eXtensible Multi Security: Contracts for .NET Platform

Wiktor Zychla
University of Wroclaw, Poland
wzychla@ii.uni.wroc.pl

Abstract

This paper presents XMS – a language independent security framework called *eXtensible Multi Security* which is designed to verify that modules written in .NET languages are safe with respect to *Contracts* Safety Policy. Dynamic verification engine uses code instrumentation to supervise the execution and validate contracts at run-time. Static verification engine is based on the Proof-Carrying Code paradigm where it is up to the Code Producer to construct a *Certificate of Safety* – formal logic proof enclosed in the code – which can be used by a Code Consumer to verify that the code is secure.

Keywords: intermediate language, dynamic verification, static verification, contracts

1 INTRODUCTION

Distributed systems play a major role in today's computer systems. However, the convenience and freedom offered by such systems is sometimes misused. The software and the hardware is a potential victim to a malicious virus, the data is a potential victim to a trojan horse or a spy-software.

In fact, there is a lot of carelessness when dealing with distributed systems. There are critical bugs found even in vital parts of Operating Systems and commonly used applications. It is still very easy to trick the trusting user and make him run a malicious code on his system and it is usually impossible to check the software and decide if it is secure or not.

Alas, over forty years after the Internet has been born, the majority of users still have to believe that the software they buy or download is secure in the sense that it will not do any harm to their hardware and data.

Widely spread antivirus software can detect several thousands of computer viruses. That's good. Alas, it is able to detect only these viruses that are known. That's bad. If a new virus is released, my machine is probably vulnerable again.

Runtime environments can dynamically supervise the code execution and disallow the execution of some potentially harmful activities. That's good. They cannot however make sure that the code runs correctly. That's bad. Even the fancy managed code is not a bit helpful when the banking software steals money from my bank account.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

.NET Technologies 2006
Copyright UNION Agency – Science Press,
Plzen, Czech Republic.

The goal of eXtensible Multi Security (XMS) is to unify various ideas in one coherent and extensible platform. XMS evolved from logic systems that form a powerful certification framework based on a notion of Proof Carrying Code (PCC, [11]).

The original PCC approach focuses on type-safety. However, the type-safety does not guarantee that other important aspects of safety are preserved. In fact, various aspects of security are rather independent. For example, the code can be type-safe but not correct or type-unsafe but perfectly safe from the control flow point of view.

This is where the XMS starts. XMS infrastructure focuses on selected notions of security and applies them to the existing Microsoft Intermediate Language (MSIL) Runtime Environment. XMS is designed in the spirit of .NET platform – digital certificates are **language independent**. Certificates are put in **attributes** and then stored in binary **meta-data** so that they do not play any role in the code execution but instead they can be used in the verification process.

XMS is a Work in Progress – currently about **60%** from over 200 MSIL opcodes are supported by Static Contracts certification tools. Since compilers of some high-level languages use only selected subsets of MSIL opcodes, XMS is yet compatible with some existing high-level compilers, for example the C# compiler.

1.1 Static vs Dynamic Security

In *dynamic* checking, the safety policy is constantly checked at the run time. This of course requires the existence of a virtual machine or a runtime environment that would be powerful enough, in the sense that it could detect any activity that breaks the safety policy. An example of such an infrastructure is the Java Virtual Machine or the Microsoft .NET Framework. Both "supervise" the execution of code, and enforce precise

checks before any potentially dangerous instruction is executed.

On the other hand in the process of *static* checking the code is verified without being actually run. The answer of a static check is always positive or negative, and the code is accepted or rejected. It is impossible to break the execution in the middle, as in the dynamic approach. A static checking does not necessarily need any support from a runtime environment. In this sense it is more general than the other. However, the static security policies are usually less precise because all non-trivial security policies are undecidable. The user must then accept the fact that some programs would be misjudged which means that some perfectly legal programs could be rejected (the opposite situation, where an illegal code was accepted, would be a true disaster and should never happen).

All these observations lead to an obvious conclusion: there is no perfect way to enforce a safety policy. The best what we could probably do would be to put the advantages of dynamic and static checking together in a framework that unifies all advantages of these two.

1.2 XMS = Static + Dynamic Security

This is exactly what XMS is. On one hand, XMS is built to certify .NET code and that is why .NET dynamic security policies are still validated in run-time. On the other hand, XMS is built on the top of the Proof Carrying Code paradigm and that is why the safety policies are verified statically.

