

# Securing the .NET Programming Model (Industrial Application)

Andrew Kennedy

*Microsoft Research Ltd, 7 J J Thomson Ave, Cambridge, United Kingdom*

---

## Abstract

The security of the .NET programming model is studied from the standpoint of fully abstract compilation of  $C^\sharp$ . A number of failures of full abstraction are identified, and fixes described. The most serious problems have recently been fixed for version 2.0 of the .NET Common Language Runtime.

*Key words:* Security, contextual equivalence, full abstraction,  $C^\sharp$ , .NET

---

## 1 Introduction

Type safety is a crucial aspect of security on Microsoft's .NET platform just as it is on the Java Virtual Machine (JVM). Enforcement of type system rules by the .NET Common Language Runtime (CLR) and JVM rules out a number of dynamic errors, such as using an integer in place of an object reference, or invoking a method with the wrong number of arguments. The .NET platform also uses types to securely isolate *application domains* ('software processes'), ensuring that information cannot leak accidentally between domains that are implemented by a single shared heap.

Typically, a type loophole (in design or implementation) opens up the possibility of a malicious program executing arbitrary code. But the security of a program can rely on more subtle properties, some of which can arguably be considered type safety aspects (access control for fields and methods, sealing of classes, etc) but others which go beyond it.

Abadi noted the relevance of *full abstraction* to the security of a programming model [1]. A translation is fully abstract if it preserves and reflects obser-

---

*Email address:* [akenn@microsoft.com](mailto:akenn@microsoft.com) (Andrew Kennedy).

vational equivalence. So if source-language compilation is not fully abstract, then there exist contexts (think ‘attackers’) in the target language that can observably distinguish two program fragments not distinguishable by source contexts. Such abstraction holes can sometimes be turned into security holes: if the author of a library has reasoned about the behaviour of his code by considering only source-level contexts (*i.e.* other components written in the same source language), then it may be possible to construct a component in the target language which provokes unexpected and damaging behaviour.

Abadi identified one such failure of full abstraction in the translation from Java to JVM bytecodes. The “Industrial Application” studied in this short paper is the investigation of full abstraction properties of the compilation from C<sup>#</sup> [4,5] to the Intermediate Language (IL) executed by the .NET CLR [6,7].

## 2 The bad news

We first present some bad news: six C<sup>#</sup> coding patterns, made insecure by the failure of full abstraction. For each pattern we present an equivalence that holds in C<sup>#</sup> but whose compilation to IL breaks the equivalence, for the 2002 specification of IL [6]. All flaws were identified by the author; some were independently discovered by product teams at Microsoft.

### *Problem 1: Overridden methods are exposed*

In C<sup>#</sup> and other object-oriented programming languages, method overriding can be used to wrap methods from existing classes with additional functionality, safe in the knowledge that client code cannot access the overridden method. A typical use is to add parameter validation:

```
class InsecureWidget {
    // No checking of argument
    public virtual bool Put(string s) {...}
}
class SecureWidget : InsecureWidget {
    // Check argument before delegating to superclass
    public override bool Put(string s)
    { return Valid(s) ? base.Put(s) : false; }
}
```

From IL, but not from C<sup>#</sup>, overridden methods can be called directly, subverting the abstraction:

```

.locals (class SecureWidget sw)
ldloc sw    ldstr "Invalid string"
call void InsecureWidget::Put(string) // direct call

```

To demonstrate that this feature does indeed break full abstraction, we must exhibit two observationally equivalent source language expressions and a context that distinguishes their translations. We could flesh out this example by making the validity check observable. Instead, consider the following, simpler example of method override:

```

public class Counter {
    protected uint count = 0;
    public void Inc() { count++; }
    public virtual uint Get() { return count; }
}
public class ParityCounter : Counter {
    public override uint Get() { return count%2; }
}

```

Here we have defined a simple counter class whose internal state is hidden from clients, but accessible from the subclass `ParityCounter`, which refines the class to expose only the parity of the counter. The following equivalence holds in  $C^\sharp$ :

```

void Bump(ParityCounter p) { p.Inc(); p.Inc(); }
≈ void Bump(ParityCounter p) { }

```

At the level of IL, these methods can be distinguished by making a direct call on `p` to the overridden method `Counter.Get`, thus demonstrating that the translation from  $C^\sharp$  to IL is not fully abstract.

