Secure extensible type system for efficient embedded operating system by using metatypes*

Gilles Grimaud, Yann Hodique, and Isabelle Simplot-Ryl
L.I.F.L. CNRS UMR 8022, Université de Lille I, France
email:{grimaud,hodique,ryl}@lifl.fr

Abstract

In the context of extensible system for small secure embedded devices, we present an extensible type system for typed intermediate languages that unifies in a unique hierarchy type systems from various source high level languages and ensures integrity and confidentiality. To increase execution efficiency and use flexibility, we propose a dynamic binding mechanism that allows the programmer to describe the bindings of his code without breaking the type system.

1. Introduction

In this paper we present a dynamic binding mechanism which, once combined with a structured type system, allows system extensibility with a high level of security. This work is dedicated to embedded systems and, more precisely, we focus on systems for small devices that need to be highly secured, like smart cards for example. These systems also need to be flexible, as in the case of cellular phone related smart cards and their applications. To allow software deployment in a previously existing embedded system we must include a mobile code support (code loader, binder, and run-time). Nevertheless, in order to avoid lacks of performance obviously induced when piling up abstraction, it is consequently necessary to raise the operating system up to the highest level by enabling software providers to design the abstraction (components) they need closer to the hardware: in other words, operating system must be extensible. Thus, what we call extensibility is the ability of an embedded system not only to dynamically load new applications but also to load new components of the system itself. To achieve this without breaking the system reliability, we need a reliable extension mechanism.

Usual component bind consists in a clean calling convention, for example when you are linking to C modules. In the case of dynamically loaded components, the binding consists in (i) loading the required component in the run-time environment (in memory) and then (ii) modifying the caller by injecting the appropriated calling convention towards the loaded component. Some Operating System toolkits like Think [7] provide a ”binding factory” scheme which allows dynamic interception of any call. Nevertheless, Operating Systems and for example kernel components define dedicated binding methods to achieve building links with the underlying hardware. Thus, our mechanism has to be flexible enough to allow different binding policies.

Paper overview. Section 2 argues the use of an intermediate language for embedded systems and presents a type system that allows extensibility and security. Section 3 describes a dynamic binding mechanism used to specialize bindings at loading time. The mechanism, combined with our type system, gives a lot of flexibility without breaking the security properties. Section 4 presents with some experimental results.

2. System design

Our notion of reliability is defined by integrity and confidentiality, which are the minimal properties we want to provide to the loaded applications. Most of the time, these properties rely on correct typing of applications and elementary rules that are sometimes expressed in informal specifications as “it is not allowed to cast” or “it is not allowed to forge a pointer”.

In order to offer extensibility facilities but still reliability, we advocate the use of an intermediate language as target for various high-level languages. The intermediate language should be compact. It should be strongly typed to help at verification. At last, it should offer facilities to implement embedded verification mechanisms: the verification has to be performed at bytecode level for all source languages in order to avoid redundance of efforts. Moreover, an embedded system needs to be autonomous in terms of verification. The system reliability does not rely on the application provider nor on a third-party.

* funded by the Mosaques project, CPER TAC 2005-2008 & FEDER

2.1. Intermediate language

Since the apparition of Java, there has been a lot of research works related to the use of intermediate languages, in particular in the area of embedded systems. Intermediate languages are interesting because they can be high-level enough to express properties such as types, that are lost in assembly code, and low-level enough to be almost executable as is. In general purpose computing, intermediate languages are now frequently used (*e.g.* the Java Bytecode, CLI[1] for .net) but modern embedded devices support also mobile code with Javacard[2] and Multos[2] for example.

It is of a great help in embedded systems to have access to high-level properties, since most security policies rely on these ones, and embedded devices are wanted to be highly secure. The intermediate language can be closely bound by the type system it uses. For example the object system of Java is expressed deeply in the Java bytecode, which has a semantic similar to what can be found in the Java language. It is necessary to be able to map directly properties such as types, so that they can be used in embedded verification after loading the code into the device. Anyway, as we want to support many high-level languages, we cannot just specialize the type system to a particular one.

