eXtensible Multi Security Contracts for .NET Platform

Wiktor Zychla
wzychla@ii.uni.wroc.pl

Institute of Computer Science
University of Wroclaw, Poland

.NET Technologies 2006

Outline of Part I

1. Static vs Dynamic Security
2. Design by Contract
3. What is eXtensible Multi Security

Outline of Part II

4 Introduction to Proof-Carrying-Code

5 Central Theorem of PCC

6 PCC Certification Protocol

Outline of Part III

7 PCC for XMS
8 Symbolic Evaluation
9 How it works
10 First example
11 Other Aspects of OO Languages
12 Example

Outline of Part IV

13 High-Level Paradigms

14 Compilation issues

15 Integration Strategies

Outline of Part V

16 Dynamic XMS Contracts

Outline of Part VI

17 Applications of XMS

Outline of Part VII

18 Validation of XMS Certificates

19 Implementation Details

Outline of Part VIII

20 Future of XMS

21 Availability of XMS

Part I

Overview of XMS

Security Policy

The Security Policy is a formal set of rules and restrictions that somehow tells us which programs are valid and which are invalid and should be considered illegal, unsafe.

- memory safety
- type safety
- control flow safety
- information flow safety
- code correctness

Language-Based Security

The Security Policy must be formal and objective.
Language-Based Security Policies exploit the semantics of programming languages, operating systems and/or runtime environments.

Enforcing a Security Policy

How do we enforce a security policy?

- Dynamic security
  - Policy is constantly checked at run time
  - Needs to be supported by a runtime environment
- Static security
  - Validation result does not require the code to be actually run
  - Valiadtion may reject valid code
  - Does not to be supported by a runtime environment

Design by Contract

Communication between entities is based on obligations which take the form of *predicates*.
Specification of a method is a quadruple:

$$Spec_F = (Sig_F, Pre_F, Post_F, Inv_F)$$

where Sig_F is a method's signature, Pre_F is a precondition predicate, $Post_F$ is a postcondition predicate, Inv_F is a partial function that maps instruction numbers to invariants.

DBC Security Policy

The Design By Contract Security Policy states that a method F is safe when

- the precondition Pre_F holds upon the invocation
- the postcondition $Post_F$ holds when F returns
- a invariant $Inv_F(i)$ holds when i-th instruction is executed

eXtensible Multi Security Framework

eXtensible Multi Security Framework is a security framework for Microsoft Intermediate Language. It currently supports static and dynamic Contract Security Policy. Its primary focus is static verification.

- Static verification engine
  - works directly on MSIL
  - based on Proof-Carrying-Code paradigm
- Dynamic verification engine
  - much easier than the other
  - instrumentates code by using Context-Bound objects

Evolution of XMS

- DBC/PCC implementation for a toy C-like language
- concurrent work on other formal security policies
- currently beeing ported to the enterprise world [.NET]

Benefits of XMS

- XMS is designed to certify the MSIL language, one of the most widely used enterprise intermediate languages.
- To support XMS the .NET Runtime Environment does not need to be changed in any way.
- XMS certificates are compatible with existing high-level .NET languages. A high-level language developer does not need to know MSIL to certify the code.
- XMS certificates are built around the notion of PCC thus inherit all desirable properties of PCC:
  - the certificates are sufficient to guarantee that the code is valid,
  - the authority of a code producer is completely insignificant to the code security.

Part II

Proof-Carrying-Code Paradigm

Proof-Carrying-Code

Proof-Carrying-Code (PCC) paradigm has been proposed by George Ciprian Necula in 1998. It is a generalisation of many eariler Language-Based Security techniques.
Three key ideas

- Verification Condition (VC), a logic predicate that contains the information about the program execution.
- Verification Condition Generator (VCGen), a utility which rebuilds VCs from modules of given language
- Proof Checker, a utility which is able to verify the correspondence between a logic predicate and its formal proof

Central PCC Theorem

The Central PCC Theorem states that:

For given Safety Policy S and code F, if the Verification Condition for S applied to F is valid, i.e.

$$S \models VC_S(F)$$

then the code F is safe according to S.

