SAFE COLLABORATION IN EXTENSIBLE OPERATING SYSTEMS:

A STUDY ON REAL TIME EXTENSIONS

D. Deville[◇†] Y. Hodique[◇‡] I. Simplot-Ryl[◇]

◇ LIFL*, Univ. Lille 1, UMR CNRS 8022
† INRIA Futurs, POPS research group ‡ ENS Cachan

59655 Villeneuve d'Ascq Cedex, France

{deville,hodique,ryl}@lifl.fr

keywords: exokernel, safe extensibility, embedded systems, static analysis

Abstract: This article proposes a solution to guarantee safe interaction to components that are willing to collaborate in an extensible operating system, dedicated to small embedded systems such as smart cards, that guarantees isolation. We propose a simple way to verify the behaviour of some components using an extension of the type system by adding information about argument passing modes to the method signatures (*e.g.* is an argument read or written). We present a formalization of a PCC-like algorithm (off-card proof generator and on-card proof verifier) to statically check the passing modes of the components in the Camille exokernel for smart cards. We apply our technique to ensure trust between collaborative real time extensions with the aim of supporting *safe* dynamic loading of scheduling policy.

Introduction

The interest for embedded systems has grown a lot with the development of electronic devices. Some of them are designed for small embedded objects like smart cards, RFID tags or personal digital assistants, whose common characteristic is their very restricted resources. Their operating systems need to be

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

light weight but also adaptable to face the evolution of their environments. Recent trends [1] in embedded operating systems advocate component architecture and dynamic loading of system extensions to favour extensibility. However, this should not result in lowering the security in critical objects like smart cards. We present issues concerning safe use of untrusted components in the context of dynamic verification of components in CAMILLE [1], an exokernel [2] for smart cards. We first give a short presentation of the system and its basic security properties. We argue, in Section 2, that to go further, for example when extensions try to collaborate, we also need to verify some functional properties of system components. We propose to test the components on all possible inputs when it is possible and when we may have a complete control of the internal state of the used functions, like a scheduling function for example. In Section 3, we present an algorithm to determine the access mode of a function to its arguments and to the rest of the system. This algorithm is divided into two parts: an off-card one to compute the access modes and an embedded one to verify the proof when loading new components, similarly to Proof-Carrying Code (PCC) [3]. Then, we give in Section 4 some elements about the soundness of the algorithm. Finally, we present in Section 5 some experimental results that demonstrate that our mechanism is light enough to be embedded in smart cards.

1. The CAMILLE operating system

CAMILLE is an extensible operating system [1] designed for resource-limited devices, such as smart cards. It is based on an exokernel architecture [2] which advocates the principle of not imposing any hardware abstraction in the kernel, which is only in charge of demultiplexing resources. CAMILLE provides secure access to the various hardware and software resources used 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. The exokernel architecture is well suited for resource-limited devices such as smart cards. Starting from a minimal kernel and adding only needed system abstractions makes it possible to build small systems that include only the services that applications will actually use. Moreover, the flexibility of the exokernel architecture allows to dynamically add new applications or services that were omitted when the system was built, thus supporting post-issuance of applications (*i.e.* ability to dynamically load new applications once the

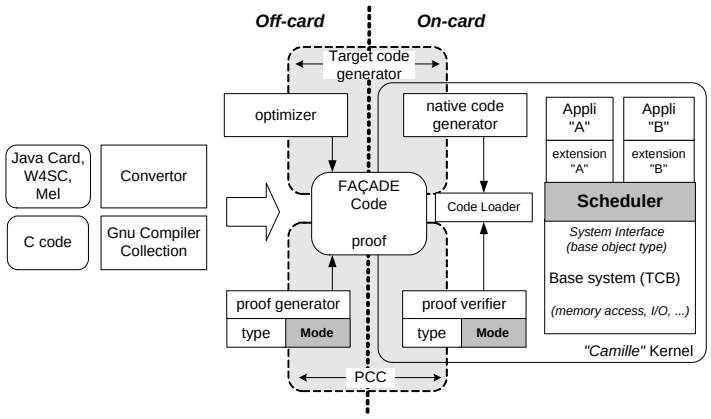

Figure 1: CAMILLE architecture and software infrastructure.

device is in the user's pocket). Thereby, abstractions can be programmed to best fit the needs of the applications and fully exploit the available resources.

