Flexible Bindings for Type-Safe Embedded Operating Systems

Damien Deville, Christophe Rippert, Gilles Grimaud

IRCICA/LIFL, Univ. Lille 1, UMR CNRS 8022
INRIA Futurs, POPS research group*

{Damien.Deville,Christophe.Rippert,Gilles.Grimaud}@lifl.fr

1 Introduction

This paper presents the binding model implemented in Camille, an extensible operating system for resource-limited devices. Modern embedded systems need on the first hand to fully exploit the limited hardware on which they run and on the other hand to dynamically adapt themselves to changes in their runtime environment. Camille is an exokernel which support static customization of components and dynamic loading of system extensions. Dynamic kernel and application adaptation is implemented by an inter-component communication model. This model is based on flexible bindings which permit to fully customize the way components interact with each others. Bindings can be static, virtual or compiled to guarantee performances of inter-component communications. This paper shows that it is possible to build a flexible operating system without sacrificing runtime performances, even for devices as constrained as smart cards.

We first present the architecture of the Camille exokernel and the intermediate language Façade into which applications and system components are translated to ease type verification. We then describe the component model implemented in Camille and the inter-component communication scheme based on embedded binding factories. We then details the binding generation process and the various verifications which can be enforced when implementing bindings. We present some experimental results we have obtained when monitoring the performances of our native code generator. Finally, we conclude and discuss the future work we plan to conduct concerning extraction of selected properties from generated code.

2 The Camille exokernel

Camille [1] is an extensible operating system designed for resource-limited devices, such as smart cards for instance. It is based on the exokernel architecture [2] and advocates the same principle of not imposing any abstractions in the kernel, which is only in charge of demultiplexing resources. Camille provides secure access to the various hardware and software resources managed by the system (*e.g.* the processor, memory pages, native code blocks, etc.) and enables applications to directly manage those resources in a flexible way.

Camille is designed to ensure portability, extensibility, confidentiality and integrity of applications and system extensions. Portability is guaranteed by using Façade [3], an object-oriented intermediate language. Application and operating system extensions which are programmed using a high level language as C or Java for instance, are translated into Façade by a code

*This work is partially supported by grants from the CPER Nord-Pas-de-Calais TACT LOMC C21, the French Ministry of Education and Research (ACI Sécurité Informatique SPOPS), and Gemplus Research Labs.

converter or a dedicated compiler, before they are loaded in the embedded system. CAMILLE currently includes a Java to FAÇADE converter and a backend for generating FAÇADE from C using `GCC`. Embedded operating systems built with CAMILLE are extensible since both applications or system extensions can be loaded dynamically. To avoid the overhead usually induced by interpreting an intermediate language, CAMILLE includes an on-the-fly native code generator which compiles extensions while they are loaded in the system. Finally, confidentiality and integrity of extensions are guaranteed since FAÇADE supports the Proof Carrying Code model [4]. Extensions are thus validated when loaded in the operating system by a verifier, which ensures their type-correctness. The type-checking algorithm has a linear cost thanks to the simplicity of the FAÇADE language [5].

CAMILLE itself is written using a type-safe subset of the C language. The kernel is made of a set of components whose only task is to safely expose the hardware resources. The kernel forms a trusted computing base and can be translated in FAÇADE using a customized version of `GCC`[1]. FAÇADE is a very simple and compact intermediate language composed of only five instructions[2]. FAÇADE includes three control flow instructions (`jump`, `jumpif` and `jumplist`), and two instructions for method invocation (`return` and `invoke`). Since all operations are mapped as method invocations, a FAÇADE program can be seen as a composition of bindings between the components implementing the operations.

