Policy-Driven Separation for Systems-on-a-Chip

Ted Huffmire
ArchLab
huffmire@cs.ucsb.edu

Tim Sherwood
ArchLab
sherwood@cs.ucsb.edu

ABSTRACT
Many embedded applications are implemented on systems-on-a-chip (SoCs) that contain multiple interacting components that may operate at different trust levels and clearance levels. These components typically share resources such as memory. For example, a SoC might contain two CPU cores, a Crypto core, and a DSP core, all sharing memory. A similar example is a multi-core processor in which different cores operate at different levels of security. Our goal is to make sure that cores share resources nicely by developing efficient hardware mechanisms.

Providing separation among these modules is a crucial security primitive. Separation is the enforcing of the legal sharing of a resource such as memory among multiple cores. Suppose Alice works for Company A, and Bob works for Company B. Alice’s core must not be able to obtain the data of Bob’s core and vice-versa.

The easiest way of providing separation is to implement each core on its own physically distributed chip. However, physical separation makes it impossible to realize the cost savings of increased integration. Another option is to implement the separation in software. Separation kernels [1] use software virtualization to provide each application with the same level of isolation it would have with its own physically distributed processor. The disadvantage of this approach is that there is extra overhead associated with providing virtualization. In addition, the design complexity of modern out-of-order microprocessors makes it difficult to implement separation kernels with a verifiable level of trust. Furthermore, since separation kernels are a software-based scheme, they will not work for embedded applications that lack code and a program counter.

Standard memory protection, which relies on a page table to prevent processes from accessing memory outside their assigned virtual address space, is not a workable solution for embedded systems either. In standard memory protection, if a process attempts to access memory outside this range, a page fault occurs, and the process is terminated. Since standard memory protection was never designed with rigorous security as a design requirement, it is inadequate for applications that need to be able to handle multiple levels of classified data. Moreover, many embedded systems do not have a disk, operating system, page table, or TLB. Mondrian memory protection [2] provides memory protection at the granularity of a word rather than a page, but it suffers from the same problem.

An alternative way of providing separation is to use a reference monitor that enforces the legal sharing of memory. It is possible to design a system in hardware in which every memory access must be approved as legal by a reference monitor that enforces a memory access policy that every core on chip must obey. This approach combines the cost savings of increased integration without the overhead of software complexity. Reference monitors are not new; in fact, they belong to the class of enforcement mechanisms known as execution monitors [3]. However, our formulation of reference monitors in the embedded domain is new.

Our approach uses a specialized compiler to translate a memory access policy expressed in a specialized language to a table format that can be represented by a memory tile. This memory tile can be loaded onto a SoC, and the reference monitor enforces the policy specified by this tile. Our language specifies which modules (subjects) have which access rights to which ranges of memory (objects).

Figure 1 shows an example of a simple isolation scenario. Module₁ and Module₂ are in separate compartments. Module₁ can only access (read from or write to) Range₁, and Module₂ can only access Range₂. This scenario would be expressed in our language as the following:

Access → [Module₁, rw, Range₁]![Module₂, rw, Range₂];

Policy → (Access)*;

Our policy compiler, which was made with the help of Lex and Yacc, converts this policy to a regular expression. Next, the compiler converts the regular expression to an NFA, and then it converts the NFA to a minimized DFA. Finally, the DFA is converted to a table format that can be loaded onto a special memory tile located on the chip. Our reference monitor enforces the policy specified by the memory tile.

In addition to isolation, our policy language is powerful enough to express a variety of classic security scenarios, including both stateless (fixed) and stateful (transitional) policies. Our language-based approach provides flexibility because changing a policy is much easier than redesigning by hand a reference monitor expressed in a hardware description language.

Figure 1: An isolation policy.

Systems often need to be designed to be responsive to external events. For example, if the system comes under attack, it makes sense to change from a less restrictive memory sharing policy to a more restrictive policy. A mechanism that can dynamically switch the policy enforced by the reference monitor is needed to achieve this goal. We call this mechanism a configuration manager.
The design of a configuration manager involves many interesting questions. Since area is at a premium, it is impossible to store every possible policy on a chip. We will make the following assumptions: that policy switching is rare, that we always switch to a more restrictive policy, and that the number of policies to switch between is small. All we need to do is have additional memory tiles for the different policies, and a simple hardware mechanism to tell the reference monitor which tile to use. The configuration manager will need to obey a meta-policy that specifies the conditions under which switching can occur and which core has the authority to perform a policy switch. A more sophisticated implementation will use a hot-swappable design in which one tile contains the active policy, and another tile is used for loading a new policy.

The fact that every memory access must go through the reference monitor raises an import issue. Multiple simultaneous memory references will have to be serialized, negatively affecting system performance. It is possible to overcome this problem by implementing several independent, distributed reference monitors that enforce the same policy. In the case of a stateless policy, all that is necessary is to duplicate the reference monitors, but in the case of a stateful policy, the distributed reference monitors will need to communicate with each other so that they are all in the same state.

Our policy language is designed with the restriction that it is only as powerful as a regular expression. This restriction exists because we are translating the policy to a DFA, which is equivalent to a regular expression. There may be some Turing-complete security policies that we cannot express in our language. Is there a way for us to modify our design in order to enforce a larger class of policies?

In any security technology, usability is a key concern. The reference monitor is only as good as the policy it is enforcing. Engineers need to be able to construct precise policies. Unfortunately, our memory access language is a relatively low-level language, and working in this language is prone to error. We must design a higher-level language for expressing concepts such as isolation and controlled sharing, and we must design a compiler to translate this high-level language into our low-level memory access language.

In addition to being usable, good security technology also must be low-cost. Our policy compiler can generate reference monitors that are efficient in terms of area and cycle time, but we must analyze the overall system performance of a realistic embedded application that uses our reference monitor approach. We plan to apply phase classification, which has been successful at analyzing systems in the microprocessor domain, to the embedded domain. Phase analysis classifies the repeating behavior of computer programs, and it enables more accurate simulation. Since phases repeat, it is unnecessary to perform the detailed simulation of a phase when it reoccurs. We plan to leverage this to our advantage in the quantitative analysis of our security methods.

Building such a realistic embedded application will force us to confront several issues related to the architecture of our security methods. For example, we must make sure that modules cannot spoof their identities and therefore confuse the reference monitor. We must also figure out how to make sure that every memory access goes through the reference monitor. We must determine the interface between the cores and the memory controller, and we must figure out how the reference monitor will connect to the memory controller. In addition, circuit elements can have values of 0, 1, X (undefined), and Z (high impedance line). In fact, there are around a dozen different values that have been discovered. However, security works in the binary domain (0 or 1). These different values (X, Z, etc.) could affect the security of any hardware security approach.

REFERENCES