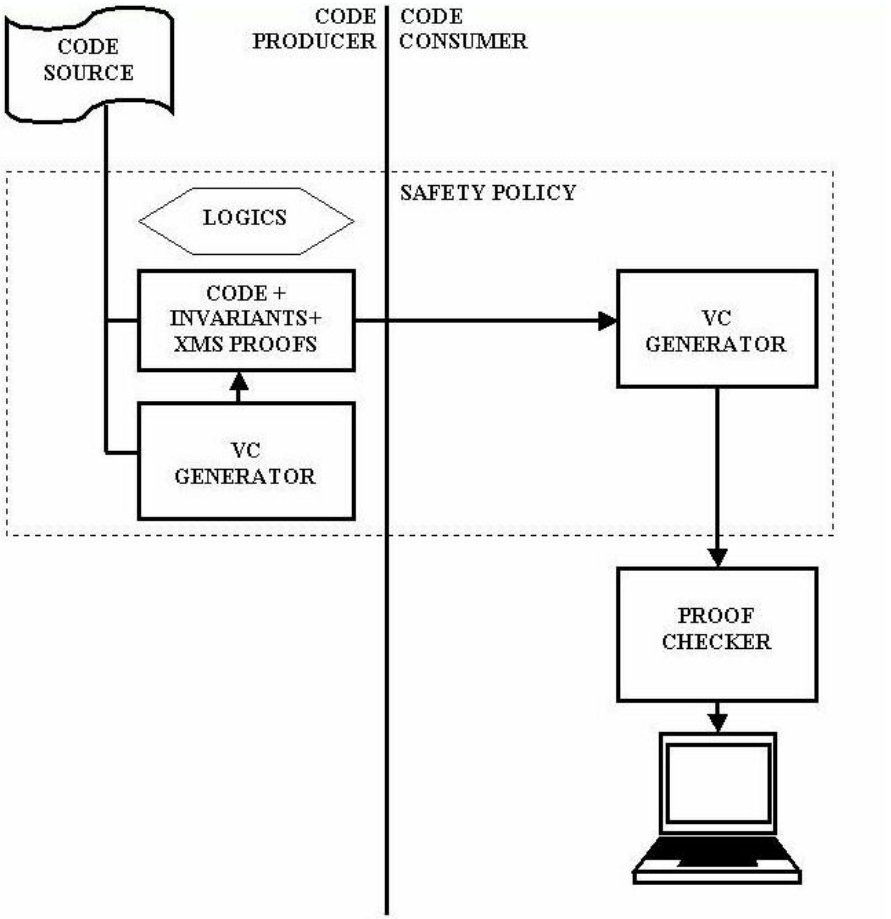
Central PCC Theorem - challenges

Such generality raises severe challenges:

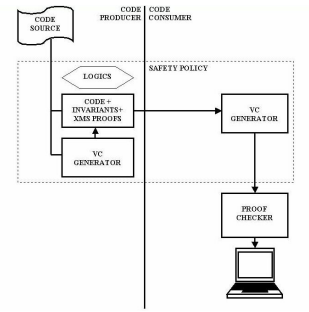
- safety policy S must be expressed with a formal logic
- sound and complete proof system must exist for S
- VCGen must be built for the language
- the Security Theorem must be proved

Original PCC was defined for Type-Safety of simple generic RISC-like assembly language.

PCC Certification Protocol



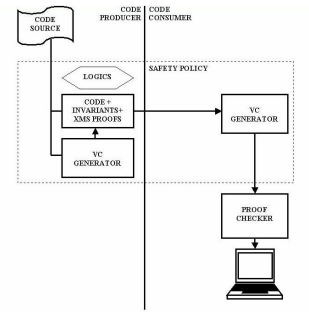
PCC Certification Protocol



The Code Producer:

1. adds method specifications to the source code,
2. uses VCGen to build and encode Verification Conditions (VC),
3. constructs proofs for VCs,
4. embeds VCs and proofs as a metadata (metadata is not used at runtime but is extracted in the certification process).