Another advantage is that the intermediate code is more compact than its native equivalent, so that it is less expensive to transfer applications to typically slow devices. As we want our intermediate language to be efficient, we prefer to avoid the use of a virtual machine when possible. That is, we want to have the possibility to compile our intermediate language into native code. This is highly relevant for the "kernel" part of the system: those primitives should be as efficient as possible. As for applications, we let the possibility to have very high-level function calls, that slow down the execution, but make the code more concise.

2.2. Extensibility and efficiency

The type system of our intermediate language must be extensible by the user. A natural way is to make it object-oriented. But we want to avoid the usual drawbacks of this approach. Some fully object-oriented languages, as Smalltalk, pay the price of the object layer even when it is not really needed (from a low-level point of view). For example, when performing an addition between two bytes, it's a pity not to generate a single microprocessor `ADD` instruction. The object concepts are highly time-consuming, mostly in the most basic part of computation. Since the embedded systems are usually low-resources devices, we need to avoid that situation.

1 www.ecma-international.org/publications/standards/Ecma-335.htm

2 www.multos.com

In conclusion, we wish a system that can support and manage any object concept that can be found in high-level source languages. We also want all of them to be unified in a single hierarchy that can be extended as much as needed to accept new elements. For managing the different aspects of our system (highly extensible, focused on efficiency), we chose to introduce a notion of metatype in our type system. The metatypes help us to classify our type system by grouping similar behaviours. They are in charge of inserting an abstraction layer over the object mechanisms to be installed, and also to make the compilation of the intermediate language instructions more flexible, in order to allow the generation of highly optimized machine code when needed.

2.3. Metatypes as types

Metatypes can be built naturally enough via metaclasses (which are often used to build reflexive systems, like Lisp [1]), that is, classes whose instances are classes themselves. The primary purpose of the metaclasses is to provide an abstraction over the type-related mechanisms (mostly object-oriented ones, but not only) that are really implemented at the classes level. For example, the existence of the concept of instance, or the existence of dynamic bindings should be defined in a metaclass. A class would naturally provide the mechanisms defined in its metaclass.

Metatypes are not disjoint: they define some common properties for differents types. For example, the subtyping relation exists in several type systems but is not defined the same way. Our metatype hierarchy is defined using single inheritance so we can take advantage of method override and method inheritance. The concepts expressed in metatypes are the following: i) how the instances of the classes are structured (for example immediate values or references, composite objects or not), ii) what it means to inherit from a parent (for example being able to overload a parent's method), iii) what it means to receive a method invocation on an instance (static calls *vs* dynamic calls).

Figure 1 presents an example of metatype subsystem with square boxes for metatypes and round boxes for types. `Word` is the local top for the metatypes branch, it defines anything represented by a memory word. `Value` is the kind of classes whose objects are not references, `UnBoxed` defines some types whose instances are accessed by references but are statically typed, while `Boxed` is the kind of classes whose objects are references and keep track of their dynamic type. For example the concept of dynamic bindings (also called *virtual methods*) will be defined only for the instances of `Boxed` (until such a feature is needed somewhere else). On the other hand, for instances of `UnBoxed`, we have only statically resolved method calls (since there is no notion of dynamic types). `Class` inherits from `Boxed` but adds a notion of single inheritance. Thus

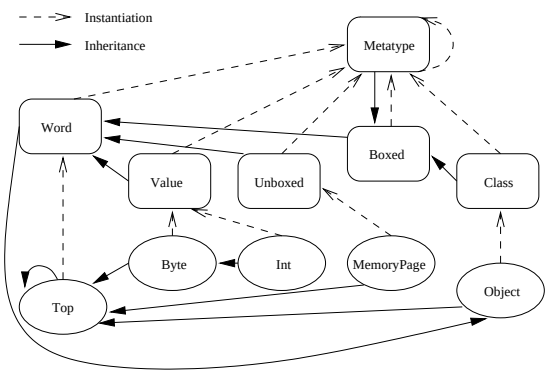

Figure 1. Metatypes hierarchy

the lookup behaviour is fixed. Doing this, we obtain a clean and well-ordered type system that supports different subsystems without interference. The user is allowed to create a new type by inheriting from an existing one.

2.4. Design details

Our common definition of inheritance (shared by any instance of `Word`) is really minimalist: a derived class can be used as a parent class. This can be done through the use of a method `isAKindOf()`, whose behaviour is specified in `MetaType`, the common metatype for all metatypes. This gives a unique upcast system for the whole hierarchy, where traditional languages like Java behave differently for "real" objects and *primitive types*. It is interesting to note that we define inside the metatypes concepts that are usually part of the typing system itself.