Figure 1 describes the distribution of CAMILLE components between the terminal and the card: system components and applications can be written in a variety of languages (for instance Java and C), then the source code is translated into a dedicated intermediate language called FAÇADE [4] by suitable tools (*e.g.* a Java to FAÇADE converter or the GNU Compiler Collection for C code). Using an intermediate language enhances the portability of the components (as the use of Java bytecode). Portability is also enhanced by limiting the number of machine-dependent system primitives and ensuring that they can easily be adapted from one platform to another. These primitives are grouped in a component called the "Base system". To guarantee efficiency of the system and the applications, the FAÇADE code is translated into native code using embedded on-the-fly compilation techniques.

FAÇADE is a very simple fully object-oriented language, which can easily be type-checked. It is composed of five instructions (`return`, `jump`, `jumpif`, `jumplist` and `invoke`) and uses virtual registers as working variables. To guarantee integrity of embedded code, a proof of the type-correctness of each program is computed by a proof generator and included in it when it is loaded on the card. This "Proof-Carrying" code [3] is verified [5] on the embedded system when loading a new application. Once the type correctness of the application is established, the FAÇADE code of its methods is translated into native code. This ensures that no program loaded by the card will com-

promise the integrity of the system. Thanks to all these features, CAMILLE provides: *Portability*, *Security* and *Extensibility*.

2. Problematics

The FAÇADE intermediate language combined with the PCC type-checker guarantees isolation of operating system extensions. The problem of trust between extensions arises when they try to collaborate (*e.g.* to share a resource, or to use a component loaded by another one). Operating system component programmers need to provide ways of verifying bindings between components. In general-purpose operating systems and execution platforms, verifying components is known to be very complex. The resources needed for the verification dramatically increase with the range of possible inputs and with the size of components. Such an analysis is usually performed using high-level languages like JAVA or domain-specific languages and tools, like for example in BOSSA [6], a framework used for the development of safe real time schedulers. We will focus on an intermediate language: FAÇADE. CAMILLE components, thanks to the exokernel approach, are fully tailored to a particular service and their possible inputs are usually bounded. Thus, we propose a method to check the functionality of some components, based on a static analysis of code and an exhaustive test.

The use of this technique will be illustrated on dynamic loading of real time extensions in CAMILLE (in smart card, lots of periodic treatments with deadline exist, like for example the generation of cryptographic session keys in SIM Card to maintain the communication with the GSM infrastructure). As advocated in the exokernel literature [2, chapter 3], the basic time allocation in CAMILLE is fair: CAMILLE guarantees each extension to have an equal access to the microprocessor. To ensure success of the CPU allocation even for very unbalanced task distributions, we advocate the need for extensions to sometimes collaborate and share their access to the microprocessor. Each extension willing to collaborate must propose a scheduling policy (called *plan*) working for its own set of periodical tasks: This function is in charge of electing the task, that will run for the next time slice. All collaborative extensions are grouped under the control of a supervisor and a global *plan* is elected amongst all the functions to schedule their tasks.

The problem is now to elect a function: some extensions will have to trust a *plan* function from another extension so they are no more isolated.

To elect a function, we propose a simple algorithm that consists in testing all the functions and electing the first that succeeds in scheduling the tasks (no task misses its deadline).

The extensions have to trust the *plan* function, therefore the test must be reliable: A malicious function could cheat and favour its extension, thus causing famine for other extensions. A set of periodic tasks is a periodic system whose period (called the hyper-period) is equal to the least common multiple of the periods of all the tasks. If the *plan* function is a pure deterministic function, it is sufficient to exhaustively test it on the hyper-period to prove its correctness. However, optimized *plan* functions need access to memory areas, thus are not side-effect free, and may have internal states. A similar problem was exposed by Engler in [2, chapter 4], where he uses a function `owns()` to secure the access to meta data for disks management. This function is supposed to be an *Untrusted Deterministic Function*, that is, a deterministic function whose state is visible to a verifier. It is trivial to show the deterministic property in the case of pure functions but more complex in the case of more usual side-effect functions.