PCC Certification Protocol



The Code Consumer:

1. uses VCGen to build Verification Conditions,
2. checks if the same VCs have been supplied with the code by the Code Producer,
3. validates the correctness of proofs (certificates).

PCC Certification Protocol

The protocol may fail at some point at the Code Consumer side. Specifically:

1. the binary may not contain the metadata that is required to build Verification Conditions,
2. the predicates built at Code Consumer side can differ from these supplied with the code,
3. proofs supplied with the code can be invalid in the sense that they do not prove Verification Conditions.

If the protocol **fails** for any of these reasons the Code Consumer should **reject** the code as unsafe.

Part III

Static XMS Contracts for MSIL

Formal Semantic of MSIL

The semantics of the IL language is well documented in the CLI Draft. However, it rather takes a semi-formal form.

Because a precise semantics is a core of XMS infrastructure we had to reformulate it in a concise, formal manner.

Formal Semantic of MSIL

We model the execution state as a tuple $\Sigma = (i, \rho)$ that contains a program counter $i \in Dom(F)$ and a local memory context ρ. In a fixed context, we will sometimes write $(i, (l_A, l_V, h, H, s))$ instead of (i, ρ)

The operational semantics is a formal judgement of a form $F \vdash (i, \rho) \mapsto (j, \rho')$. It means that the execution of F takes one step from state (i, ρ) to state (j, ρ').

We assume that $0 \in Dom(F)$ and that the execution of F starts in a state $\Sigma_0 = (0, l_A, l_V, h, H, \epsilon))$.

Formal Semantic of MSIL

Example judgements:

$$\frac{F_i = \texttt{add}}{F \vdash (i,\ldots,u\cdot v\cdot s) \mapsto (i+1,\ldots,u+v\cdot s)} \qquad \texttt{add}$$

$$\frac{F_i = \texttt{ldarg}\ v}{F \vdash (i,\ldots,s) \mapsto (i+1,\ldots,l_A(v)\cdot s)} \qquad \texttt{ldarg}$$

$$\frac{F_i = \texttt{call instance}\ \ C\ \ \ T::G}{\begin{array}{c} F \vdash (i, \ldots, u_n \cdot \ldots \cdot u_0 \cdot p) \mapsto \\ G \vdash (0, l_A[a_{\texttt{this}} \mapsto p, a_0 \mapsto u_0, \ldots, a_n \mapsto u_n], \ldots, \epsilon) \end{array}} \qquad \texttt{call instance}$$

$\vdots$ $\qquad\qquad$ $\vdots$

PCC Theorem for XMS

The General PCC Theorem for XMS Contracts is stated as follows:

A method F is safe with respect to Static Contracts if for any initial state $\Sigma_0 = (0, \rho_0)$ such that $\rho_0(Pre_F)$ and any state $\Sigma = (i, \rho)$ reachable from the initial state we have that if $F_i = \texttt{ret}$ then $\rho(Post_F)$.

$$Safe_{SC}(F) \iff$$

$$\forall_{\Sigma_0=(0,\rho_0),\, \Sigma=(i,\rho)}\ \rho_0(Pre_F) \wedge \Sigma_0 \mapsto^* \Sigma \wedge F_i = \texttt{ret} \Rightarrow$$

$$\rho(Post_F)$$

XMS VCGen

Verification Conditions are generated by Symbolic Evaluation of code. The evaluation is defined as a recursive function that takes four parameters, written as

$$SE_F(i, \sigma, \mathcal{L}, b)$$

where

- F is a method whose body is evaluated
- i is an address of evaluator's current instruction
- σ is a *symbolic store*
- $\mathcal{L}$ is a loop stack

XMS VCGen

The symbolic evaluator is run against all methods in a module $\mathcal{M}$ and the global verification condition is build using the resulting predicates. In a simplified form:

$$VC(\mathcal{M}) = \bigwedge_{F \in \mathcal{M}} VC(F)$$

$$VC(F) = \forall a_0, \ldots, a_n . \sigma_0^F(Pre_F) \Rightarrow SE(0, \sigma_0^F, \emptyset, \texttt{true})$$

where:

$$\sigma_0^F = (I_A[a_i \mapsto a_i], I_V[v_i \mapsto 0], \epsilon)$$

XMS VCGen

During the scan, the Symbolic Evaluator simulates the method's execution by updating the symbolic store with respect to current instruction.

