A Distributed WCET Computation Scheme for Smart Card Operating Systems

Nadia Bel Hadj Aissa, Christophe Rippert, Damien Deville, Gilles Grimaud

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

{Nadia.Bel-Hadj-Aissa,Christophe.Rippert,Damien.Deville,Gilles.Grimaud}@lifl.fr

Abstract

Computing WCET in a resource-constrained device such as a smart card in a safe manner raises some difficulties. Indeed, most of the classical algorithms for computing WCET do not address resource-limitation or security issues. In this article, we propose to distribute the computation process between the off-card part running on a powerful workstation and the on-card part specific to the hardware included in the smart card. We also guarantee the safety of our computation process by inserting assertions in the generated code and preventing information leaks from the card to the outside.

1 Introduction

Smart card operating systems have to face very hard constraints in terms of available memory space and computing power. Nonetheless, the specifications of most smart card platforms impose strict deadlines for communications between the card and the terminal to which it is connected. This advocates the real time paradigm to guarantee response times and thus introduces the need for computation of WCET on these very constrained devices. Besides, smart card operating systems have very strict security requirements which must be taken into account by all parts of the operating system, including the WCET computation algorithm. Unfortunately, most of the classical algorithms for computing WCET do not address resource-limitation or security issues. We propose in this paper a novel scheme for safely computing WCET on a very constrained device such as a smart card.

*This work is partially supported by grants from the CPER Nord-Pas-de-Calais TACT LOMC C21, the FP6 Integrated Project INSPIRED, the French Ministry of Education and Research (ACI Sécurité Informatique SPOPS), and Gemplus Research Labs.

We first present the CAMILLE operating system for smart cards, and then describe the main issues when computing WCET on very constrained devices. We then detail the architecture we propose to compute WCET in the CAMILLE operating system and illustrate it on an example of a simple embedded algorithm. We conclude by presenting the future work we plan to conduct.

2 The CAMILLE architecture

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 manipulated 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.

System components and applications can be written in a variety of languages (including Java, C, etc). The source code is translated in a dedicated intermediate language called FAÇADE [3] by appropriate tools. Using an intermediate language enhances the portability of the various components is a way similar to Java bytecode. To guarantee the efficiency of the system and the applications, the FAÇADE code is translated into native code using an embedded compiler. This compiler converts FAÇADE programs when they are loaded in the device, and performs machine-dependent optimizations to exploit fully the underlying hardware. FAÇADE is an object-oriented language including only five instructions: `jump`, `jumpif`, `jumplist`, `return`, and `invoke` which can be easily type-checked due to its simplicity.

Thus, CAMILLE architecture is divided in two parts. The off-card part is in charge on compiling the application or system components into FAÇADE and compute the proof

of their type-correctness which is included in the generated binary [4]. The on-card part loads this binary, checks the proof, and then translate the FAÇADE program into native code using the embedded compiler. Thus, CAMILLE takes advantage of the computing power and memory space available on the workstation on which runs the off-card part to perform costly operations. The WCET computation scheme we propose is based on a similar distribution between off-card and on-card parts, as detailed below.

3 Distributing the WCET Computation process

WCET can be computed either statically or dynamically. However, the WCET computed by a dynamic analysis can be less than the real execution time of the code [5], which is not compatible with hard real time constraint. Thus, we focus on a static analysis which is more suitable in our context. Since static analysis usually result in a pessimistic estimate [6, 7], one of our goal shall be to reduce the degree of pessimism as much as possible. Classical techniques for computing WCET include the tree-based [6], the path-based and the Implicit Path Enumeration Technique [8] algorithms.

At the source code level, the tree-based method uses a combination of an abstract syntax tree with a timing schema approach [9]. It works on the source code of the program to extract both its logical structure and the annotations introduced by the programmer. In CAMILLE, the on-card part of the system does not have access to the source code of a dynamically loaded extension, but only to FAÇADE binary code and its proof. As FAÇADE is a low-level language close to assembly, no high-level instructions like `loop` or `if-then-else` are included in the FAÇADE instruction set. Thus, FAÇADE code does not include any way to guess the high-level structure of the program, which means that the tree-based technique cannot be used in our context.