As we need functions with side effects, we propose to use functions with the following conditions: We know whether the function performs an action on parameters (*e.g.* read, write, link to another parameter), and also whether it performs a dangerous action on the rest of the system (*e.g.* read a non-deterministic data like the date or modify the global state).

The previous properties are verified at loading time. Thus, a function may use some parameters to store an internal state, but we keep control on them. A function using non-deterministic system information or a global state is rejected. The exhaustive test is performed every time a new collaborative extension is loaded in the system. As soon as a plan function passes this test, the exokernel installs it and starts using it. The schedule trace obtained at testing time is not stored in memory for future use, because memory is a scare resource of smart cards. The new plan function is installed right after the end of the current round to ensure no task will miss its deadline. After each hyper-period the memory zones used by the function are reseted. Thus, as a malicious plan function is not able to detect in which phase it is called, it cannot cheat.

3. Algorithm

We present in this section the algorithm used to compute the access modes of a method. Because of CAMILLE's design, any computation is done by a method call and an object cannot access another one if it was not given a reference to the latter (through an argument in a method call for example). So, to perform the analysis, we focus on computing if some arguments are either used in such a way that they can give a malicious method an internal state, or remembered (as a reference) for future use. This is somewhat an extension of the embedded Escape Analysis [7] since we focus on the persistence of information through method calls, or to the side-effect analysis [8] for the internal state part. Since FAÇADE is an object-oriented language, every method invocation involves, in addition to its explicit arguments, a special argument named `this` that represents the object that receives the call. Some `static` methods modify the global state (as in `new` invocation), so we introduce a second special argument named `world` that represents the rest of the system (*i.e.* anything outside the scope of the explicit arguments). Other methods of course have access to the global state through and only through these `static` methods. In the rest of this paper, "arguments" refers to this extension of the primary arguments vector, and a method might be considered as a mapping over the extended arguments. There are several alternatives for arguments use:

Definition 1 (Passing Modes)**.** *Let $E = \{R\} \cup A$, where $A \subset \mathbb{N}$ is a finite set (its cardinal is the maximal number of arguments for a function), and R represents the return value. One element of the following lattice is associated with each argument :*

- $\bot$*: default state, the argument is not used in any dangerous way,*
- $\top$*: the argument was used to obtain an internal state (happens when writing to a field of an object for example),*
- $Link_\alpha$ *for $\alpha \subseteq E$: a reference on the argument has been kept by calling the method (for example, if it has been attached to the return value, then $R \in \alpha$). Any further modification of an element of α modifies the initial argument. The partial order here is given by $\subseteq$*

In the rest of the paper, $\mathcal{M}$ will denote the set of elements of this lattice.

```
method::Sign() { // sign is the signature to be computed
  stack.push(entry_point);
  while(! stack.empty()) {
    line = stack.pop();
    switch(instructions[line].type()) {
      case Jump(target):
        // << is a injection operator: any dependency at line is copied at target.
        if(deps[target] << deps[line]) stack.pushIfNotIn(target);
      case JumpIf(target):
        if(deps[target] << deps[line]) stack.pushIfNotIn(target);
        if(deps[line+1] << deps[line]) stack.pushIfNotIn(line+1);
      case JumpList(targets):
        foreach i in targets
          if(deps[i] << deps[line]) stack.pushIfNotIn(i);
      case Invoke(m):
        if(deps[line+1] << m.applyOn(deps[line])) stack.pushIfNotIn(line+1);
      case Return(ret): // args is the arguments' array
        foreach i in args do { // |= is an accumulative bitwise "or"
          if(deps[line][i].contains(ret)) sign[i] |= IO_R;
          foreach j in args do
            if(deps[line][i].contains(args[j])) sign[i] |= IO_j;
        }}}}
main() { // cat is the set of methods to be considered, the "catalog"
  until(! cat.modified()) {
    foreach m in cat do  m.signature = m.Sign();
  }}
```

Figure 2: Signature computation algorithm.