From the Evaluator's perspective there are two types of MSIL instructions. For some instructions SE does not produce anything, it just changes the state of the symbolic store. For other instructions SE not only changes the state of the symbolic store but also produces a part of the Verification Condition.

XMS VCGen Opcodes

The evaluation is sometimes easy and obvious.

- In case of `ldc` instruction SE puts the integer parameter at the top of the symbolic stack.
- In case of all arithmetic instructions (`add`, `sub`, `mul`, ...) SE performs the symbolic evaluation and puts the result back to the symbolic stack.
- The `ldarg`, `ldloc` and `ldsfld` instructions put the value from local argument, local store or the shared store (respectively) at the top of the stack.
- ...

XMS VCGen - Opcodes

$$F_i = \texttt{ldc i4}\ u \wedge \sigma = (i, \ldots, s) \Rightarrow SE_F(i) = SE_F(i{+}1, \ldots, u{\cdot}s, \mathcal{L})$$

$$F_i = \texttt{add} \wedge \sigma = (i, \ldots, u{\cdot}v{\cdot}s) \Rightarrow SE_F(i) = SE_F(i{+}1, \ldots, u{+}v{\cdot}s, \mathcal{L})$$

$$F_i = \texttt{ldarg}\ v \wedge \sigma = (i, \ldots, s) \Rightarrow SE_F(i) = SE(i{+}1, \ldots, l_A(v){\cdot}s, \mathcal{L})$$

However, since the MSIL is an Object-Oriented language there are also difficult cases.

XMS VCGen - Branches

A branch is a first interesting type of opcode. Encountering a brach, Symbolic Evaluators splits in two independend evaluators, one for each branch:

$$F_i = \texttt{bge l} \ \wedge \ l < i \ \wedge \ Inv_F(i) \neq \epsilon \Rightarrow \texttt{fail}$$

$$F_i = \texttt{bge l} \ \wedge \ \sigma = (i, \ldots, u \cdot v \cdot s) \Rightarrow$$

$$SE(i) = \sigma(u) < \sigma(v) \Rightarrow SE(i+1, \ldots, s, \mathcal{L}) \wedge$$

$$\sigma(u) \geq \sigma(v) \Rightarrow SE(l, \ldots, s, \mathcal{L})$$

XMS VCGen - Returning

Another interesting opcode is `ret`:

$$F_i = \texttt{ret} \ \wedge \ Sig_F = C \ F(\ldots) \ \wedge \ \sigma = (i, \ldots, u \cdot s) \Rightarrow$$
$$SE(i) = \sigma(Post_F[u/VALUE])$$
$$F_i = \texttt{ret} \ \wedge \ Sig_F = \texttt{void} \ F(\ldots) \ \wedge \ \sigma = (i, \ldots, s) \Rightarrow$$
$$SE(i) = \sigma(Post_F)$$

An easy example

Consider following C# code:

```
public int Abs( int x )
{
  if ( x >=0 )
    return x;
  else
    return -x;
}
```

The specification would be:

$$Pre_F = \texttt{true} \qquad Post_F = VALUE \geq 0$$

An easy example

It translates to:

```
Int32 Abs (Int32 x)
// Code Size: 15 Bytes
.maxstack 2
.locals (System.Int32 V_0)
L_0000: ldarg.1
L_0001: ldc.i4.0
L_0002: blt.s L_0008
L_0004: ldarg.1
L_0005: stloc.0
L_0006: br.s L_000d
L_0008: ldarg.1
L_0009: neg
L_000a: stloc.0
L_000b: br.s L_000d
L_000d: ldloc.0
L_000e: ret
```

An easy example

The resulting Verification Condition is:

$$\texttt{forall x. true} \Rightarrow$$

$$((\texttt{x} >= 0 \Rightarrow \texttt{x} >= 0) \,\&\, (\texttt{x} < 0 \Rightarrow -\texttt{x} >= 0))$$

This predicate holds and because of the PCC theorem for XMS we conclude that the Contract Security Policy holds for any execution of the method.