### *Problem 2: Boxing breaks encapsulation*

Encapsulation is a fundamental concept in object-oriented programming. It “ensures that users of an object cannot change the internal state of the object in unexpected ways” (Wikipedia). A number of standard .NET types encapsulate their state *immutably*; `String` is an example that is heap-allocated, and `DateTime` and `Int32` are so-called *value types* that also prohibit mutation. Values with such types can be *boxed* to obtain a heap-allocated version; as far as  $C^\sharp$  is concerned, the boxed value behaves in the same way as the unboxed one, so properties such as immutability are preserved. Here is some code that makes use of boxing:

```

// A dictionary keyed on strings
class StringDict {
    private Hashtable dict;
    public object Get(string s) { return dict[s]; }
    internal void Set(string s, object o) { ... }
}
class HR {
    public static StringDict salaries;
    ...salaries.Put("akenn", (object) ...)
}

```

```

// In another component, far away...
int mysalary = (int) HR.salaries.Get("akenn");

```

The `Set` method is `internal` and so cannot be invoked by client components; so it would seem that `salaries` can be updated only from within this component. Unfortunately, the `unbox` instruction in IL, used to compile the `(int)` cast above, produces an ‘interior’ pointer to an object that can be used to replace the boxed value in place. And so boxed data, such as the salary record above, can be mutated unexpectedly:

```

ldsfld StringDict HR::salaries    ldstr "akenn"
call object StringDict::Get(string)
unbox System.Int32                // Obtain interior pointer
ldc.i4 10000000                   // That'll do nicely
stind.i4                           // Replace my salary

```

Failure of full abstraction can be demonstrated by the following equivalence which is broken by compilation to IL:

```

public int g(object o) { int i = (int) o; f(o); return i; }
≈ public int g(object o) { int i = (int) o; f(o); return (int) o; }

```

### *Problem 3: Exceptions are not **Exceptions***

In the C<sup>#</sup> language, only objects whose type derives from `System.Exception` can be thrown as exceptions. Programmers sometimes use this assumption in a transaction-like coding pattern, where recovery code is placed in a catch-all block:

```

try {
    // perform some action, to completion
} catch (Exception e) {
    // undo action if an exception was thrown
}
// Action either completed, or was fully undone

```

Unfortunately, the catch-all doesn't catch all, as IL permits objects of any class to be thrown:

```

newobj instance void System.Object::.ctor()
throw

```

Suppose that the action in the try-block involved a call to code over which an attacker had control; then this code could raise an exception such as above, and possibly leave the program in a bad state.

To see that full abstraction is broken, consider the following C<sup>#</sup> statements, which are indistinguishable from C<sup>#</sup> but can be distinguished at the level of IL by defining an appropriate object-throwing method *f*:

```

int i=0; try { f(); i++; } catch(Exception e) { i++; }
≈ int i=1; try { f(); } catch(Exception e) { }

```

*Problem 4: bools are not two-valued*

Values `false` and `true` have type `bool` in C<sup>#</sup> – and no other value inhabits the type. So it is reasonable for a programmer to suppose that the `WriteLine` statement in this method can never be executed:

```

public void NotNot(bool b) {
    bool c = !b;
    if (!c != b) { Console.WriteLine("This cannot happen"); }
}

```

Unfortunately, in the type system of the runtime, `bool` and `byte` are interchangeable, with no conversions applied. So `Foo` can be tricked by passing it a value not equal to 0 or 1 (the representations for `false` and `true`):

```

ldc.i4 2    call void NotNot(bool)

```

Thus the following equivalence is not preserved by compilation:

```
public bool Eq(bool x, bool y) { return x==y; }
≈ public bool Eq(bool x, bool y) { return (x==true)==(y==true); }
```

*Problem 5: this can be null*

In C# the keyword `this` refers to the object on which a method was invoked. If a method is invoked on a null value then `NullReferenceException` is thrown. So a programmer would reasonably expect that `this` can never be null. Consider the method `Act` below, which executes some privileged action only if the object has previously been registered through a public method.

```
class Widget {
  // Instance registered for privileged action, private to class
  private static Widget registered = null;

  // Only register object if it passes authentication
  public void Register() {
    if (Authenticate()) { registered = this; }
  }

  public void Act() {
    if (this == registered) {
      // Perform privileged action
    }
  }
}
```

Unfortunately, it is possible from IL to trick the method `Act` into executing the privileged action prior to registration by passing it a null instance:

```
ldnull
call void Widget::Act()
```

It is clear that the following contextual equivalence, valid in C#, is not preserved by compilation to IL:

```
public bool m() { return this==null; }
≈ public bool m() { return false; }
```

*Problem 6: out parameters are also in*

In contrast to Java, C# supports call-by-reference as well as call-by-value. A special form of call-by-reference, so-called *out* parameters, *must* be assigned by the callee before returning. Moreover, according to the C# Specification [5, §17.5.1.3], “an output parameter is initially considered unassigned and must be definitely assigned before its value is used”. Typically, *out* parameters are used to return multiple values from a method, as in the following example:

```
// Reader interface, int denotes number of repeats of a string
interface IReader { int Get(out string result); }
class C {
    public static void TestGet(IReader reader)
    {
        string s = GetSecret();
        ...
        int n = reader.Get(out s);
        ...
    }
}
```

Here, a programmer has used variable *s* for two distinct purposes in method *TestGet*, first assigning it some privileged information, and subsequently using it to return data from a call to *Get*. This is sloppy programming – but given the C# specification, it should not be insecure.