To do the verification we propose an algorithm that performs an abstract interpretation [9] of the code to affect a "signature" to the method in terms of passing modes of arguments: the signature of a method with n arguments is an element of $\mathcal{M}^n$. The goal is to perform a complete computation of the behaviour of a method regarding its arguments, based on the main idea that the signature of a method depends on the signatures of the methods it calls.

The algorithm we present does not apply for the "native" (machine-dependent, not expressed in FAÇADE) methods. Therefore these methods have to be treated by hand and given a signature according to their semantics. Note that this kind of methods only appears in the CAMILLE kernel: there are currently about 120, covering essentially arithmetical operations and memory management. These methods form the first firm "base of trust" for signatures. Once a signature has been computed, it is added to the "base of trust". The analysis must deal with invocations of methods it has not yet considered and with recursive calls. Our policy is to associate a default signature to such a method, where every argument's mode is set to $\perp$. This may lead us to re-compute the signature of the methods for which a false assumption was made.

```
bool verify(method,sign) {
  bool res = true; dependencies_vector dep = initDeps();
  foreach instr in method.body() do {
    if(instr.hasLabel()) { // apply() computes the action of instr over dep
      // includes() check if computed deps are compatible with proof informations.
      res |= (instr.proof.includes(dep.apply(instr)));
      dep = instr.proof;}
    else
      dep = dep.apply(instr);}
  return res;}
```

Figure 3: Verification algorithm.

The complete algorithm (given in pseudo-C++ in Figure 2) works on successive computations for every method in a catalog (function `main`). It computes the signatures of the methods belonging to the catalog, starting in a state where all the passing modes are $\perp$ and making it evolve. All the methods called by methods of the catalog must belong to the catalog or to the "base of trust". The computation of the signature of a method is performed by `Sign`. It is assumed that any invoked method has a known signature, so that whenever this assumption is proved to be incorrect, the algorithm (function `main`) has to perform a new signature computation, at least for the methods that were depending on this one. Therefore we reach a fix-point when no assumption is negated anymore. A second fix-point has to be reached for each method signature to ensure that the computation gives the right dependencies for the method, according to the current hypothesis. Therefore, the `Sign()` function keeps looping until there are no more possible change in `deps`. We maintain a global description (`sign`) of the mode of each argument and an array of dependencies (`deps`) between variables and arguments per instruction: we say that *a depends on b* if a modification of b may lead to a modification of a. For a line, this array is a function of its predecessors and the instruction: `Invoke` leads to some alterations to `sign` and `deps` according to the signature of the method (`applyOn` is invoked to perform the desired changes on `sign` and `deps`). The resulting signature is the unification of the ones found at each exit point (`Return` instruction).

Since we consider very small embedded systems, we need to be careful about the computing power and memory needed for any action. The most important thing is to make sure that any on-card computation remains linear with respect to the code size. In order to achieve this, we apply a method used in classical Proof-Carrying Code [3], which consists of keeping the computed

information (dependencies) only at label points, as a proof. Doing this, we keep memory usage at a low level, since typical methods don't have too many labels, and give the card the possibility to verify the correctness of the signature. To summarize, we give the card three elements: the code that has to be loaded, a signature describing its behavior, and proof elements to enable the system to verify a method before it can be used. Note that one can write a proof that will mark some parameters as greater than they really are (with respect to the order relation described by the lattice), but what is really important is to make sure that a mode is not marked as too low.

Due to the object-oriented nature of the FAÇADE language, we have to deal with the problems of inheritance, and method overloading. Since we never know what is the actual method called by an `Invoke` instruction (it can be any method from a subclass), a static check is not sufficient. We have to ensure at loading time that signatures of methods that overload other ones are compatible. In a first approach, we enforce signatures to be the same, other solutions will be discussed in Section 5.