A key point in the use of metatypes is also that there is a close relation between the notion of inheritance (at the class level) and the one of instantiation (at the metaclass level). It has to be impossible for a class `B` to inherit from a class `A` if the metaclass of `A` is not a superclass for the metaclass of `B`. Given that, we have a clean separation between incompatible types. One could imagine adding one day a new kind of classes that would support a notion of multiple inheritance, and the previous property would prevent its instances from inserting themselves into the "single-inheritance" branch. By keeping things isolated, we ensure that the properties we proved on a type system will remain valid when adding a new kind of types. The separation of type subsystems has to be ensured in each metatype, since the latter defines its own notion of inheritance. To keep things safe, it is forbidden for a non-privileged programmer to introduce a new metatype: that operation has to be proved safe and consistent. But from the user's point of view, there is no exception to the extension rule: we install that security policy simply by writing the class `MetaType` in such a way that it can't be instantiated once the system is running.

3. Embedded compilation process

To avoid the drawbacks of an object-only system, we want to be able to install things in such a way that the performances at runtime are nearly optimal. We want to resolve at load-time some of the calls that are so heavy when done completely dynamically. To fulfill our need for speed, we can't rely on a virtual machine to interpret bytecodes. Instead, we want to introduce an embedded compilation process that will transform the high-level intermediate language into its executable form. It is vital to fill the gap between the intermediate form and the machine code, at least for critical operations (such as arithmetic ones), that are used permanently during the execution of any program. However, we don't want to impose native code generation: depending on the case, there could be some better policy.

3.1. Mechanism flexibility

We use a load-time binding mechanism of the code that allows the system conceptor to customize the way things are going to be concretized. It could be direct compilation to native code by a call to the "code generator", as well as call to one or several other methods, or even virtual code.

This binding is realized by a dedicated method per class: the `bind()` method is called by the loading process to realize bindings. This method is just a normal one, that is written in intermediate language. It receives under a symbolic form the method of the class that is to be bound, so that only one `bind()` per class is needed. The `bind()` method is not intended to be invoked directly, since the normal type system would not be able to verify symbolic datas. For the programmer, there is a `callBind()` available that wraps around `bind()` and performs needed checks. An important point is that a `bind()` of a class `A` is called each time some other class tries to install some code that manipulates `A` (through an instance for example). The binding does not specify an absolute way of compiling a method, but a compilation that depends on the calling context.

In addition, it is possible to let a `bind()` ignore a method, in which case, the superclass implementation is used instead. Thus, the compilation of high-level code to intermediate code allows us to unify the type systems of the sources languages whereas the loading mechanism allows us to specialize the bindings. This mechanism is interesting from an optimization point of view: it empowers the program to compile itself into the most efficient form. In opposition with traditional languages like `java` that make use of many `native` methods, this binding method acts as a late compiler that is aware of the call context of a method. Furthermore, it makes it possible to fully exploit the capabilities of a platform, by generating specific machine instructions for driving a particular device for example.

If we have access to really low level compilation processes, we are not restricted to it: the binding method can also be used to combine existing bindings, and from there to generate optimized composite operations without rewriting anything but the glue between the components.

Another possibility brought by the binding mechanism is to apply aspect-oriented programming techniques to the program one develops. By writing an appropriate `bind()` method, one can specialize the meaning of a method. The binding provides some kind of basic aspect weaver that can be used in non-trivial cases. It is rather simple to imagine how we could implement a class invariant using `bind()` as shown in the following code

```
void check_invariant() { ... }
void bind() {
  switch(METHOD) { ... }
  invoke(CHECK_INVARIANT); }
```

First, the programmer has to code a method to verify the invariant, and then the `bind()` method of the class does the following: for every call to a method of this class, the real call to the method is followed by the verification of the class invariant. The point here is to note that the assertion is checked *after* each method invocation, and not *at the end* of each invocation, as it would have been easy to do (and less expressive) with a classical `assert()`.

