A Web navigator with applets in Caml

François Rouaix*
INRIA

Abstract

This paper reports on the implementation of applets in Caml Special Light, a strongly typed functional language. The experiment shows the major contributions of strong typing and powerful module systems for supporting secure execution of mobile code. Security checks are almost entirely performed once at compile-time, and authentication proves that checks have actually been done.

Keywords: mobile code, strong typing, authentication

1 Introduction

A hot topic in Web-related research and development is *mobile code*, an approach where programs are considered as documents, and should therefore be accessible, transmitted and displayed (i.e. evaluated) as any other document. Mobile code is integrated in the Web, and in particular in HTML documents, through *applets*, a form of hypertext links refering to invocation of functions implemented in a mobile program.

This paper reports on the implementation of Caml Special Light applets in the MMM browser. Caml Special Light is a strongly typed functional language of the ML family. This work shows how some results of language design and type system research provide direct elegant solutions to problems raised by applet technology.

The paper is organised as follows: after stating some basic facts about the MMM browser, where this approach has been implemented, we discuss the principles of applets, the issues raised by their design; then we present the solutions we propose using familiar results on type and module systems, together with authentication. Finally, we discuss related work and give concluding remarks.

*Author's address: Projet Cristal, INRIA Rocquencourt, BP 105, 78153 Le Chesnay, France. E-mail: Francois.Rouaix@inria.fr

2 Facts about MMM

MMM is a Web browser implemented in Caml Special Light[11], using libraries for Unix system calls[9], and for the Tcl/Tk toolkit[13]. The browser is able to display asynchronously several documents in several windows. MMM implements full `HTML 2.0` [2], including forms and in-lined images, and supports the applet system described in this paper.

The CamlTk library is based on a small C interface for calling the Tcl/Tk `eval` function directly, without producing full Tcl phrases that would have to be parsed by Tcl. The various commands and functions of Tk are made available in Caml through library code generated from a high-level description of the corresponding Tcl/Tk functions. This interface transforms, as much as possible, the untyped string-based programming style of Tcl/Tk into a typed, functional style.

The system represents less than 10000 lines of source code, in about 50 modules. Its architecture makes it easy to extend the browser to support new protocols, evolution of HTML 3.0, or to change the behaviours of the history or cache mechanisms. MMM is portable to virtually any Unix/X Window platform (currently known to run of `DEC/OSF`, `SunOS4.1`, `Solaris`, `SGI IRIX`, `Linux`, `FreeBSD`).

3 Applets

An *applet* is a program dynamically loaded in the browser and executed in the browser context. More generally, applet support means that the browser has the capabilities of retrieving code from a distant host, linking it dynamically, and executing some functions in that code.

3.1 Issues

There are two main issues when designing applet support.

portability: Applets should be fully portable, that is, architecture and operating system independent.

security: Applet security means that one can load and execute a foreign program without fear that the program will crash the browser, or destroy files on the client machine, or send private data on the network, ... Security has many facets: it includes typical problems of invalid memory access due to bad typing, that sometimes can be turned into security holes. Other holes come from executing library code with undesirable

effects (deleting files, overwriting files, reading files and sending them on the network, executing arbitrary programs, ...). Finally, the security hazards known as *denial of service* exist here, since an applet can do arbitrary computation, or simply consume so much resources that it makes the browser or the machine unusable.

3.2 Solutions

Apart from the pure networking part of applets (using the standard HTTP protocol), all other issues fall in the category of language design, be it in the compiler backend or in the front end and environment (types, modules).

3.2.1 Bytecode compilation

Portability, although easily obtained using source code and interpreters, is best achieved with bytecode compilers: some advantages are reasonable efficiency and protection of source code. Caml Special Light has two compilers: one producing native code (not used here), and one producing bytecode (enhanced version of [8]). The bytecode is independent of the hardware architecture (32 or 64 bits, little or big endian). Caml programs compiled to bytecode are 5 to 10 times slower than programs compiled to native code[7].

3.2.2 Strong typing, modules

Many security problems are solved by strong typing, since it guarantees safety of memory access. Here, we mean by strong typing a mathematical (proven) property of the language that the value computed by a program accepted by the compiler has the type statically determined for this program. In particular, this means that the compiled program cannot fail or produce system errors. Type-checking during separate compilation must of course be complemented by safe linking.

Unsafe library calls must not be made available to applets, and this can be achieved easily if the module system controls the functions exported for linking. To be able to write reasonably powerful applets, the author should have access to a safe subset of "standard libraries" of the language. Then, to interact with the browser, a subset of the browser internals should be exposed to the applet. Safe libraries are discussed in more details in section 4.