4. Outline of the proof

In this section we give an idea of how to prove the correctness of the previously presented algorithms.

First of all, for every instruction and every argument of a given method, the total number of dependencies is bounded (by the number of variables) and can only grow from one iteration to another since a computed dependency is never negated : once a variable is said to contain a particular reference, it remains true forever. The treatment of a method stops as soon as there are no more changes in the dependencies, and is deterministic, that is, `Sign` terminates. For every method and every argument, the associated passing mode can only grow and is bounded by $\top$ (see Definition 1), so:

Lemma 1. *The algorithm terminates.*

Suppose now that the algorithm stops after T iterations of the `Sign` function, and let (s_n) be the sequence of signatures computed by those steps. Let p be the number of functions in the dictionary. The stopping condition implies that for each i such that $T - p < i \leq T$, we have $s_i = s_{i-p}$ (s_i and s_{i-p} refer to the same method). The algorithm is deterministic and $s_i = \texttt{Sign}(\{s_j\}_{j \in \{i-p \ldots i-1\}})$. Therefore for each i such that

$T \leq i, s_i = \texttt{Sign}(\{s_j\}_{j\in\{i-p\ldots i-1\}}) = \texttt{Sign}(\{s_j\}_{j\in\{i-2p\ldots i-p-1\}}) = s_{i-p}$. Thus, we have:

Lemma 2. *The stopping condition for the algorithm is a sufficient condition to produce a correct signature (one more iteration would not change anything).*

Let us now consider the `Sign` method. Let $|code|$ be the number of instructions of the method's code, $|args|$ the number of its arguments, and $|locals|$ the number of local and temporary variables. Let's consider the set $\mathcal{E} = \{(l_i, D_j)\}_{i\leq|code|, j\leq|args|*(|args|+|locals|)}$ where l_i is a program line (instruction), and $D_j : args \rightarrow args \cup locals$ a dependency function. The set $\mathcal{E}$ is the Cartesian product of the space of lines and the space of possible dependency relations.

We consider the graph $\mathcal{G}$ over $\mathcal{E}$ that is induced by the relation $(l_1, D_1) \rightarrow (l_2, D_2)$ such that l_2 is a successor of l_1 and the application of line l_1 over the dependencies D_1 leads to dependencies D_2 before applying line l_2 (see the injection operator `<<` in Figure 2). The graph $\mathcal{G}$ also contains nodes that do not correspond to a valid state (*e.g.* the node $(0, D_\emptyset)$, where $D_\emptyset$ is such that for each x of $args$, $D_\emptyset(args) = \emptyset$). The set of nodes that can be accessed during the execution (that is, the subsets of dependencies for each variable and line of code) matches with the graph $\mathcal{G}' \subseteq \mathcal{G}$ that has (l_{in}, Id) as root node (l_{in} represents the entry point for the method, Id the *identity* function). For any $k \leq |code|$ and x of $args$ the dependencies for x at line l_k are given by $\bigcup_{(l_k,V)\in\mathcal{G}'} V(x)$ (we need to consider every valid path to l_k). Thus, we have:

Lemma 3. *`Sign` provides the complete set of dependencies for a method.*

Since the algorithm computes the correct dependencies, any modification of a variable is seen as a modification for any argument that is *linked* to that variable:

Proposition 1. *For a function f that only invokes methods that have a known signature, one computation of `Sign` provides the correct signature.*

To conclude with the signature algorithm, we consider the following property:

Proposition 2. *If a method's signature is incorrect, then the algorithm keeps looping (i.e. a correct solution is found as soon as the algorithm stops).*

We prove Proposition 2 by contradiction. Suppose a signature remains incorrect after the algorithm has stopped. For this method there is an instruction with incorrect dependencies, so that there is a method called by the first one that has an incorrect signature. One can suppose this method is different from the one we are analyzing. By recurrence, this leads to the fact that every signature is incorrect, which is obviously false.

