Lightweight Module Isolation for Sensor Nodes

Nirmal Weerasinghe, Geoff Coulson
Computing Department, Lancaster University, UK
[n.weerasinghe, g.coulson@lancaster.ac.uk]

Abstract *There is an increasing tendency in sensor networks (and related networked embedded systems) to push more complexity and 'intelligence' into end-nodes. This in turn leads to a growing need to support isolation between the software modules in a node. In conventional systems, isolation is achieved using standard memory management hardware; but this is not a cost-effective or energy-efficient solution for small, cheap embedded nodes. We therefore propose a software-based solution that promises isolation in a significantly lighter-weight manner than existing software-based mechanisms. This is achieved by frontloading effort into offline compilation phases and leaving only a small amount of work to be done at load time and run time.*

1. Introduction

As sensor networks (and related networked embedded systems) evolve and grow in prominence there is an increasing tendency to push more complexity and 'intelligence' into end-nodes. There are three main reasons for this. First, it enables more interesting applications that are not possible with simple data aggregation [15]. Second, sensor network architects are increasingly designing around the trade-off between processing and communication overheads (i.e. moving computation closer to data on the grounds that the energy cost of sending a byte over the radio is equivalent to thousands of processor cycles). Third, many sensor networks have inherently long deployment lifetimes, and this implies a need to restructure/update node software on an ongoing basis, and also to accommodate mutually-distrustful modules from multiple sources.

As a consequence of the above trends, it is increasingly necessary to be able to preserve inter-module isolation on nodes in the face of faulty and untrustworthy code. Otherwise, for example, a faulty module could easily corrupt another module, or execute some functionality to which it had no right (e.g. reconfigure an I/O device). In traditional architectures such errors are prevented through hardware support for virtual memory and supervisor/ user modes: each module owns a private address space, and the hardware ensures that modules interact with the rest of the system in a controlled manner (e.g. through a system call interface). However, such hardware support is non-existent on many popular MCUs [7,16] in use in sensor networks. Furthermore, this situation seems unlikely to change in the foreseeable future due to the complexity and cost (in terms of both energy use and chip size) that these hardware mechanisms would introduce.

In the past, researchers have developed a number of *software-only solutions* for inter-module isolation (e.g. [1,2,3]). These designs are superficially suitable for our purpose, but the fact that they have been designed with a completely different motivation in mind (i.e. to minimise the overhead of inter-module calls in traditional architectures) means that they have significant drawbacks when applied to our problem space. As discussed in detail in Section 4, these drawbacks include large binaries and high processing overheads that are not feasible on tiny embedded nodes. More recent examples of isolation-related work specifically targeted at the sensor node domain are t-kernel [5], which takes its cue from [2]; and Maté [4], which is an application specific virtual machine. But again, as discussed in Section 4, these systems have significant drawbacks—e.g. t-kernel suffers from large code size, Maté suffers from programming language specificity, and both suffer from high run-time overhead.

We therefore propose in this paper a software-only solution to the module isolation problem that attempts to overcome the problems of earlier work. In particular, our solution attempts to minimise memory and execution overheads at the sensor node by exploiting cheap CPU cycles at the pre-deployment stage while retaining a lightweight enforcement capability at the end-nodes. Our approach is also is programming language- and largely architecture-independent.

The remainder of this paper is structured as follows. Section 2 describes our design, and Section 3 considers the costs and other implications of our design. Finally, Section 4 compares our approach with related work and Section 5 offers our conclusions.

2. Design

2.1 Assumptions and Constraints

We assume that our target processors are RISC based MCUs with memory capacities in the order of kilobytes. We assume a generic RISC instruction set with a load-store architecture and support for standard addressing modes (i.e. immediate, direct, indirect and indexed). In terms of constraints our design forbids self-extending or self-modifying code and imposes a fixed-size block-based memory management scheme. These constraints, which are motivated by the need for efficient isolation, should have little or no effect on most applications.