Thanks to this powerful mechanism, the programmer has access to the definition of a binding, and is allowed to customize it. Most of the possibilities are already investigated in other platforms, for example in Java a reflexive Just In Time compiler in [5], method interception in AspectJ in [3], stack introspection for callee monitoring in [8]. All these mechanims are here realized by one mechanism and are not dependent of a source language.

3.2. Security concerns

Naturally, the binding methods are fully responsible for the consistency of the generated code, and a lot of checks have to be performed to ensure for example the soundness of the type system after the code has been turned into native. As a consequence, at the end of every compilation there should be a whole bunch of tests to make sure everything went correct, which is impossible without imposing constraints over the programmer. Without these checks, it would be trivial to bypass the type system and manipulate pointers. For example the programmer could write some methods `int cast(Object)` and `bind()`, that would just compile the first one into a no-op to transform the first argument into an `int` by simply copying it into a variable of type `int`.

```
int cast(Object) { return 0; }
void bind() { ...
  case CAST:
    copy(ARG1, ReturnVariable); ... }
```

To control the access to the code generator, we need an intermediate layer that is in charge of checking the adequation between the signature of the method and the generated code. For each type we use its *metatype* to make sure its behaviour is legal.

3.3. Binding scheme

Metatypes are responsible for the type correctness regarding the target of a method call. For example, a malicious programmer could try to call a *virtual method* on a object whose class is an instance of `Value`, which could lead to pointer generation. In such a case, the metatype has to reject the compilation because the origin of the call is not a valid type for the operation. Here, the current instance is an `int` and not an instance of `A_CLASS`.

```
void cast(int i) { return; }
void bind() { ...
  case CAST:
    callBind(A_CLASS, A_METHOD, ctxt); ... }
```

Let us now expose in details the binding scheme via an example (Figure 2). We consider a method `m()` of a class `C` that wishes to perform an addition between two bytes (an integral type, represented by the class `Byte`). First, the `bind()` method that is in charge of `m()` builds a container `ctxt` and fills it with the informations related to the call context of the method, then checks that the arguments are compatible with the signature of the method.

Since we perform an addition between two bytes, we are calling the method `Add()` of the class `Byte`. The second check to perform is to make sure that the class that will realize the binding (`Byte` here) allows it in `ctxt`. If this is the case, then it is called. The addition has no reason to be forbidden somewhere, so we assume this check is completed. Moreover, it is highly likely that the compilation of such an addition will be realized by a single machine operation, so that the `bind()` method of `Byte` will define a behavior for `Add()`: this is the one that is called.

If we look again at our previous example, the first set of tests are trivially filled, since `cast()` doesn't do anything. The problems arise in the `callBind()` where the attacker tries to call a virtual method on an `int`. The context (`ctxt`) is tested against the signature of the method, and the binding is rejected.

3.4. Discussion

Another motivation for using metatypes is to bring together at a single place the code used to check that a compilation is applied in well controlled conditions and provides conform code. Doing this, we make code safer by avoiding duplication.

Any type checking operation or cast has to be performed in the body of a method that belongs to a metatype. Our security policy ensures that it is illegal for a non-privileged programmer to call a cast. To make things safe we need to make sure that any cast is encapsulated in a system call. So, one has to call the `bind()` of a well-known action that can in turn perform the cast. Those system calls must be the last step before the code is converted to machine code, and loses any notion of type. That policy can be easily implemented using the binding system itself, since it is just a type verification on the caller, which is possible when binding.

Since the last type checking occurs right before the generation of machine code, we have some kind of portable verification: provided that the generator is correct, we don't need to prove anything else to have the consistency of the type system on a new platform.

Though it was not a major point at the very beginning, the optimization possibilities when using metatypes are really interesting.

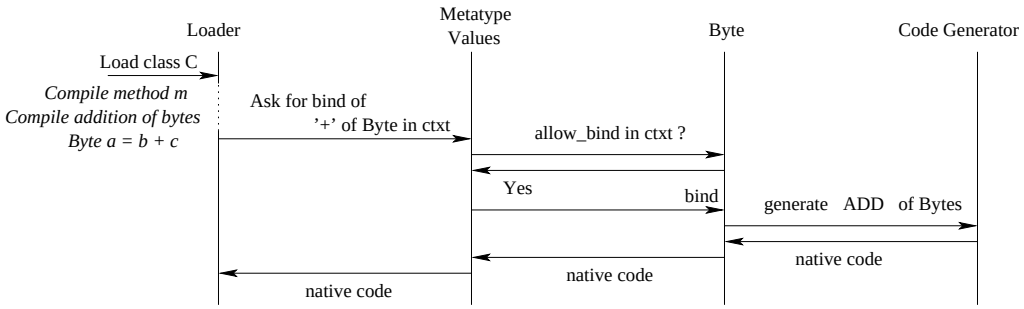

Figure 2. Binding scheme.

