Securing the .NET Programming Model  
(Industrial Application)

Andrew Kennedy  
Microsoft Research Ltd  
7 J J Thomson Ave, Cambridge, United Kingdom  
akenn@microsoft.com

1. INTRODUCTION

The .NET programming model is fundamental to security on Microsoft’s .NET platform, just as the Java programming model forms a part of the security model on the JVM. Type safety is a crucial aspect of both systems. Enforcement of type system rules by the .NET Common Language Runtime (CLR) and the Java Virtual Machine (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 despite sharing a 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 observational 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 there may exist components written in the target language which provoke 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♯ to the Intermediate Language (IL) executed by the .NET CLR.

2. THE BAD NEWS

We first present some bad news: three C♯ coding patterns, made insecure by the failure of full abstraction.1

Problem 1: Overridden methods are exposed

In C♯ 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:

```csharptext
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♯, overridden methods can be called directly, subverting the abstraction.

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

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♯ 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:

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

The Set method is internal and so cannot be invoked by clients; 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:

```csharptext
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
```
Problem 3: Exceptions are not Exceptions

In the C♯ language, only objects whose type derives from System.Exception can be thrown. Programmers sometimes use this assumption in a transaction-like coding pattern, where ‘back-off’ 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.

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 therefore efficient; anything less direct is likely to cost more.

- We can increase the expressivity of the source language (C♯) 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♯ 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.

Each of the problems identified earlier have now been fixed by changing the specification of IL.

Problem 1: We simply rule out direct calls to non-final virtual methods, except when invoking an overridden 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 that prohibits update-in-place [3].

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

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 abstractions of the kind discussed in Section 2 be proposed?

Nonetheless, (aiming for) full abstraction is just a start. Languages inevitably contain weaknesses, C♯ 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♯. 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. And as the semantics community knows, the right way to think about such issues is by studying observational equivalence.

What next? Well, of course we have no proofs of full abstraction. (We do not even have complete abstract formalizations of C♯ or IL). Neither do we have proofs of type safety – but at least there are studies of subsets of IL [2], 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; instead, 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[2]

In summary, the aim of this work is to ensure that C♯ programmers can reason about the security of their code by thinking in C♯. More precisely:

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

This can only be achieved if compilation is fully abstract.

5. 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.

6. REFERENCES


Exercise for the reader: using the IL specification, show how use of bools is not fully abstract!