2.2 Overall Architecture

Our design employs the following four phase architecture:

1. *Compilation*. Modules written in a high-level language are compiled to a *virtual instruction set*, which is close to a generic RISC instruction set architecture (ISA) but augmented with special 'emulated' instructions for memory management and calling across protection domains.
2. *Transformation*. VIS code is transformed offline to the target CPU's native instruction set, except for the

'emulated' instructions. This process also produces a per-module 'register partitioning policy' (see below) which is shipped to the end-nodes along with the native code.
3. *Verification*. A lightweight online verifier on the end-node ensures that the code is 'safe' before allowing execution.
4. *Execution*. Execution is dispatched to a small runtime environment when one of the emulated instructions is encountered.

The key advantage of our design is that the verification and execution processes are simple enough to be practical for tiny embedded nodes; the more complicated processes are carried out at the pre-deployment stage. Furthermore, the Trusted Computing Base (TCB) is small and manageable as it comprises only the online verifier and the small runtime environment.

The main concepts employed in our design are as follows. The scope of protection is called a *domain*, and each domain can hold a number of program modules and their associated data (including stacks). Both modules and their data are held in fixed-size *blocks* of memory. The *virtual instruction set* (VIS) provides instructions to create and destroy blocks and also to perform cross module and cross domain calls. When a block is created, the calling program receives a *handle* to the block. Memory within a block is then accessed either by specifying an indexed offset to the handle (i.e. indexed addressing) or through a non-forgeable *pointer* that is derived from the handle (i.e. indirect addressing). The transformation phase derives a *register partitioning policy* which maps handles and pointers to designated registers on which only subset of addressing modes are allowed.

Given the above concepts, the module isolation problem can be expressed as follows:

1. Within a domain, we constrain loads/stores to be from/to blocks owned by that domain.
2. Within a domain constrain, we control transfers to be to routines within that domain.
3. Where cross-domain calls are involved, we constrain control transfer so that calls may enter routines only at their 'official' entry point[1]. This also applies to calls to the runtime environment.

Next, we elaborate on VIS and each of the four phases.

2.3 The Virtual Instruction Set

The VIS is intended as a generic low-level intermediate language to which a range of high-level languages can be mapped. The bulk of the instruction set is close to a 'generic' RISC ISA with the exception that an 'infinite' register set is available (this is motivated by our register partitioning approach described in Section 2.5) and that only the following addressing modes are used: immediate, direct (used mainly with control transfer instructions and occasionally for memory mapped I/O), indirect (used with our special pointers) and indexed (used with handles).

VIS's 'emulated' instructions are as follows[2]:

crtb reg	*create a new data block and return a handle to it in reg*
crtp reg, offset	*given a handle in reg and a numeric offset, create a new data pointer*
dsrt reg	*destroy the data block whose handle is in reg*
ptoh src_r, dst_r	*cast a pointer in src_r to its parent handle and place the latter in dst_r*
salloc reg s	*allocate a stack frame of size s and place a pointer to it in reg*
call target	*do an intra- or inter-domain call to the given target address*
ret	*return from function*

The *crtb* instruction creates a new block and returns a handle to it. As mentioned, handles are used for indexed addressing. Thus, given a handle H, the expression X(H) would form an address within the block referred to by H. Index X could range from 0 to the maximum block size.

Given a handle, *crtp* can then be used to create a pointer to the block. This is simply a matter of performing a logical AND between the handle and the offset. This results in a pointer in which the high *n* bits refer to a block and the low *m* bits refer to an offset within the block. Pointers are used for indirect addressing and are safe in the sense that they cannot be made to point outside of their block. This is prevented by limiting the manipulation of pointers to the block offset bits. For example, in a deployment with 256 byte blocks, only the last *m*=8 bits of the pointer can be manipulated. Within these limits, which are enforced by mechanisms to be described in Section 2.5, arbitrary arithmetic/logic operations can be applied to pointers.

Each of the above instructions has the effect of placing the result (i.e. a handle or a pointer) in the specified virtual register. A pointer can be cast back to a handle using *ptoh*, and a block can be destroyed using *dsrt*. As well as handles and pointers to data, handles and pointers to code are also supported, as are handles to stack frames as discussed next. Stack frames are allocated using *salloc* which has the effect of placing a frame handle in the specified virtual register (which will later be mapped to the physical stack pointer register). Frame handles are similar to any other handle except that the indices have to be within the allocated frame (multiple stack frames within a block are supported).

Finally, the *call* and *ret* instructions support both intra and inter domain calls. Other control transfers can be handled using native instructions. For example, direct jumps (where the target address is given in the instruction) or branches are allowed where they are within a module.

