET. First of all, different permission-types are independent from each other in .NET, whereas in Java permission objects of different types are not unrelated, as one permission can imply a permission of another type. For example, *AllPermission* objects imply every other permission. By carefully specifying Implies for every permission-type (this definition will be similar to our definition of *IsSubsetOf*), we believe this can be solved in a straightforward way. Secondly, Java uses *doPriviliged* instead of *Assert* to add permissions to the dynamic permission set. *doPriviliged* adds all permissions granted to the callee to the dynamic permissions set, whereas *Assert* just adds a single, specific permission (or permission set). Since Java's *doPrivileged* is essentially a simplification of .NET's *Assert* (*doPrivileged* can be emulated using the appropriate *Assert* statements), we think this difference will not lead to any problems.

Inheritance

A method level contract states the permissions required by a certain method. As such a contract is essentially a precondition, it can only be weakened at the subclass level (in line with Liskovs substitution principle), meaning that an overriding method cannot require more permissions than its corresponding base class method.

While this may seem problematic at first, our analysis of the BCL has shown that situations where the set of required permissions depends on dynamic binding occur only rarely. Many security sensitive methods cannot be overwritten since they are static or sealed (this includes constructors) or since their class is defined as static or sealed. Examples are the classes *File* and *Dns* or the method *Assembly.Load*.

Static Checking to Increase Performance

Successful verification of a component guarantees that this component will not throw any *SecurityException*s when (1) the minimal permission set described by the component-level contract is a subset of the static permissions that are assigned to it by the security policy, (2) it is only called under a dynamic permission set that satisfies its method-level contracts and (3) all methods that are called by the verified component respect their contracts. If all components within a VM verify, we know that no *Demand* will ever result in a *SecurityException* and that it is safe to turn off run-time checking. In other words, instead of starting a stack inspection, *Demand* immediately returns normally. For applications that rely heavily on Code Access Security, the performance gains can be considerable. Note that for the *Test* operations a stack walk is still required, so in-lining is restricted.

6 RELATED WORK

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

In this paper, we extend our earlier work on contracts for stack inspection [11] by distinguishing between component-level and method-level contracts and by introducing the new security operation *Test*.

Pottier, Skalka and Smith [3] developed a security typing system and showed that in a program that passes their type checker, no *Demand* ever fails at run time. Our preconditions are more expressive, and consequently less conservative, than their typing system. As opposed to [3], our analysis is path-sensitive, meaning that it takes branches within method implementations into account when verifying the correctness of a method. For instance, for

if (C) { **new** $DnsPermission().Demand();$ }

Pottier does not take the condition C into account, and their type checker deduces an

overly conservative type that requires *DnsPermission* to be in the dynamic permission set, regardless of C. Furthermore, while our approach can handle parameterized permissions, [3] considers permissions to be atomic: a piece of code either has the permission , or does not have the permission at all. For some types of permissions, such as *FileIOPermission*, this is too restrictive. For instance, consider the following permission:

new *FileIOPermission*(‘‘`/tmp`’’).*Demand*();

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: [3] can algorithmically infer the type (i.e. the contract) 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*.

In [10], Abadi and Fournet present an extension of the traditional stack inspection system, which not only prevents luring attacks based on method calls, but also deals with other possible interactions, such as reliance on results generated by untrusted code. More specifically, they propose a system where the dynamic permission set depends not only on the rights of code currently on the call stack, but on rights of all code that has ever been executed. We believe our solution can easily be extended in order to support the approach presented by Abadi and Fournet. First of all, the definition of the SPS-transformation needs to be changed slightly since calling a method permanently influences the dynamic permission set at the call-site. We propose adding an extra out-parameter to every method which returns the updated dynamic permission set. Secondly, we also need postconditions that constrain the value returned by this out-parameter, in order for modular, static verification to be possible.

A similar idea for validating security properties of programs by relying on general purpose program verifiers is discussed in [13]. For verifying whether a certain security property holds, they proceed in three steps. First, annotations corresponding to the property are generated, secondly these annotations are propagated throughout the program and finally they input the fully annotated program to a program verifier. Their technique for propagating annotations is path insensitive. Nonetheless, we are currently investigating whether using their propagation algorithm could allow us to trade off some of the completeness of our approach for a reduction in annotation overhead.

In [12], 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 insecure control flow in the library.

Koved, Pistoia and Kershenbaum [2] 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 security-passing style transformation used in this paper was first proposed by Wallach et al. In [8], they show how at least in some cases the performance of stack inspection can be improved by applying this transformation.

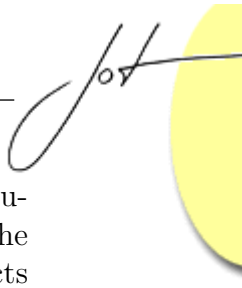
7 CONCLUSION AND FUTURE WORK

We proposed formal contracts for stack inspection-based sandboxing. A component declares minimal and optional static permissions needed by the component implementation, and each method exposed by the component declares the precondition on the dynamic security context that ensures that the method implementation will not throw security exceptions. We worked out our solution in the context of Spec# and the .NET Code Access Security system and showed that the contracts are verifiable using the Spec# Program Verifier.