3 Binding generation in Camille

CAMILLE supports various types of bindings, including fully dynamic and static ones. All bindings are described by the `invoke` instruction in FAÇADE, but their implementations can vary. For instance, components can interact using virtual method calls, static method calls or through *external bindings*. An external binding can be seen as an inline of the callee method code into the caller component, which usually speeds up greatly the execution of the operation. Bindings are implemented when translating extensions from FAÇADE to native code using the CAMILLE embedded compiler. Bindings can be translated to native code using a linear algorithm thanks again to the simplicity of FAÇADE. This binding creation scheme differs from standard component-based operating systems. In the THINK architecture for instance [6], bindings are created through explicit calls to dedicated components called binding factories and are typically instantiated by using interceptors to permit dynamic replacement of components. CAMILLE advocates a different approach, since bindings can be either specified at source-level as in THINK, or defined using customization tools and instantiated through automatic code generation. The definition of bindings through system customization tools is simplified thanks to the modular approach used both for the operating system and the applications. Similarly, FAÇADE eases the instantiation of bindings since tools only have to generate an `invoke` instruction, which will be implemented as needed by the embedded compiler.

Each component in CAMILLE provides a method `bind` which is in charge of checking the legitimacy of binding creating requests and generating the type safe native code implementing bindings to this component. This method materializes the concept of binding factories in the CAMILLE component model. It is called when a method using a binding is loaded and compiled. Several components are involved in the compilation of bindings. For every compiled method, a persistent instance of component `Code` is created. This instance contains the signature of the method and the generated binary code. During the compilation of the loaded component, a transient instance of component `CompileContext` is also created. This object includes all the necessary information for translating the FAÇADE code of the method into an executable format (which can be either binary or a bytecode interpretable by the system). For instance, it can

[1] This means that the kernel itself can be dynamically loaded and translated into native code when bootstrapping the system.

[2] FAÇADE can be seen as a "RISC for bytecode" approach.

```
L_muladd: // d <- s muladd a b
  // read DestVar
  d <- CardCode ReadVar  stream

  // d <- s mul a
  CardInt bind src  dest code cc stream MUL  // read a in stream

  // d <- d add b
  CardInt bind dest dest code cc stream ADD  // read b in stream
```

Figure 1: Composite binding for a `MulAdd` operation.

provide a translation table to map FAÇADE variables to hardware registers, which can be used by the register allocation algorithm. Instances of `CompileContext` also contains type information used by the PCC verifier. Operations related to code generation are defined by intances of component `Code` and are implemented in instances of component `CodeN` to generate native code and in instances of component `CodeV` to generate bytecode. Generating bytecode destined to be interpreted, instead of native code, can be relevant if reducing the footprint of the generated code is more important than the runtime performances of the method.

FAÇADE code is processed through an instance of the abstract component `Stream`. It is simple for instance to produce an input stream from the embedded communication serial line. Loading a new extension consists in creating an instance of component `Code` with an instance of `Stream` as a parameter. For every FAÇADE instruction, the instance of `Code` determines which component implements the associated operation. The `bind` method of this component is then called to implement the binding. For instance, when compiling the simple expression `dstByte <- srcByte + #1`, component `Code` uses the type information associated with the `srcByte` object to determine that it is an instance of component `Byte`. Thus, it is the `bind` method of component `Byte` which is used to implement the addition. In this simple case, the operation can easily be externalized and so inlined in the code of the method performing the addition, by using the instance of `CodeN` to generate the binary code.

For more complex operations, bindings can be implemented as static or virtual method calls. A component inheriting a method from one of its parents can also choose to delegate compilation of bindings to this method by calling the `bind` method of its superclass. Thus, compilation of a binding can result in a lookup of the `bind` method which will really compile it, in a way very similar to the lookup done when calling a virtual method in standard object-oriented languages, but realized here during linking. In this way binding policies can be inherited.

Compilation methods can also be overloaded when creating a new component, to customize or perform additional checks when linking two components. Another interesting example would be to increment a counter each time the `copy` operator of a component is called (this operator is used to copy the reference of a component to another). Thus, it would be possible to check whether the component is still referenced before unallocating its memory space, thereby preventing some null pointer exceptions. Finally, compilation methods can be composed to produce more complex bindings. For instance, a programmer needing an operation to compute the `dst <- src * b + a` expression[3] simply needs to implement a binding calling the `bind(mul, ...)` method followed by a call to `bind(add, ...)` as illustrated in Figure 1.