Unfortunately the typing rules for IL do not distinguish *out* parameters from ordinary call-by-reference. An attacker can pass an instance of *IReader* to *TestGet* that breaks the *out*-parameter invariant and so leak the privileged information:

```
.class public Attacker implements IReader {
    .field public static string Leak
    .method public virtual int Get([out] string& s) cil managed {
        ldarg s
        ldind.ref
        // We've got hold of s, let's keep it somewhere
        stsfld string Attacker::Leak
        // In fact, we don't even need to assign to s
        ldc.i4 1
        ret
    }
}
```

The following simple equivalence is not preserved by compilation:

```
{ int i = 5; x.m(out i); }  
≈ { int i = 7; x.m(out i); }
```

### 3 The good news

Fortunately, it is possible to fix failures of full abstraction such as these, in one of three ways.

- We can change the *translation*, i.e. the compilation scheme. But it's often the case that the non-fully-abstract translation was direct and efficient; anything less direct is likely to cost more. For example, Problem 4 can be fixed by generating code to 'normalize' integers typed as `bool` at appropriate points in the code (see below).
- We can increase the expressivity of the source language ( $C^\sharp$ ) to add back contexts corresponding to those in IL, thereby identifying fewer terms. For example, we could solve Problem 3 by supporting the throwing of arbitrary objects; or we could solve Problem 1 by supporting direct calls in  $C^\sharp$  on overridden methods. Although technically this is a solution to the 'full abstraction problem', it weakens our ability to create abstractions in the source language.
- We can reduce the expressivity of the target language (IL) to rule out problematic contexts, and therefore identify more terms. This is the ideal fix; but it is a 'breaking' change to the specification of the target. This is particularly awkward when multiple source languages are supported, as is the case in the .NET CLR.

Problems 1, 2 and 3 identified above have been fixed in version 2.0 of the .NET CLR by changing the specification of IL [7], and implementing these changes in the verifier (type-checker) and runtime system.

**Problem 1.** At verification time we simply reject direct calls to non-final virtual methods, except when invoking an overridden method from the overriding method (a so-called *base* call).

**Problem 2:** The mutable boxed value type problem is solved by changing the typing rule for the `unbox` instruction to return a special kind of interior pointer – called a “controlled mutability” pointer in the IL specification [7] – that prohibits update-in-place.

**Problem 3:** We wrap all non-`Exception`-deriving exceptions inside objects of type `RuntimeWrapperException`, and unwrap the exception when caught by a language (such as Managed C++) that supports arbitrary exception objects.

That leaves three (known) problems.

**Problem 4.** It turns out that most logical operations on `bools` interpret zero as false and non-zero as true, and hence are not affected by the possibility of values other than 0 or 1. The exceptions are `==` and `!=`, which perform a simple bitwise comparison on bytes. So it would appear at first that we can simply fix the C<sup>#</sup> *compiler* by generating code for equality on booleans that interprets any non-zero value as `true`. Unfortunately, that’s not enough, as it may be the context which performs the equality test. The following equivalence is not preserved:

```
bool b = f(); g(b);
≈ bool b = f(); g(b ? true : false);
```

Consider a distinguishing IL context in which the function `f` returns the value 2 and the function `g` tests for ‘boolean’ equality between its argument and the value 2. A better fix is therefore to change the behaviour of equality on booleans at the IL level.

**Problem 5.** The `null` instance problem is mitigated somewhat by the fix for Problem 1, as virtual calls must check for null already, and so with this in place only non-virtual instance methods can now invoke on null instances. One solution is to alter the translation of C<sup>#</sup> to IL, by inserting a null-check in a prologue to every non-virtual instance method, throwing `NullReferenceException` if the check fails. Alternatively, the IL specification could be changed so that the semantics of the `call` instruction on instance methods is to throw an exception when passed a null instance.

**Problem 6.** The ideal fix for this problem is simply to replicate the C<sup>#</sup> rules at the level of IL, verifying that `out` parameters are not accessed until they have been assigned a value.