The IPET algorithm generates a set of constraints from the Control Flow Graph of the program. The WCET estimate is then generated by maximizing the sum of the products of the execution counts and execution times of the basic blocks forming the CFG. Constraint solving or Integer Linear Programming can be used to solve this maximization problem. Obviously, a simplex algorithm for instance is much to costly in terms of memory and CPU resources to be executed on a smart card. Since the costs of the basic blocks are unknown off-card, the whole algorithm must be executed in the smart card, which is not realistic for complex programs.

The path-based technique can be assimilated to the classical problem of finding the longest path in a graph, which can be solved for instance by a Dijkstra algorithm [10]. The path-based analysis searches the most costly path in the CFG. Considering the memory space necessary to compute WCET for complex programs with many possible paths, and the heavy computations it implies, it is not possible to use this technique as is in a smart card. Thus, we propose to distribute the computation of the WCET between the off-card and the on-card parts of CAMILLE, so that the most costly operations are done off-card. The path search algorithm cannot be applied off-card as it requires the knowledge of the cost of each path to select the most costly one. Indeed, only the card knows about the worst case timing behavior because it depends closely on the target architecture. Moreover, the off-card part cannot quantify the execution time of each FAÇADE instruction which are handled differently by the on-card backend according to the compilation context (*i.e.* optimizations). Exporting relevant information from the card would make it possible to compute WCET in the off-card part. In fact, exporting a profile containing the exact code generated by the embedded compiler and the cycle number corresponding to each native instruction would allow the computation to be finalized off-card. Unfortunately, carrying out such sensitive information from a secure area as the smart card to the outside is reproved by smart card manufacturers in order to prevent both technology leaks and potential timing attacks [11] against the cryptographic protocols implemented in the card for instance.

In the next section, we show how we propose to distribute the WCET computation process by simplifying the CFG outside of the card before sending it to the on-card part of the system.

4 Implementation in CAMILLE

The WCET computational process has to be split up into two phases. In a first step, in the off-card part of CAMILLE, a weighted control-flow graph must be figured out as shown in Figure 1. Each node in the graph represents a basic block (*i.e.* a sequential piece of code without any jumps or labels: labels start a block, and jumps end a block). Iterations are represented by edges labelled with the upper bound of the loop.

Then, a parser flattens the control-flow graph obtained into a tree. This eases the computation of the WCET by the on-card part of the system, since searching the most costly path is less resource-demanding in a tree than in a cyclic graph. Conditional statements are represented by separate branches in the tree. Loops are replaced by a tag on the node representing the execution count of the block. In the case of nested loops, the inner loop is tagged by the product of its execution count and the outer loop one, as illustrated in Figure 1, where BB_4 will be executed $n_4 \times n_5$ times.

Once the tagged-tree is built, it is sent to the card within the binary containing the FAÇADE code and the proof. The embedded compiler is responsible for searching the most

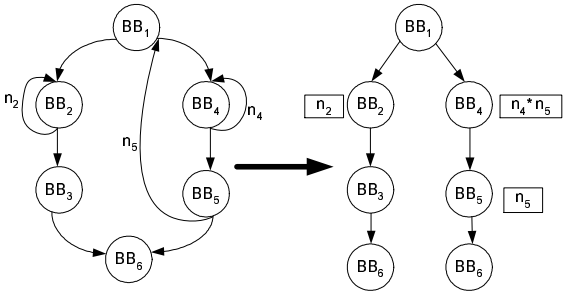

Figure 1. Tranformation of the CFG to tagged-tree.