Caml Special Light fullfills all these requirements: as other languages of the ML family, it is strongly typed; its module system, based on [10], provides flexible control on exported functions; type-safe linking relies on checking that modules required by a compilation unit match modules provided by

the program to which it is linked. This verification is based on equality of MD5 digests of compiled module interfaces. Digests avoid adding complete compiled interfaces, which can be quite large, to compiled bytecode files. The only important difference between traditional linking and foreign bytecode linking is in the initial set of exported modules (all modules vs. safe modules).

Dynamic linking in Caml Special Lightcan be configured so that only a given set of modules (the *safe* modules) are exported for linking with mobile code.

The only remaining risk is premature termination of the browser due to an exception raised by an applet, or more generally by the foreign code. Exceptions are widely used in languages of the ML family, for pattern matching failures, range checking of some operations, efficiency in some algorithms such as searching, and error reporting. For this reason, we decided not to attempt any restriction on exception raising by applets. Instead, we catch exceptions raised by applets by trivially encapsulating the applet invocation. Moreover, an applet can install callbacks that will later be called directly by the toolkit event managing loop. Thus, this loop is protected against exceptions, although normally it doesn't have to be.

3.2.3 Authentication

All previous security guarantees, based on strong typing and module systems, are invalidated if the bytecode is not the unmodified output of an unmodified compiler. Someone could decide to write bytecode by hand, or tamper with the output of the compiler, and thus defeat the linker verifications.

A first solution, as in Omniware [12], would be to run foreign programs in a special environment with runtime checking, at a very low level (memory access). Other approaches where applets are transmitted in source form ([1, 15]) also require special interpreters with run-time checks.

A second solution, used for Java [5], is to transmit type information with the bytecode, and check that the bytecode is "well-typed" w.r.t. this type information (see below).

In our approach, we decided to use authentication as a proof that the security checks, relying entirely on the properties of the type and module system, have actually been made at compile-time.

In practice, applets are stored on a server as bytecode files, authenticated using PGP, a widely recognised cryptographic tool. The user of the browser is prompted each time some foreign bytecode has to be loaded. The signature of the applet is shown, so the user can decide to accept or reject the bytecode depending on the trust granted to the author of the signed file. This mech-

anism can be refined by installing trusted compilers on publically accessible trusted sites. The applet author would develop and test an applet locally, then send the final version (in source form) to a trusted compiler who will compile it and sign the produced bytecode with its own signature. Finally, the signed bytecode file is made available on the author's server. Independent trusted compiler services limit the number of sources to be trusted by a user.

More generally, the authentication approach is radically different than previously proposed approaches, since the security checks are performed at compile-time instead of load-time or run-time. This has several advantages:

- speed: checks are made only once. There remains only the final authentication check at load-time, presumably less costly than full bytecode analysis (or extensive run-time checks) for applets beyond toy examples.
- evolutivity: the security checks, here restricted to type checking, can be extended to other program analysis (which are more studied and understood on source programs than compiled programs). Moreover, the cost of security checking can grow arbirarily since it is done only once by the applet author and/or by the authentication source.
- flexibility: the amount of security checking (notably restrictions on available libraries) may depend on the authentication source, such that, for example, applets developped by known sources may have access to more resources and functions than applets from arbitrary sources.

The approach could even by used for arbitrary binary programs, using the signature as the only criterium of trust, although of course portability would be lost, as well as the advantages of strong typing as a proof of program soundness.

4 Safe libraries

A "safe" applet environment is obtained very easily from existing libraries by the following construct, called *module thinning*:

```
module Foo = (Foo : sig
   type t = Foo.t =  the type definition
   val f :  some type
   ...
   end)
```

where Foo is a module provided by the browser. This definition produces a restricted version of the original module Foo, exporting only the types and values specified in the signature expression `sig ...end`. Exported types can be left abstract, or fully specified, if the value constructors must be made available.

With this construct, there is in most cases no need to rewrite libraries to provide a safe version. The exported Foo module contains (a subset of) the values of the original Foo, so there is no duplication of code.

Other components of the safe environment may have to be encapsulated. For example, a safe IO module for file operations (opening, closing, ...) has the same interface as the standard IO module, but an implementation where each potentially dangerous operation is subject to confirmation by the browser's user.

Currently, applets in MMM have access to several *safe* libraries, including:

- all the standard data structure libraries (lists, arrays, sets, hash-tables, lexers, ...);
- a library of safe IOs, where opening files is subject to confirmation by the user;
- almost all of Tk functions (except those who would allow an applet to control widgets that it has not created itself);
- browser internals; this allows an applet to add new protocols, to parse HTML, to define new semantics of link activation, to add menus, to invoke navigation functions, etc...