To construct a certificate for the method we would only need a formal proof of above predicate.

Other Aspects of OO Languages

- Backward branches
- Method calls
- Objects and arrays
- Polymorphism
- 0-values
- Exceptions
- Delegates, Events, Generics (under research)

Backward branches

Since the recursion of SE must not be infinite we must guard all backward jumps with invariants.

When an invariant is seen for the first time, all variables and stack slots which are modified by the loop body are set to fresh, symbolic values and the invariant is appended to the Verification Condition.

When an invariant is seen for the second time, it is appended to the Verification Condition in the new state of Evaluator.

Method calls

A method call makes VCGen to put its precondition as an assumption into the predicate and then initialize a new state with all variables which could be modified inside the called method (out parameters) set to new, fresh values.

If the method returns a value, a new fresh value is put onto the symbolic stack and the substituted postcondition is guarded by the fresh value universally quantified.

Objects and arrays

Objects are evaluated symbolically, fields are stored in a dictionary. Arrays are stored as index-value dictionaries. Each operation on an

array results in branches that put unification expressions as assumptions.

Polymorphism

Is it not known until the run-time which exact method is called from a class hierarchy. VCGen relies here on subcontracting paradigm according to which contracts of inherited methods must depend on contracts of base-class methods.

0-values

Contracts must allow to use original values in postconditions. VCGen uses special form of an assumption for the Verification Condition of a method to support such possibility.

Example

Consider another C# code (easier to read than MSIL):

```
public static int ComputeGDC( int x, int y ) {
  int k = x;
  int l = y;
  while ( k-l != 0 ) {
    if ( k > l )
      k -= l;
    else
      l -= k;
  }
  return k;
}
```

We have also:

$Pre(F) = x \geq 0 \wedge y \geq 0$
$Post(F) = VALUE = GCD(x, y)$
$Inv(.) = GCD(x, y) = GCD(V_0, V_1)$

Example

Correspondning Verification Condition is:

```
forall x. forall y. (x >= 0 & y >= 0 =>
(((x-y) =0=> x = GCD(x,y)) &
 ((x-y)!=0=>GCD(x,y)=GCD(x,y) &
    forall V_0_. forall V_1_.
      GCD(x,y)=GCD(V_0_,V_1_)=>
        ((V_0_>V_1_ =>
         ((((V_0_-V_1_)-V_1_) =0=>
            (V_0_-V_1_) = GCD(x,y)) &
          (((V_0_-V_1_)-V_1_)!=0=>
             GCD(x,y)=
             GCD((V_0_-V_1_),V_1_)))) &
         (V_0_<=V_1_ =>
          (((V_0_-(V_1_-V_0_)) =0=>
          V_0_ = GCD(x,y)) &
           ((V_0_-(V_1_-V_0_))!=0=>
           GCD(x,y)=
           GCD(V_0_,(V_1_-V_0_)))))))))
```

Part IV

Towards High-Level Langauges

High-Level Languages

- A high-level language developer should not be forced to learn MSIL language. In particular, a solution where a high-level code is first compiled to MSIL and then manually certified is unacceptable.
- A high-level compiler should not require any major changes to support the certification. It would be perfect, if the high-level compiler did not require any changes.

High-Level Language Compilation Issues

- Static Contracts Invariants have the form $Inv_F(i) = (P, ...)$ where i is the MSIL instruction number and P is the invariant predicate. It could be however extremely difficult to determine the MSIL instruction number for given high-level instruction, since it would require a deep knowledge of compiler transformation routines.
- During the compilation to MSIL, names of local variables are omitted.

These issues can be relatively easy solved for high-level languages with simple translation schemes (C#, VB.NET).