Here is a short summary of XMS benefits:

- XMS is designed to certify the MSIL language, one of the most promising and widely used intermediate languages.
- XMS certificates are compatible with high-level .NET languages. A high-level language developer **does not need to know** MSIL to certify the code.
- To support XMS the .NET Runtime Environment **does not need to be changed** in any way.
- 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.

2 COMPARISON TO RELATED WORK

Formal verification of software has long history ([15]). The PCC framework ([11]) was a milestone at this area. PCC was proposed to certify the type-safety of low level languages as the alternative to the TAL ([14]). There are several main PCC research directions:

- exploiting the core of PCC paradigm ([5])
- applying PCC type-safety to industrial environments ([9])
- developing other safety policies for research languages ([1], [2])

For many reasons the type security is strongly desirable for assembly-level languages. In such approach the primary goal of PCC is to validate the language compiler by detecting compile-time bugs. This idea was further adopted to certify the type safety of Java binaries at machine-level (SpecialJ compiler described in [9]).

Initially XMS started as a PCC variant for a toy-like object language. After migration to .NET platform, XMS marks out its own way:

- XMS does not certify type-safety of the low level language but instead it allows to certify other safety policies of the MSIL language.
- Since the certificates can be applied to any high-level language, XMS is more general than solutions bound to a single low-level ([11]) or high-level ([6]) language.
- XMS will ultimately adopt other security policies, such as Non-Interference, to its verification engine

Currently, as a contract verification framework, XMS competes with specialized contract frameworks for .NET Platform like the Spec# ([20]). Major differences between these two:

- Unlike XMS, Spec# is bound to a single language - it is a superset of C#.
- Unlike XMS, Spec# is bound to a single safety policy (contracts). XMS is an extensible framework with pluggable verification engines
- In Spec# contracts are declared using the language extensions and turned into inlined code during the compilation. In XMS, contracts are external to the language (attributes) and code instrumentation techniques are used for dynamic analysis
- Spec# uses its own intermediate representation of the code, BoogiePL, which is interpreted and transformed before it is provided to the theorem prover. XMS uses symbolic evaluation to build verification traces directly from the .NET Intermediate Language code.

3 XMS CERTIFICATES

3.1 Certification Scheme

The PCC certification scheme is based on **Verification Conditions**, logic predicates that contain information

about the execution of programs. XMS Verification Conditions are built by the **VCGen** – a tool that scans MSIL binaries and performs symbolic evaluation of the MSIL code. The Theorem of XMS Safety (3.1) says that **if** certificates are provable **then** properties of corresponding programs hold. Thus, formal proofs of Verification Conditions can be used as **digital certificates**. Such certificates are unbreakable since it is impossible to hack a formal logic system if it is proven to be sound and correct.

The XMS certification protocol assumes that a Safety Policy is shared between Code Producer and Code Consumer. The protocol is shown in Figure 3.

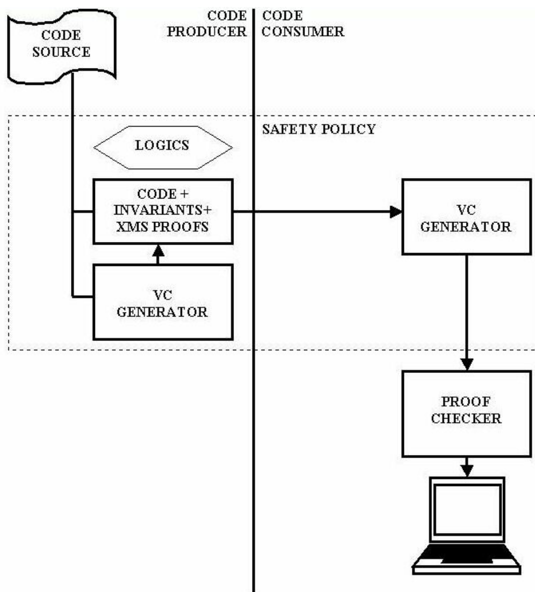

Figure 1: XMS Certification Protocol

The Code Producer and Code Consumer use the same public and verified safety policy that define logic and algorithms to build Verification Conditions for each logic.

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).

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).

Note that the protocol can fail at some point at the Code Consumer side. Specifically:

1. the MSIL binary does 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.