5 Related work

5.1 Obliq

Obliq[3] is an untyped interpreted language with distributed scope. While applets as presented here only allow the transmission and the execution of compiled code (i.e. a compilation unit), Obliq offers a much more powerful mechanism. Computation is distributed (closures migrate, instead of compilation units), and network transparent (network references are handled by distributed lexical scoping).

5.2 Java

Java is a concurrent object-oriented language close to C++. Several notions of C or C++ have been removed, notably pointers and pointer arithmetic, structures and unions (only objects and arrays remain), and coercions. The only functions are object methods. Java has some form of strong typing (type errors may still be detected at run-time), as well as a bytecode verification algorithm. This algorithm perform various checks (e.g. validity of object field adressing, data conversion, ...) with the help of type information remaining in the bytecode (e.g. different instructions for integer and pointer loading, field access by name), and type information relative to class interfaces, attached to the bytecode. A separate class name space is used for foreign code to avoid built-in class spoofing.

The bytecode verifier relies on two simple requirements on the abstract machine instruction set, to be able to construct a "proof" of security by induction on the code. It remains to be understood what limitations these requirements impose on the Java compiler, and what impact they may have on performances. The Java virtual machine is presented as a possible target for other programming languages, but the type information in the bytecode and associated interfaces, especially because of its semantics, may limit the kind of languages for which this goal is reachable. One could also comment that it might not be necessary to remove so many constructs from an object-oriented language to get this form of strong typing. In particular, recent research [6] in type systems for object-oriented language indicates that objects could be added to a strongly typed language of the ML family.

5.3 Omniware

Omniware is a proposal for an uniform virtual machine, with translation to native machine code. The virtual machine has been used as a target for C and C++ compilers. A run-time environment is also provided with support for concurrency, IO and graphics. The basis of security in this approach is a technology called *software fault isolation*, and can be compared to an MMU-type memory protection. Checking of resources access are added during the translation of bytecode to native code. Security should therefore be built and conceived upon very low-level mechanisms. The emphasis in this work in the performance of code obtained after the translation, so that arbitrary C applications, including interpreters for other languages, can run with satisfactory speed.

5.4 Browsers

Various teams are currently working on applet support in Web browsers, although no work has been published at the time of writing. Experimental prototypes have been made available for testing : we know of Surflt[1], entirely written in Tcl/Tk, where applets are Tcl source code. The main weakness here is that of source code transmission, especially Tcl code, where issues of typing, modules (or name spaces) are left open. There is also Grail [15], written in Python, using Tcl/Tk for the user interface. Here again, Python is an interpreted dynamically typed language, and thus security could only be achieved with special runtime environments. Finally, the GNU project has announced[14] work on the TkWWW browser to support safe downloadable extensions using Guile.

With a dynamically typed interpreted language, a safe runtime environment has to check almost any operation performed by the program: access to data structures must be checked, since values may be of inadequate types. Moreover, value spaces must be severely controled if arbitrary pointers can be forged, for example by converting integers.

6 Discussion

6.1 The MMM experiment

Our implementation of applets in Caml Special Light proposes a number of solutions, based on direct application of research results. Several applets have been developped, including graphics animation, interactive graphics, and browser extensions. There are currently two areas where more work is required: concurrency and support for versioning.

The Caml Special Light system used for MMM does not support concurrency. Although one can simulate some amount of concurrency with event loops and first-class functions (explicit continuation passing style), true concurrency would help in avoiding "denial of service" attacks, by having each applet run on a separate thread, and providing support to kill threads. The latest Caml Special Light compiler supports threads (at the bytecode level), so we can expect to exploit them in future versions. Also, an experimental version of Concurrent Caml Light has been developped on top of an earlier release of the compiler, as well as a certified concurrent garbage collector [4].

Versioning of the applet programming interface (API), defined by the various "safe" modules that are exported by the browser, has not been fully studied. It is an important issue to ensure compatibility of future versions of

the browser with older applets.

A first solution is to add explicit version information to module names of the API. Further releases of the browser would then export all versions, using the module *thinning* construct presented above. An applet would always use explicitly a given version of the API. Another solution would be to change the semantics of type safe linking: instead of checking equality of module interfaces required by an applet with module interfaces provided by the browser, one could check for inclusion. This scheme would allow extending interfaces of the safe environment by adding new functions for example, but requires attaching complete compiled interfaces to bytecode files.

6.2 Which language for applets

Applets have a lot of applications, many of them still to be discovered. The first experiments, written in Java, consisted in adding animations inside HTML documents. More generally, applets provide a notion of "active" documents, and alleviate the number of network connections and the load on servers in some applications. The main interest of applets is probably a new approach to software distribution, because of portability and absence of installation procedure. It could be feasible, in principle, to distribute full applications as mobile code. This might be unrealistic if the applet language is not powerful enough to write large programs.