Future Work

As we have presented it, our solution is rather verbose and requires quite a lot of annotation overhead. We'd like to explore ways to automatically infer some of the sandboxing contracts in order to reduce this overhead. An analysis of the use of Code Access Security in the Rotor BCL has shown that most occurrences of permission demands are instances of the following pattern: a method validates its parameters, creates an appropriate permission object possibly based on method parameters, demands the permission and subsequently asserts sufficient permissions to make sure the rest of the method execution will not throw further *SecurityException*s. For methods that follow this pattern, inferring appropriate method-level contracts is fairly easy. In particular, if the *Demand* is specified declaratively (40% of the *Demand*s in de 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 method-level contracts is future work.

To deal with the fourth disadvantage listed in section 3, we could extend our solution to specify and verify history-based access control instead of standard stack inspection. As described in section 6, we would need to extend the method-level contract to also include a postcondition specifying the effect of a method on the security context. And secondly, next to an extra in parameter, the SPS-transformation



also needs to add to each method an additional out-parameter, returning the security context after completion of the method. It is not clear to us yet whether the additional annotation overhead of writing postconditions in method-level contracts would be workable in practice.

Acknowledgements The authors would like to thank Wolfram Schulte and Erik Poll for their comments and feedback on an early draft of this paper. We would also like to thank the reviewers of .NET Technologies '05 for their useful comments and feedback.

REFERENCES

[1] Drew Dean, Edward W. Felten, Dan S. Wallach and Dirk Balfanz. Java Security: Web Browsers and Beyond. Technical Report 566-97, Department of Computer Science, Princeton University, February 1997

[2] Larry Koved, Marco Pistoia and Aaron Kershenbaum. Access Rights Analysis for Java. In Proceedings of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2002

[3] Francois Pottier, Christian Skalka and Scott Smith. A Systematic Approach to Static Access Control. ACM Transactions on Programming Languages and Systems, volume 27, number 2, 2005

[4] Mike Barnett, K. Rustan M. Leino and Wolfram Schulte. The Spec# Programming System: An Overview. In CASSIS 2004, LNCS vol. 3362, Springer, 2004

[5] David Stutz, Ted Neward and Geoff Shilling. Shared Source CLI Essentials. O'Reilly, 2003

[6] Dan S. Wallach, Andrew W. Appel and Edward W. Felten. SAFKASI: A security mechanism for language-based systems. ACM Transactions on Software Engineering and Methodology, volume 9, number 4, 2000

[7] Brian A. LaMacchia, Sebastian Lange, Matthew Lyons, Rudi Martin, Kevin T. Price. .NET Framework Security. Addison wesley, 2002

[8] Adam Darvas and Peter Müller. Reasoning about Method Calls in JML Specifications. Formal Techniques for Java-like Programs, 2005

[9] Jan Smans, Bart Jacobs and Frank Piessens. www.cs.kuleuven.ac.be/∼jans/casverify, February 2006

[10] Martin Abadi and Cédric Fournet. Access Control based on Execution History. In Proceedings of the 10th Annual Network and Distributed System Symposium (NDSS'03), 2003

[11] Jan Smans, Bart Jacobs and Frank Piessens. Static Verification of Code Access Security Policy Compliance of .NET Applications. In Proceedings of the Third International Conference on .NET Technologies, 2005

[12] Frederic Besson, Tomasz Blanc, Cédric Fournet and Andrew Gordon, From Stack Inspection to Access Control: A Security Analysis for Libraries, 17th IEEE Computer Security Foundations Workshop, 2004

[13] Mariela Pavlova, Gilles Barthe, Lilian Burdy, Marieke Huisman and Jean-Louis Lanet. Enforcing High-level Security Properties for Applets, Smart Card Research and Advanced Applications (CARDIS), 2004

[14] Cédric Fournet and Anrew Gordon. Stack Inspection: theory and variants. Symposium on Principles of Programming Languages, 2002

ABOUT THE AUTHORS

Jan Smans is PhD student at the Katholieke Universiteit Leuven in Belgium and is a Research Assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.-Vlaanderen). His main research interest is in specification and verification of security properties of object-oriented programs. He can be reached at jan.smans@cs.kuleuven.be.

Bart Jacobs is PhD student at the Katholieke Universiteit Leuven in Belgium and is a Research Assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.-Vlaanderen). His main research interest is in specification and verification of concurrent and/or security-critical, object-oriented programs. He can be reached at bart.jacobs@cs.kuleuven.be.

Frank Piessens is a professor at the Department of Computer Science of the Katholieke Universiteit Leuven in Belgium. His research interests are in security aspects of software, including security in operating systems and middleware, security architectures, application security, secure programming languages, Java and .NET security, and software interfaces to security technologies. He can be reached at frank.piessens@cs.kuleuven.be