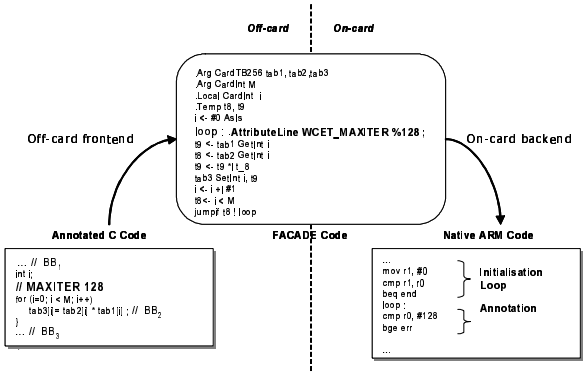

Figure 2. An example of annotation translation.

costly branch in the tagged-tree. As it decodes the FAÇADE instructions one by one, the embedded compiler has to translate the basic blocks of FAÇADE code into native ones.

Once the translation phase is finished, each basic block is assigned an execution time which corresponds to the sum of the number of cycles that will be consumed by each instruction. Then, the tagged-tree is used to compute the WCET of the program by starting with the root node and summing up the execution times of each basic block belonging to the same branch. The formula used to compute the WCET of the right branch is therefore:

$$WCET(Right\ branch) = WCET(BB_1) + n_4 \times n_5 \times WCET(BB_4) + n_4 \times WCET(BB_5) + WCET(BB_6)$$

The WCET of the respective branches are then compared and the global WCET value is sorted out. If the deadline of the program can be met, the code can be executed, otherwise an error message is sent to the off-card part.

To compute the n_i used in the formula presented above, we use annotations inserted in the source code either by the programmer or by code analysis tools. Figure 2 illustrates the CAMILLE compilation scheme of an annotated C code. The off-card part should be extended with a static flow analysis tool capable of translating the assertions inserted by the programmer in the C code to FAÇADE annotations. Figure 2 shows an example of such a programmer-inserted annotation, represented by the C comment `// MAXITER 128` which declares that the following multiplication loop will iterate 128 times. This C comment is simply translated by the static flow analysis tool into the FAÇADE annotation `.AttributeLine WCET_MAXITER %128;`.

While decoding FAÇADE instructions, if it reaches an annotation, the embedded compiler needs to verify it. For instance, if the off-card part claims that a loop will not iterate more than 128 times, the embedded compiler has to explicitly insert code to exit the loop when the loop has iterated 128 times.

The assertion `cmp r0, #128; bge err` compares the register `r0` which stores the value of variable `M` with the declared number of iterations (128). If `r0` is greater than 128, the program exits the loop and branches to an error label.

5 Conclusion and future work

We presented in this paper the scheme we propose to safely compute WCET in a resource-constrained operating system. By distributing the computation between the off-card part running on a powerful workstation and the on-card part specific to the hardware included in the smart card, we are able to circumvent the very strict memory and CPU limitation of the device. We guarantee the safety of our scheme by inserting assertions in the generated code to validate the annotations sent by the off-card part, and by preventing information leaks from the card to the outside. Finally, we show that our scheme can be easily implemented in a secure smart card operating system as CAMILLE. We are now working on an extended architecture which would permit to safely export hardware information outside of the card. This would allow using the IPET technique to compute the WCET off-card without risking to comprise the security of the embedded system.

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, 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] J. Engblom, A. Ermedahl, M. Sjdin, J. Gustafsson, and H. Hansson. Worst-Case Execution-Time Analysis for Embedded Real-Time Systems. *Software Tools for Technology Transfer, special issue on ASTEC*, 2001.

[6] A. Colin and I. Puaut. A Modular and Retargetable Framework for Tree-Based WCET Analysis. In *13th Euromicro Conference on Real-Time Systems*, June 2001.

[7] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Real-Time Systems*, 2:249–274, 2000.

[8] Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software using Implicit Path Enumeration. In *Workshop on Languages, Compilers & Tools for Real-Time Systems*, pages 88–98, 1995.

[9] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Journal of Real-Time Systems*, 1(2):159–176, September 1989.

[10] E.W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[11] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO '96*, volume 1109 of *LNCS*, pages 104–113. SV, 1996.