We still have to make sure that the embedded verification algorithm computes the same signature as the off-card algorithm. The only difference is at the label points. For simplification, one can assume that every predecessor of a label is a jump instruction (if it is not the case, let's add a `jump <next_line>` before it). Let $(j_1, \ldots, j_n)$ and $(l_1, \ldots, l_n)$ be the vectors of dependencies for jump and label respectively. If the data provided by the embedded proof are correct, then for each i in $\{1 \ldots n\}$, $j_i \subseteq l_i$. In turn, if for each i in $\{1 \ldots n\}$, $j_i \subseteq l_i$ for every jump, the vector of dependencies associated with a label may be too big, but that does not matter since we are willing to authorize pretending a behavior is dangerous even if it is not. Then, we can establish the correctness of the verification:

Property 1 (Correctness)**.** *The result given by the verification algorithm is weaker (which means that a trivial behavior can be tagged and verified as dangerous) than the result one would obtain by computing everything with the signature algorithm, but it is correct.*

5. Experimental Results

The code loading process of CAMILLE was modified in order to verify the access modes of functions as illustrated in Figure 1. Previously described algorithms were evaluated on the source of the CAMILLE exokernel. This case study represents 20K lines of C code which are translated into 5K lines of FAÇADE and support all components and services described earlier. Figure 4 summarizes some information extracted from the source.

The only "*Link*" value that we encountered over the whole CAMILLE kernel was the $Link_{\{R\}}$ one, which makes us propose an approximation of the model. We want to keep only a 3-valuated flag ($\bot$, $\top$ or $Link_{\{R\}}$), and map any other value to the top of the lattice. Doing this, we may consider some methods as more intrusive than they are, but we save a lot of memory. Without this simplification, we need to consider $2 + 2^{19}$ states for the flag (16 explicit arguments, 2 implicit and R), so that a method signature

	number of Instructions	number of Labels	number of Args	number of Locals	number of Temporaries
Total (for kernel)	3114	393	396	240	289
Average (per method)	17	2	2	1	1
Worst (per method)	167	22	9	11	5

Figure 4: Kernel statistics.

takes 342 bits to be encoded, whereas it is reduced to only 29 bits with our simplification.

The core algorithm currently needs 4 iterations to compute the signature of the 320 methods given by the kernel (120 of them are "Base system" methods that need to be tagged by hand), making it hard to embed the complete computation (instead of the single proof checker). The FAÇADE language itself imposes some limits. The most complex function one can build using FAÇADE language is limited to 256 labels, would use 512 working variables. Thus, in theory the size of the embedded proof (even with the previously exposed simplifications) could reach about 300 kilobytes for a method in the worst case. In practice, this limit is never reached. Statistics presented in Figure 4 lead to a proof of 200 bits for the most complex method of the kernel. This is small enough to be embedded and verified by the card.

The last thing we have to deal with is inheritance and overloading. In order to statically check the method signatures, we have seen that a method that overloads another one must have the same signature. We may first extend this definition by asking for a lower signature (*i.e.* more restrictive) as in the following definition.

Definition 2 (Signature order)**.** *Let f be a function with a signature $s = (m_1, \ldots, m_n)_{m_i \in \mathcal{M}}$. Let $s[i]$ denote m_i. A partial order over the set of signatures can be defined as following:*

$$s_1 \leq s_2 \iff |s_1| = |s_2| \wedge (\forall i \in \{1, \ldots, n\}, s_1[i] \leq s_2[i]).$$

This ensures that the newly loaded method respects the constraints imposed by the already-loaded ones and is more permissive: the overloading method must respect the signature of the overloaded one but not the contrary.

Note that even this definition can be rather limiting since we often want to have "empty" body for the top-level classes (one can see them as simple interfaces) which will be tagged as "$\perp$-methods" and will forbid lots of

implementations. This drawback can be avoided by introducing the ability to over-evaluate the signature of a method, which can be achieved in two ways. First, we could allow a programmer to manually give signatures to some methods. Secondly, we could try to automatically compute the "best signature": This is the solution we have chosen to avoid programmers to deal with our mechanism. The signature algorithm allows us to load a group of classes (the kernel for example), and adapt the top-level signatures (taking into account the lower-level ones) to allow the loading of the complete group. The on card verification mechanism does not have to be changed since it only verifies that the announced signature is correct.