High-Level Language Compilation Issues (C#, VB.NET)

- First issue is adressed with the additional scan of the binary code where we discover instructions $I = (i_0, \ldots, i_k)$ that are targets for backward jumps.
  We could then take:

$$Inv_F(i) = \begin{cases} P_j & \text{if } i = i_j \text{ for some } j \text{ and } j \leq n \\ \epsilon & \text{in other case} \end{cases}$$

- The second difficulty is addressed by "virtually" renaming consecutive local variables to $v_0, \ldots, v_n$ and using these "virtual" names in specifications by a high-level language developer (a little knowledge of compiler translation schemes is required in few cases)

Integration Strategies for other languages

no integration or limited integration Developers are forced to consult the compiler output to find exact MSIL structure and then put appropriate attributes either at language level or at MSIL level

attribute integration The language recognizes XMS attributes and knowing its own translation schemes puts the attributes in appropriate places inside MSIL

language integration The language syntax is augmented with contract expressions which are compiled as XMS attributes

Part V

Dynamic XMS Contracts

Dynamic XMS Contracts

There are two main techniques of code instrumentation for the .NET platform, .NET Profiler API and context-bound objects.

For now XMS uses context-bound objects and able to intercept method invocations and returns. Predicates are evaluated dynamically using .NET dynamic code creation technique.

Example

Consider following C# code:

```
[XMSIntercept]
public class Test : ContextBoundObject
{
  [Process(typeof(XMSProcessor))]
  public void Swap( ref int x, ref int y )
  {
    int z = x;
    x = y;
    y = z;
  }
  ...
```

The specification would be:

$$Pre_F = \texttt{true} \qquad Post_F = x == y_0 \ \wedge \ y == x_0$$

Example

Actual client code:

```
int u = 0, v = 1;
t.Swap( ref u, ref v );
```

Engine outputs:

```
Preprocessing Test.Swap.
Specification found:
Pre=[true]
Post=[x == y_0 && y == x_0]
Precondition : true
Substituted expression : true
Evaluated expression : True
Postcondition : x == y_0 && y == x_0
Substituted expression : 1 == 1 && 0 == 0
Evaluated expression : True
```

Part VI

Applications of XMS

Obvious applications

- Producer-side dynamic testing
- Producer-side static verification
- Client-side static certification

Anonymous Computation

Suppose that a party **A** needs expensive computation to be performed on some private data. **A** is unable to perform the computation locally. Suppose that party **B** is able to perform the computation for **A**.

However, **A** does not want its private data to be revealed to **B** and **B** does not want its algorithm to be revealed to **A**.

Using XMS as a certification framework and .NET Web Services as remote computation layer, **A** and **B** can rely on following **XMS Secure Computation Protocol**:

Anonymous Computation

1. **A** and **B** ask a trusted party, **C**, to make a Web Service, **W**, available to both **A** and **B**
2. **B** publishes its service on **W** together with XMS specification and certificates
3. **A** asks **W** for the specification of **B**'s service, checks if the specification meets his/her requirements and asks **W** to verify that **B**'s service is correct with respect to its specification using XMS Protocol
4. **W** verifies the **B**'s service and sends the verification result to **A**
5. **A** checks the verification status and if it is positive, sends its data to **W** and collects the results

Part VII

XMS Internals

Validation of XMS Certificates

There are three possible approaches to theorem proving and proof checking. XMS does not favour any but currently uses the first one.

- A tactical theorem prover (Isabelle, Coq) can be used for proof construction and proof validation.
- Proofs can be encoded in a metalogic (LF).
- A logical interpreter can be used as a proof checker.

Implementation Details

- Both engines are written in C#
- Static engine is about 1500 lines long relies on some external layers (MSIL Reader, Parser). It currently supports about 70 percent of MSIL opcodes
- Dynamic engine is about 250 lines long

Part VIII

Closing Comments

Future of XMS

- support for more MSIL instructions and builtin predicates (Static Verification)
- other code instrumentation techniques (Dynamic Verification)
- better integration with high-level languages
- other Safety Policies

Availability of XMS

- the XMS engine has not been yet released to public
- complete details (including MSIL formal semantics, Symbolic Evaluator definition and the proof of Security Theorem) will be available in my PhD thesis (expected in few months)