XMS introduces the concept of **Verification Traces**. While Verification Condition is a predicate that captures any execution of a method, the Verification Trace is a predicate that represents execution of a single trace of a method. And while Verification Condition acts like a digital certificate which verifies code correctness, Verification Traces can be used by developers to identify possible invalid execution traces in the code - any Verification Trace that is not provable refers such possible invalid execution sequence.

3.2 Contracts

The **Design By Contracts** paradigm lays the base for systematic object-oriented development [7]. It defines a precise framework where software components can be seen as communicating entities whose interaction is based on mutual obligations. These obligations take the form of predicates: preconditions, postconditions and invariants. It means that the specification of each method must be 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 MSIL instruction numbers to invariants.

Currently XMS supports Eiffel-style Contracts [8] with the complete compatibility (i.a. subcontracting) as the ultimate goal. Contracts are provided in attributes.

3.3 Dynamic Contracts

There are two main techniques of code instrumentation for the .NET platform, .NET Profiler API and context-bound objects. .NET Profiler API is a great way for transparent instrumentation since is it completely decoupled from the source code. It is however COM-based and thus not portable. For now XMS uses then context-bound objects and by implementing `IContributeServerContextSink` interface it is able to intercept method invocations and returns.

3.4 Static Contracts

Both Security Policy for static verification and accompanying Theorem are stated formally using formal operational semantics of the .NET intermediate language (defined for XMS formally by following the .NET draft [21]).

Definition 3.1 (Safety Policy of Contracts). *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* $\rho(Post_F)$*. We will denote this fact as* $Safe_{SC}(F)$.

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

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

$$\rho(Post_F)$$

A module $\mathscr{M}$ *is safe with respect to Static Contracts if all methods from the module are safe. We will denote this fact as* $Safe_{SC}(\mathscr{M})$.

$$Safe_{SC}(\mathscr{M}) \Longleftrightarrow \forall_{F \in \mathscr{M}} Safe_{SC}(F)$$

This formal definition gives the base of the XMS Static Contracts certification. It says that if a method's precondition is satisfied when the execution begins then the method is safe only if the postcondition is satisfied when the execution is about to end. It also says that Static Contracts are *modular* which means that a module is safe only if all its methods are safe.

The underneath theorem is the central part of Static Contracts for XMS. It formally states the soundness of the VC-based certification framework.

Theorem 3.1 (Theorem of Safety for Static Contracts). *If the verification condition for a given module M is valid, i.e. if* $\models VC(\mathscr{M})$ *then all executions of any module methods are correct with respect to their contracts, i.e.* $Safe_{SC}(\mathscr{M})$.

The VCGen algorithm the theorem refers to and the proof of the theorem are long, technical and will not be presented here. Both are inductive on the MSIL instruction set.

Symbolic evaluation of an object language (like MSIL) which is a heart of the algorithm arise several issues:

arithmetics an arithmetic instruction causes VCGen to update its symbolic store to new state.

conditionals a conditional jump causes VCGen to split the symbolic evaluation into recursive paths for all branches. Conditions became assumptions inside the verification predicate.

backward jumps backward jumps could lead to infinite analysis. VCGen requires then that each backward jump targets instructions which have *invariants* provided. Invariants are validated when they are seen for the first time and then validated again when a backward jump is encountered.

method calls a method call makes VCGen to put the method's 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.

objects objects are evaluated symbolically.

arrays an array is stored as a index-value dictionary.

polymorphism is it not known until the run-time which exact method is called from a class hierarchy. VCGen relies here on a *subcontracting* paradigm ([8]) 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 to support such possibility.

3.5 XMS Architecture

XMS Architecture is presented in Figure 2. Both engines (static and dynamic) are written in C#.

The main core of the dynamic verification engine is about 250 lines long and uses .NET context attributes and message sinks to instrument the code at run-time. Expressions are evaluated using .NET dynamic code execution technique.

The main core of the static verification engine has currently 1500 lines of code but uses external parser for specification parsing and external IL decompiler. The symbolic evaluator maintains the state of evaluated code between recursive calls and produces either one Verification Condition or a set of Verification Traces for each method. A simple windowed user interface is provided for user's convenience.