We argue that source based applets (i.e. untyped interpreted languages) do not provide an adequate platform for large scale programs: security relies on ad-hoc restricted interpreters, yet slower than normal interpreters. Moreover, some applets providers would prefer not to distribute the applet source code.

Once taken the decision of choosing bytecode compilation, all other issues regarding language design remain, as well as the choice of *when* the security checks are performed. Our use of strong typing and bytecode authentication to guarantee security has the advantage, over the Java approach, to keep a full fledged language available to the applet author, instead of a core language whose constructs and compilation are restricted by security considerations. Moreover, we believe that security checking by program analysis (in particular type checking) is more promising when working on source programs, which retain some logical structure, than on compiled programs. The current state of the art also provides much more results on source analysis than bytecode analysis.

7 Conclusion

Applets are the major technical innovation of the Web in 1995. A browser supporting applets can safely download and execute programs compiled and stored on a distant site, adding a new dimension to the Web by allowing active documents, and pioneering a new approach to application distribution. Applet support involves a number of issues in language design and implementation: compilation issues, with the necessity of bytecode compilation to guarantee hardware architecture and operating system independence; separate compilation, so that an applet only includes its own code, and not the library it relies on; dynamic linking, complementing separate compilation; security, in several aspects, including type safety, but also control of environment.

Our work shows that the Caml language offers good solutions to all these issues. Using authentication, we avoided doing security testing on the client side, and relied on the existing, unmodified compiler, exploiting its well-defined type and module systems to provide comprehensive security checking.

MMM is available at `<URL:http://pauillac.inria.fr/~rouaix/mmm/>`.

Thanks: The idea of using Caml for applets was first formulated independently by Bernard Lang and Gilles Kahn. Pierre Weis gave me lots of encouragements. Xavier Leroy implemented the dynamic link library of Caml Special Light. Bytecode authentication was designed during discussions with Damien Doligez, Xavier Leroy, and Charlie Krasic.

References

[1] Steve Ball. The SurfIt! browser. WWW address `<URL:http://pastime/anu.edu.au/SurfIt>`.

[2] T. Berners-Lee and D. Connolly. Hypertext Markup Language – 2.0. HTML Working group, IETF Internet Draft, August 1995.

[3] Luca Cardelli. A Language with Distributed Scope. In *Proceedings of the 22nd Symposium on Principles of Programming Languages*, pages 286–297, jan 1995.

[4] Damien Doligez. Conception, réalisation et certification d'un glaneur de cellules concurrent. Thèse de doctorat, Université Paris 7, May 1995.

[5] James Gosling and Henry McGilton. The Java Language Environment. White Paper, Sun Microsystems, May 1995.

[6] C. A. Gunter and J. C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design.* The MIT Press, 1994.

[7] Pieter Hartel, Marc Feeley, et al. Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 1996. To appear.

[8] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1990.

[9] Xavier Leroy. Programmation du système Unix en Caml Light. Rapport technique 147, INRIA, 1992.

[10] Xavier Leroy. Manifest types, modules, and separate compilation. In *21st symposium Principles of Programming Languages*, pages 109–122, Portland, Oregon, January 1994.

[11] Xavier Leroy and Damien Doligez. The Caml Special Light system. Software and documentation available on the Web, `<URL:http://pauillac.inria.fr/csl/>`, 1995.

[12] Steven Lucco, Oliver Sharp, and Robert Wahbe. Omniware: a universal substrate for web programming. In *Proceedings of the 4th International World Wide Web conference*, pages 359–368. O'Reilly and Associates, 1995.

[13] John K. Ousterhout. *Tcl and the Tk Toolkit.* Number ISBN 0-201-63337-X. Addison-Wesley, 1994.

[14] The GNU Project. The Guile architecture for Ubiquitous computing. WWW address `<URL:http://www.cygnus.com/library/ctr/guile.html>`.

[15] Guido van Rossum. The Grail browser. WWW address `<URL:http://monty.cnri.reston.va.us/grail/>`.

[16] Pierre Weis and Xavier Leroy. *Le langage Caml.* Number ISBN 2-7296-0493-6. InterEditions, 1993.

[17] Brent B. Welch. *Practical Programming in Tcl and Tk*. Number ISBN 0-13-182007-9. Prentice Hall, 1995.

[18] Frank Yellin. Low Level security in Java. In *Proceedings of the 4th International World Wide Web conference*, pages 369–379. O'Reilly and Associates, 1995.