It is the opinion of the author that these last three problems are not as serious – indeed it is hard to contrive even artificial security exploits. Nevertheless, in the absence of fixes, we should educate programmers, warning them that `bool` can hide a byte, that `this` may be null, and that variables passed as `out` parameters can be inspected by the callee.

## 4 Discussion

One could argue that full abstraction is just a nicety; programmers don’t *really* reason about observations, program contexts, and all that, do they? Well, actually, I would like to argue that they *do*. At least, expert programmers, the

sort that produce design patterns, coding guidelines, and the like, do think this way – how else would patterns of the kind discussed in Section 2 be proposed?

Nonetheless, (aiming for) full abstraction is just a start. Languages inevitably contain weaknesses, C<sup>#</sup> included, and these weaknesses lead to security holes. For example, the mutability of arrays is a common cause of security bugs in libraries for both Java and C<sup>#</sup>. (Typically, a programmer marks an array-type field or property `readonly` but forgets that the elements of the array can be mutated.) The ability to apply checked downcasts can lead to holes too; one naïve ‘solution’ to the mutability of arrays is to pass the array at a supertype that prevents mutation (`System.IEnumerable`); this fails because the array type can be recovered through downcasting. As the semantics community knows, the right way to think about such issues is by studying observational equivalence.

The complexities of industrial programming languages and platforms can make questions such as full abstraction and even type safety hard to pin down. For example, .NET, like Java, has a *reflection* capability that destroys any sensible notion of contextual equivalence: programs can reflect on the number of methods in their implementation, or inspect the current call-stack. To be of any use at all, a definition of contextual equivalence has to ignore these capabilities (which, incidentally, are not available to untrusted components, *i.e.* the sort that we have been considering as potential ‘attacker’ contexts). Furthermore, the .NET platform does not require verification of trusted components such as system libraries – but such components must still preserve the invariants for which verification is a sound, conservative decision procedure.

What next? Well, of course we have no *proofs* of full abstraction, nor even proofs of contextual equivalences that hold in C<sup>#</sup>. Neither do we have proofs of type safety for all of C<sup>#</sup> or IL, though recently large subsets of both C<sup>#</sup> and IL have been formalized [2,3] and proved type-safe, and we understand type safety well enough to be confident that these results scale to the full language.

Given this, a more pessimistic approach is to assume that full abstraction holes abound; then, make absolutely certain that particular coding patterns are secure. Or, we might consider full abstraction at restricted *types* – surely we can be confident of the security of methods of type `int → int`?

In summary, the aim of this work is to ensure that C<sup>#</sup> programmers can reason about the security of their code by *thinking in C<sup>#</sup>*. More precisely:

A C<sup>#</sup> programmer can reason about the security properties of component A by considering the behaviour of another component B written in C<sup>#</sup> that “attacks” A through its public API.

This can only be achieved if compilation is fully abstract.

## Acknowledgements

This work has benefited from discussions with Martin Abadi, Nick Benton, Cédric Fournet and Claudio Russo. I would also like to thank members of the Microsoft product teams, in particular Chris Brumme, Vance Morrison and Paul Vick, for taking a serious interest in the issues described here, and for pushing through appropriate changes to the .NET platform.

## References

- [1] M. Abadi. Protection in programming-language translations. In *ICALP '98: Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, pages 868–883, London, UK, 1998. Springer-Verlag.
- [2] E. Börger, N. G. Fruja, V. Gervasi, and R. F. Stärk. A High-Level Modular Definition of the Semantics of C#. *Theoretical Comput. Sci.*, 336(2–3):235–284, June 2005.
- [3] N. G. Fruja. A Modular Design for the .NET CLR Architecture. In A. S. D. Beauquier and E. Börger, editors, *12th International Workshop on Abstract State Machines, ASM 2005, Paris, France*, pages 175–199, March 2005.
- [4] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language*. Addison-Wesley, 2004.
- [5] Standard ECMA-334: C# Language Specification, 3rd edition, June 2005. See [www.ecma-international.org/publications/standards/Ecma-334.htm](http://www.ecma-international.org/publications/standards/Ecma-334.htm).
- [6] Standard ECMA-335: Common Language Infrastructure (CLI), 2nd edition, Dec. 2002. See [www.ecma-international.org/publications/standards/Ecma-335.htm](http://www.ecma-international.org/publications/standards/Ecma-335.htm).
- [7] Standard ECMA-335: Common Language Infrastructure (CLI), 3rd edition, June 2005. See [www.ecma-international.org/publications/standards/Ecma-335.htm](http://www.ecma-international.org/publications/standards/Ecma-335.htm).