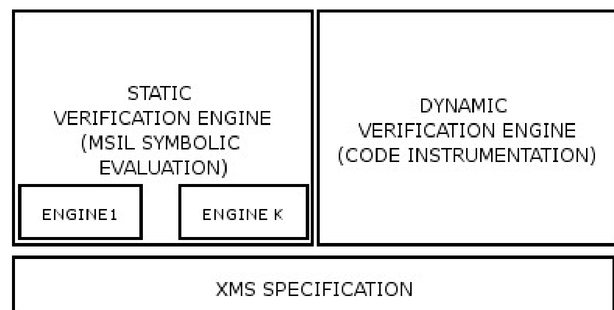

Figure 2: XMS Architecture

3.6 An Example of XMS Certification

To give a glimpse of the XMS framework we present a simple example of an interactive session with the XMS toolkit.

Suppose that the Code Producer has a code of a simple C# method to compute the GCD of two positive integer numbers using the Euclid algorithm:

```
[XMS_Spec(
  "x>=0 & y>=0", "VALUE = GCD(x,y)",
  "GCD(x,y)=GCD(V_0,V_1)", "by auto.")]
public static int GDC( int x, int y ) {
  int k = x;
  int l = y;

  while ( k-l != 0 ) {
    if ( k > l )
      k -= l;
    else
      l -= k;
  }

  return k;
}
```

There is nothing special with the code except for the `XMS_Spec` attribute. It is supplied by the Code Producer and it carries the information about the method's specification:

- the precondition: $x \geq 0 \wedge y \geq 0$
- the postcondition: $VALUE = GCD(x, y)$
- the loop invariant: $GDC(x, y) = GDC(V_0, V_1)$

The precondition establishes an initial assumption on method's parameters. The postcondition characterize method's expected behaviour. The invariant describe a constant condition that is valid in any execution of the loop.

The Verification Condition Generator (VCGen) produces the Verification Condition that captures all essential aspects of arbitrary invocation of the method. Both Code Producer and Code Consumer use VCGen invoked on a binary module:

```
> VCGen.exe gdc.exe
```

It examines the module structure, reads the MSIL binary code and specification metadata and produces the Verification Condition. Note how the specification and conditional branches become assumptions for further parts of the VC.

```
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_)))))))))
```

The last XMS attribute parameter, "`by auto.`" is the proof of the predicate supplied by the Code Producer. In our example this is a simple proof for a tactical theorem prover. The Code Consumer uses this proof to verify the reliability of the C# method – because the proof is correct for the Verification Condition ($\models VC(F)$), the Code Consumer can be sure that *any* execution of the method is safe according to the Static Contracts Safety Policy ($Safe_{SC}(F)$). Ultimately, XMS will allow to use a tactical theorem prover (Isabelle) to shorten proofs or a proof checker (Twelf) for faster validation.

Another example:

```
[XMS_Spec( 0,
  "n >= 0",
  "VALUE=sum(0, n)",
  "V_0=sum(0, V_1) & n >= V_1",
  "", "")]
public int Sum_I( int n )
{
  int sum = 0;
  for ( int k=0; k<=n; k++ )
  {
    sum += k;
  }

  return sum;
  }
```

Produces following Verification Condition:

```
forall n. (n >= 0 => 0=sum(0, 0) & n >= 0 &
  forall V_0_. forall V_1_.
  V_0_=sum(0, V_1_) &
  n >= V_1_=>
    ((V_1_<n => (V_0_+(V_1_+1))=
      sum(0, (V_1_+1)) & n >= (V_1_+1)) &
     (V_1_>=n => V_0_=V_0_ &
       V_0_=sum(0, n))))
```

Let us also look at the example of dynamic verification:

```
[XMSIntercept]
public class Test : ContextBoundObject
{
```

```
  [Process(typeof(XMSProcessor))]
  [XMS_Spec( 1,
    "true",
    "x == y_0 && y == x_0", "", "", "" )]
  public void Swap( ref int x,
                    ref int y )
  {
    int z = x;
    x = y;
    y = z;
  }
}
```

This time the method is declared to swap input values, the return value of x is equal to original value of y (y_0) and vice versa. Actual client code:

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

And the 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
```

4 FROM MSIL TO HIGH-LEVEL LANGUAGES

The .NET paradigm unifies many programming languages at MSIL level. Whether you use C#, VB.NET, Managed C++ or any other .NET language, your code can closely cooperate with any other .NET code.

Since the Verification Condition Generator works at MSIL level, it cannot determine which language was used to produce MSIL binary. And no matter if a binary was produced by IlAsm compiler, C# compiler or any other language compiler, it should be certifiable in the uniform way.

The goal of "lifting" the certification framework from MSIL to a high-level language is then executed under two paradigms:

- 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. Certificates should be then easily applicable to a high-level language code.
- A high-level compiler should not require any major changes to support the certification. In fact, it would be perfect, if the high-level compiler did not require **any** changes. In particular, existing high-level language compilers should not damage certificates that were applied to high-level code.

It seems that comparing to other security policies, Static Contracts is quite difficult to be lifted to high-level languages. There are several important difficulties that have to be addressed:

- Static Contracts Invariants have the form $Inv_F(i) = (P, \ldots)$ 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.

The first difficulty can be addressed with a clever technical trick. We would like to avoid attributing instruction numbers to invariant predicates. We would rather like to have an ordered set of invariants:

$$Invs_F = (P_0, \ldots, P_n)$$

and somehow infer Inv_F from it by mapping consecutive invariants to instructions that need invariants.

This goal can be achieved with additional scan of the binary code which could discover instructions $I = (i_0, \ldots, i_k)$ that are targets for backward jumps.

We could then take:

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

The second difficulty can be 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.

4.1 Common Certificate Specification

Both above technical tricks require that the high-level language satisfies two important conditions. These conditions are **essential** for the "lifting" process to work, so we will formulate them as the **Common Certificate Specification** (by analogy to Common Language Specification and Common Type System, two fundamental .NET paradigms). The Common Certificate Specification is as follows:

Variable Ordering Consecutive high-level language local variables become consecutive MSIL local variables.

Structure of Loops High level language loops become MSIL loops with as simple structure as possible.

While the above specification does not look formal enough, we are not going to make it formal. It is because some important existing compilers (like the C# compiler) fulfill both these requirements, so the CCS formulation should be treated as a set of guidelines for new compilers.

Both requirements are crucial for proper translation of loop invariants between a high-level language and MSIL. In an example from section 3.6 the loop invariant refers to variables `k` and `l` but in MSIL they become V_0 and V_1. Since there is only one loop in C# code, only one loop invariant should be supplied. VCGen will automatically detect instructions which invariants refer to.

In fact, the main reason that makes the "lifting" possible is that .NET high-level language compilers follow few simple and obvious patterns while producing MSIL from high-level code. This is not a coincidence and chances are that future compilers will also behave in similar way because MSIL is not a platform-native language – it is the Just-In-Time compiler which does most of fancy optimizations while translating the MSIL to platform-native language.

Of course this "simple translation with obvious patterns" rule applies mainly to C# and VB.NET, two main business languages for the .NET platform. Other languages with non-trivial translation schemes must find their own way to integrate with XMS. There are three possible **integration strategies**:

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

5 XMS IN PRACTICE

A practical implementation of PCC-oriented certification framework requires three key components: the VCGen that build Verification Conditions for given code modules, a Theorem Prover for Code Producer to build formal proof of a Verification Condition and a Proof Checker for Code Consumer to verify the proof.

The VCGen was exclusively developed for XMS and runs on the .NET platform itself. It reads .NET binaries, scans method bodies and builds Verification Conditions. An example of a session was presented in Section 3.6. Current implementation supports broad range of MSIL instructions, i.a. arithmetical and control flow instructions, instructions for addressing fields and arguments and instructions for calling methods.