We have implemented various cooperative `plan` functions using classical scheduling algorithms like for example *Round Robin*, *Earliest Deadline First* or *Least Laxity First*. We have implemented a simple *Round Robin* policy. Its C prototype is `int planRR( Task tl[], int nb_task, int curr_slice)` and its signature is found to be $(\bot, \bot, \bot, \bot, \bot)$ (the first two arguments being the `world` and the `this` pseudo arguments). According to this signature, one can deduce that this implementation is a pure function. This policy does not need any run time control as it simply elect each tasks one after the others. The second `plan` function is a standard *EDF* policy. Its C prototype is `int planEDF( Task tl[], int nb_task, int curr_slice, byte cmem[ ])` and its signature is found to be $(\bot, \bot, \bot, \bot, \bot, \top)$. The system authorizes the corresponding extension to read or write to a memory zone through the `cmem` argument. The given policy uses this memory to stock the progression of each task and, doing this, code is more optimized and simpler to write. Thanks to the verification at loading time and also to the control performed by the system after each hyper-period (the `cmem` zone is reseted as it is detected as $\top$), this policy can be considered as a trusted collaborative scheduling policy.

Conclusion

This article has presented a formalization of an access mode analysis. We proposed a formalization of a PCC-like algorithm (off-card proof generator and on-card proof verifier) to statically check this property in the CAMILLE code loader. This algorithm applied on the FAÇADE language is well suited for the heavily constrained devices that are smart cards. It was applied on the problem of trust between collaborative real time extensions in the CAMILLE exok-

ernel to support *safe* dynamic loading of scheduling *plan*. The same technique can be applied on all the system components whose input domain is finite and not too large (*e.g.* Access Control Lists).

The on-the-fly code generator can make use of signature information to perform code optimizations. For example, compilation patterns can be adapted to use variables dependencies or the fact that some variables appear to be constant. We also plan to apply similar strategies to enhance memory management capabilities (like safe memory deallocation) or provide the same kind of properties in Java bytecode as in FAÇADE.

[1] D. Deville, A. Galland, G. Grimaud & S. Jean, Smart Card operating systems: Past, Present and Future, *Proc. 5th NORDU/USENIX Conference*, Västerås, Sweden, February 2003, 14–28.

[2] D. R. Engler, *The Exokernel operating system architecture*, doctoral diss., Massachusetts Institute of Technology (MIT), 1999.

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

[4] G. Grimaud, J.-L. Lanet & J.-J. Vandewalle, FAÇADE: A Typed Intermediate Language Dedicated to Smart Cards, *Proc. Software Engineering–ESEC/FSE*, 1999, 476–493.

[5] A. Requet, L. Casset & G. Grimaud, Application of the B formal method to the proof of a type verification algorithm, *Proc. Fifth IEEE International Symposium on High Assurance Systems Engineering (HASE 2000)*, 2000, 115–124.

[6] J. L. Lawall, G. Muller & L. P. Barreto, Capturing OS expertise in an event type system: the Bossa experience, *Proc. 10th ACM SIGOPS European Workshop 2002 (EW2002)*, France, September 2002, 54–61.

[7] M. Q. Beers, C. H. Stork & M. Franz, Efficiently verifiable escape analysis, *Proc. M. Odersky, editor, ECOOP 2004, LNCS 3086*, Oslo, 2004, Springer-Verlag, 75–95.

[8] J.-P. Talpin & P. Jouvelot, The type and effect discipline, *Proc. Seventh Annual IEEE Symposium on Logic in Computer Science, Santa Cruz, California*, Los Alamitos, California, 1992, IEEE Computer Society Press, 162–173.

[9] P. Cousot & R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, *Proc. 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'77)*, California, 1977, 238–252.