First, we came to factorize a lot of code, which reduced the total size. Moreover, by introducing an embedded compilation process, we are able to perform at load-time checks that cannot be static. For example, if one wants to ensure that a given method is never called from a certain class, it cannot be ensured statically, and checking it at runtime is an overkill. At the load-time on the other hand, it is trivial to inject that check in the `bind()` method, so that every subsequent invocation respects this invariant by construction. One can view the load-time as a period where the world is locally closed: there one can perform some kind of semi-static check because the context of the call is available. This technique has two major advantages: first, we avoid the cost of a check at runtime for every invocation of the access-restricted method, and secondly there is no absolute need for complex error-handling since the code that performs illegal calls can be as well rejected.

4. Experimental results

We have implemented the previously described mechanisms in the CAMILLE[3] operating system. CAMILLE is an operating system dedicated to small objects like smart cards. CAMILLE is an exo-kernel providing a minimal number of services that are supposed to be reliable, and an extension mechanism that allows one to: i) load new applications (that is not the historical purpose of smart card operating systems), ii) load new system components in order to extend the system itself.

CAMILLE can load bytecode written in an intermediate language called FAÇADE which is a fully object oriented bytecode. We have modified the implementation to introduce our type system based on metatypes. The advantage of the use of FAÇADE has been shown in [6]: the CAMILLE system is able to verify that the loaded code is type correct in linear time using a PCC [4] like mechanism. So, the embedded system is able to verify that the loaded code is well-typed without any assistance and, using the dynamic binding mechanism describe in the previous section, we can offer specialization facilities without breaking this benefice.

Before the start of this work, there already existed a flexible binding mechanism, but it could not be available to the programmer without potentially breaking the security of the system. One benefit of our work is to make possible for anybody to write powerful (yet controlled) binding policies to suit her needs. Additionally, by centralizing the security-related stuff in a well-controlled place (the metatypes), we reduce the number of paths to the core system, and thus reduce the risk of observing unexpected behaviours.

Our implementation of the metatypes layer on CAMILLE required the introduction of 6 additional types (presented in Figure 1), but no change on the bytecode. Those classes contain the checks that are performed to ensure the security of applications when a new class is loaded. The code of the metatypes represents around 10% of the total code of the CAMILLE platform at the moment, including the type system basis, and also all checks and casts to ensure its security. The amount of code dedicated to the metatype level is not too high, provided that there is no applicative code, but only the exo-kernel part.

3 www.lifl.fr/RD2P/CAMILLE

References

[1] D. G. Bobrow and G. Kiczales. The common lisp object system metaobject kernel: a status report. In *Proc. of ACM conference on LISP and functional programming (LFP '88)*, pages 309–315. ACM Press, 1988.

[2] Z. Chen. *Java Card™ Technology for Smart Cards : Architecture and Programmer's Guide*. The Java™ Series. Addison Wesley, 2000.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[4] G. C. Necula. Proof-carrying code. In *Proc. of 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, Paris, France, 1997.

[5] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda, and Y. Kimura. Openjit: An open-ended, reflective jit compiler framework for java. In *ECOOP 2000 - Object-Oriented Programming*, pages 362–387. LNCS 1850, 2000.

[6] A. Requet, L. Casset, and G. Grimaud. Application of the B formal method to the proof of a type verification algorithm. In *Proc. of HASE*. IEEE Press, 2000.

[7] C. Rippert and J.-B. Stefani. Building secure embedded kernels with the think architecture. In *Proc. of the workshop on Engineering Context-aware Object-Oriented Systems and Environments – OOPSLA*, Seattle, USA, 2002. IEEE Press.

[8] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for java. In *Proc. of the 16th symposium on Operating Systems Principles*. ACM Press, 1997.