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

1. A tactical theorem prover (*Isabelle*, *Coq*) can be used for proof construction and proof validation. Proofs are concise and in many cases can be constructed automatically without any manual guidance. However, the prover must be present at Code Consumer side.
2. Proofs can be encoded in a metalogic (LF [19]). This results in long and detailed proofs but the proof checking procedure is cheap at the Code Consumer side. Metalogic proof checkers are short and thus reliable. Additional techniques can be used to shorten proofs ([12]).
3. A logical interpreter can be used as a proof checker ([13]). Such interpreter uses information about the proof structure provided by the Code Producer but instead of recreating the proof it actually checks if the proof exists at all.

6 SECURE COMPUTATION

One of free benefits of conforming to static verification paradigm with predicates/proofs as certificates is the possibility of using XMS for **Secure 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**:

1. **A** and **B** ask a trusted party, **C**, to make a Web Service, **W**, available to both of them
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

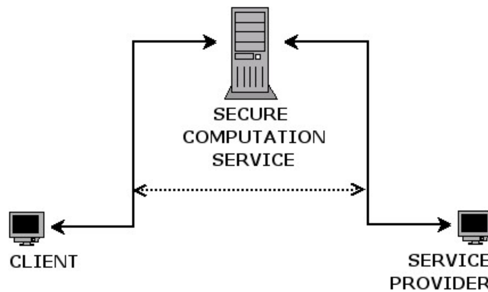

Figure 3: XMS Secure Computation Protocol

7 FUTURE WORK

Formal certificates can rely on other certification paradigms like the Model Carrying Code ([17]) where the certificate takes the form of an abstract model of the code execution and model checking techniques are involved to verify these models. The XMS will ultimately unify various approaches. The combination of PCC and MCC seems especially promising. Three main directions of future XMS development are:

- **support for more MSIL instructions and built-in predicates (Static Verification)**: Currently the static verification does not support all MSIL instructions. It is a short-term implementation goal to support complete MSIL language. For user convenience, some built-in predicates could be supported, such as `ISNULL`.
- **other code instrumentation techniques (Dynamic Verification)**: Although context-bound objects are an easy way to code instrumentation, using .NET Profiler API could make the dynamic verification faster and transparent.
- **better integration with high-level languages**: Current handling of loop invariants require high-level languages to cope with Standard Certificate Specification. This could be restrictive for some high-level languages, for example functional languages with atypical compilation schemes. A long-term goal would be to integrate XMS with such languages using one of proposed integration strategies.
- **other Safety Policies**: Contracts Safety Policy is not the only interesting Safety Policy that can be verified in a XMS manner. Other policies such as Temporal Specifications ([17]) or Non-Interference ([18]) could be adapted to XMS certification scheme.

8 ACKNOWLEDGMENTS

This work is partially supported by MEiN grant 3T11C04230.

REFERENCES

[1] Andrew Bernard and Peter Lee: Enforcing Formal Security Properties, Technical Report *CMU-CS-01-121*, 2001

[2] Andrew Bernard and Peter Lee: Temporal Logic for Proof-Carrying Code, Technical Report *CMU-CS-02-130*, 2002

[3] Andrew D. Gordon and Don Syme: Typing a Multi-Language Intermediate Code, *Microsoft Research*, 2001

[4] Andrew W. Appel: Foundational Proof Carrying Code, *LICS*, 2001

[5] Amy Felty and Andrew W. Appel: Semantic Model of Types and Machine Instructions for Proof-Carrying Code, *POPL*, 2000

[6] Arnd Poetzsch-Heffter and Peter Muller: A Programming Logic for Sequential Java, *ESOP*, 1999

[7] Bertrand Meyer: Applying "Design by Contract", *Computer*, IEEE, Volume 25, Issue 10

[8] Bertrand Meyer: Eiffel: The Language, *Prentice Hall*, 1992

[9] Christopher Colby, Peter Lee and George C. Necula: A Certifying Compiler for Java, 2000

[10] Common Language Infrastructure Specification, *ECMA-335*, 2002

[11] George Ciprian Necula: Compiling with Proofs, *CMU*, 1998

[12] George Ciprian Necula and Peter Lee: Efficient Representation and Validation of Logical Proofs, Technical Report *CMU-CS-97-172*

[13] George C. Necula and S. P. Rahul: Oracle-Based Checking of Untrusted Software, *POPL*, 2001

[14] Greg Morrisett and David Walker: From System F to Typed Assembly Language, 1997

[15] K. Apt and E. Olderog: Verification of Sequential and Concurrent Programs, *Springer-Verlag*, 1997

[16] Mike Gordon: Mechanizing Programming Logics in Higher Order Logic, Springer-Verlag, 1989

[17] R. Sekar, C.R. Ramakrishnan, I.V. Ramakrishnan and S.A. Smolka: Model Carrying Code: A New Paradigm for Mobile-Code Security,

[18] Riccardo Focardi and Roberto Gorrieri: Classification of Security Properties, *FOSAD*, 2000

[19] Robert Harper, Furio Honsell and Gordon Plotkin: A Framework for Defining Logics, *LICS*, 1987

[20] Mike Barnett, K. Rustan, M. Leino and Wolfram Schulte: The Spec# Programming System: An Overview, CASSIS 2004

[21] Microsoft Corp.: Common Language Infrastructure Specification, ECMA-335 Specification