The binding factories in CAMILLE allow enforcing a fine control on binding creation. When compiling a binding, the destination component knows from which component the binding request comes thanks to the `CompileContext` component as explained before. Thus, the destination component can check if its protection policy allows the creation of the binding before compiling it. Standard protection mechanisms as Access Control Lists or capabilities can be implemented in the kernel to manage access control policies. Some verifications can be enforced statically when compiling the binding, and interceptors or verification stubs can be used when dynamic checks are required. Similarly, `bind` methods can use different compilation schemes de-

[3]This $y = a \times x + b$ expression is frequently used for cryptographic protocols implemented in embedded systems.

pending on which component requests the binding. For instance, a `bind` method can choose to skip some access control tests if the binding request comes from a trusted component validated by a certificate. Finally, the `bind` methods can be used to inject security mechanisms in the generated binary code.

4 Experimental results

We present in this section some experimental results which validate the approach advocated in Camille. We implemented a prototype of Camille on a platform based on an AVR chipset from Atmel. This chipset includes a 8/16-bit CPU, 32 KB of ROM, 32 KB of EEPROM and 1 KB of RAM. Our prototype of Camille uses 17 KB of memory, including 3.5 KB for the PCC verifier, 8.5 KB for the native code generator and 5 KB for hardware management. These values demonstrate that the Camille architecture is suitable for very constrained devices such as smart cards. We first monitored the binding creation and code generation processes, then we evaluated the performances of the generated code for a high-level system extension (a smart card file system) and for a low-level random bit generator.

4.1 Evaluation of the code generation process

Table 1 presents the results we obtained when compiling all the Façade code composing the kernel. It first shows that almost half of the code generation delay is due to writes in EEPROM, which are a typical bottleneck on smart cards (dedicated software cache policies are usually used to circumvent this bottleneck). Thus, the core of the code generation algorithm is fairly efficient, which can be explained by the fact that the Façade language is close enough to the hardware to be easily translated into native code. This is illustrated by the figures showing that an average of only three AVR instructions are needed for each Façade instruction.

	Average	**Deviation**
Number of cycles to process a Façade instruction	80727	26926
Number of cycles ignoring writes in persistent memory	37858	4596
Number of AVR instructions per Façade instruction	3.05	1.04
Size of the code compared to Java Card byte code	$\times$ 1.8	$\times$ 0.6
Size of the proof compared to Façade code size	30%	3%

Table 1: Performances of the code generation process.

Table 1 also shows that the Façade code is almost twice as big as Java Card bytecode, which is clearly a drawback on a memory-constrained platform. However, approximately one-third of the size of the Façade code is in fact occupied by the proof of its type-correctness, which is checked dynamically when extensions are loaded in the system.

4.2 Performances of the generated code

We then monitored the performances of a high-level system abstration loaded in Camille. We chose to implement an ISO7816-4 compliant file system so as to be able to compare its performances to an implementation in Java Card. Table 2 shows that a complete file system can be implemented in less than 3 KB on a smart card using Camille. The loading process is very slow but it is only performed once and does not influence runtime performances in any way. Our implementation in Camille of this embedded file system is 20 times more efficient when accessing a byte in EEPROM than a similar implementation in Java Card, thanks to the optimizations performed by the embedded compiler. Our prototype is roughly as fast as a file system written directly in C, while ensuring the correctness of the loaded code which is obviously not the case with the C implementation.

	Average	Deviation
Size of native code generated	2934 bytes	-
Size of typing information	1539 bytes	-
Loading time	4.194 s	-
Access time to a byte of a file of size less than 2kb	25.9 μs	6.4 μs
Access time to a byte of a file of size greater than 2kb	52.3 μs	30.4 μs
Access time to a byte of a file on a Java Card	587 μs	5,7 μs

Table 2: Measures related to ISO7816-4 file system.

```
short retroaction(short state,
                  short f,
                  char L) {
    short bit_retroaction;
    char i;
    short x;

    bit_retroaction = 0;
    x = state & f;
    for (i = 0; i < L; i ++) {
        bit_retroaction += (x >> i) & 1;
    }
    bit_retroaction &= 1;
    state = (state >> 1) ^ (bit_retroaction << (L - 1));
    return state;
}
```

Figure 2: A random bit generator.

Finally, we implemented a random bit generator to illustrate the performances of generated code for low-level abstractions. This generator is based on the Linear Feedback Shift Registers algorithm [7], which is commonly used by cryptographic protocols as RSA or DES for instance. It is a typical example of an abstraction which needs to be very efficient since it is used by lots of other services in a smart card operating system. We present the implementation of the generation function in Figure 2.

We implemented the `retroaction` function in Camille, in Java Card, and directly in C. We then monitored the performances of the three implementations when generating 1536 random bits. Table 3 shows that the optimized[4] Façade version is 85 times faster than the Java Card one. The Camille implementation is also a little faster than the C implementation, which is due to the fact that the smart card C compiler does not carry out as many back-end optimizations as our code generator does.

Type of code	Time in *ms* for generating 1536 bits
Java Card	26250
Unoptimized Façade code	359
Optimized Façade code	310
C code	331

Table 3: Performance of the LFSR random bit generator.

5 Conclusion and future work

In this paper, we have presented the flexible binding scheme implemented in Camille, a type-safe component-based exokernel for smart cards. This scheme is based on an embedded compilation mechanism well suited for resource-limited devices thanks to its low footprint and runtime

[4]These optimizations consist in exploiting the life cycle of variables

performances. The code generation process is generic and can be instantiated to produce either efficient native code or compact bytecode. On-the-fly optimizations can also be performed during generation of the binary code. The embedded compilation scheme is extensible, since it supports overloading of compilation methods and delegation of binding implementation to superclasses. Bindings can also be composed to implement complex operations. Access control can be enforced when loading a new extension by adding verifications when compiling bindings. Checks can be either static by using complex conditions or static analysis, or dynamic by implementing verification stubs or interceptors. Our experiments have shown that this approach can be very efficient when using an optimized native code generator and is well-suited for constrained devices such as smart cards.

We now plan to work on adding a tool to extract selected properties from generated code. For instance, we plan to extend the embedded compilation mechanism so that it provides support for computing the Worst Case Execution Time of real-time tasks and helps evaluating the energy consumption of embedded code, which can be useful for devices functioning with short-lived batteries. We also plan to implement a code annotation tool to help the register allocation algorithm. Finally, we are working on extending the compilation scheme to support more generic properties using a code weaving mechanism inspired by the aspect-oriented paradigm.

References

[1] D. Deville, A. Galland, G. Grimaud, and S. Jean. Smart Card operating systems: Past, Present and Future. In *Proceedings of the 5th NORDU/USENIX Conference*, February 2003.

[2] D. R. Engler. *The Exokernel Operating System Architecture.* PhD thesis, Massachusetts Institute of Technology, October 1998.

[3] G. Grimaud, J.-L. Lanet, and J.-J. Vandewalle. FAÇADE: A Typed Intermediate Language Dedicated to Smart Cards. In *Software Engineering — ESEC/FSE*, number 1687, pages 476–493. Springer-Verlag, 1999.

[4] G. C. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[5] A. Requet, L. Casset, and G. Grimaud. Application of the B Formal Method to the Proof of a Type Verification Algorithm. *5th IEEE International Symposium on High Assurance Systems Engineering*, November 2000.

[6] A. Senart, O. Charra, and J.-B. Stefani. Developing dynamically reconfigurable operating system kernels with the think component architecture. In *Proceedings of the workshop on Engineering Context-aware Object-Oriented Systems and Environments*, November 2002.

[7] R.A. Rueppel. *Analysis and Design of stream ciphers.* Springer-Verlag, 1986.