2.4 Phase 1: Compilation

Apart from its block based memory management, VIS in many respects similar to a modern compiler intermediate

[1] Which functions a domain has access to is a matter of link-time access control and not is considered further in this paper.

[2] There are also instructions to create and destroy domains and to initialise domains with code blocks. However, these instructions are used only by the system loader/linker and are not considered further in this paper.

language [8] (i.e. with an infinite register file and generic RISC instruction set).

Although blocks are explicit at the VIS level, they are not necessarily visible to the programmer. For example, different compilers could choose to make the blocks explicit to the programmer [9,10] or to automatically infer allocation [11]). Furthermore, since stack management is handled by the runtime most block allocation is transparent to the compiler. However language libraries for dynamic memory management can easily be adopted on top of the block based scheme.

2.5 Phase 2: Transformation

The transformation phase is responsible for translating VIS to a specific target RISC ISA. Its other main job is to map handles and pointers to dedicated physical registers according to the afore-mentioned *register partitioning policy*. This is defined as a partition of the available physical registers into 4 categories[3]: i) data handles, ii) code handles, iii) pointers, and iv) general purpose.

A specific partitioning policy is selected for each module, the goal being to minimise register 'spills' in the context of the behaviour of the associated module[4]. For example, if a given module does not perform any dynamic memory allocation there is no need to dedicate any data handle registers.

The determination of partitioning policies is done through offline code analysis (inter/intra procedural). There are various tradeoffs to consider. For example, we want to minimise handle/pointer spills because they are costly in terms of processing cycles (as the runtime is involved in loading these). This suggests a partitioning policy that allocates many registers to handles/pointers. But if this is done, the number of general purpose registers is reduced, and spills from these are costly in terms of both memory usage (i.e. additional stack space) and code size (i.e. additional loads and stores).

Once a suitable partitioning policy is determined, register allocation and code generation can be performed. Depending on the target ISA, most VIS instructions translate one-to-one to the target ISA, except for the special 'emulated' instructions discussed in Section 2.3 which are handled by the runtime environment. Calls to the runtime environment are realised as direct mode native calls to predefined entry points (see Section 2.7). Pointer manipulation instructions are mapped to native instructions that manipulate only the low *m* bytes of the pointer (see Section 2.2). In cases where m=8 (and therefore the block size in 256), this is easily achieved by using the target ISA's byte instructions[5]. VIS's *call/ret* instructions involve calling the runtime environment because a switch between code with different register partitioning policies requires initialisation (see Section 3).

[3] Because of this, the register partitioning policy can be represented very economically: e.g. for 16 registers, 1 byte is sufficient.

[4] It is also possible to consider finer-grained register allocation policies such as per-function.

[5] In other cases, instructions that employ pointers would need to employ additional emulated instructions. We have so far only considered the case of m=8 and byte instructions.

Finally, the code is statically partitioned into blocks (normally the same size as data blocks). To deal with possibly non-contiguous code blocks, direct branches within a module are converted to index mode branches through code handles.

Thus, following the transformation phase, each module consists of number of code blocks and their associated register partitioning policy or policies.

2.6 Phase 3: Verification

When new code modules arrive at a node, static verification is performed on them before they are loaded for execution.

Direct jumps/loads/stores are easy to check as the addresses are literals embedded in the instruction. So it is only necessary to check that the target address is within the relevant block (or, e.g., within a permitted memory mapped I/O vector).

Indexed and indirect mode loads/stores/branches are verified using the attached register partitioning policy. There are two aspects to this. First, the verifier ensures that the dedicated registers are used correctly; for example, that arbitrary data is not loaded into handle or pointer registers, or that there are no memory accesses or branches through general purpose registers. Second, the verifier examines all instances of indirect and indexed addressing modes. Indirect mode instances are verified by first making sure that only the low *m* bits of pointers (i.e. those bits referring to offsets within a block) are ever modified by application code (e.g. by byte instructions in the case of 256 byte blocks; see Section 2.5). Since initialising a pointer is handled by the runtime (i.e. the high bits point at the block from whose handle the pointer was derived), the pointer will always point within the designated block. Code and data pointers are treated differently: no branches are allowed using data pointers, and no stores are allowed using function pointers (to disallow self modifying/extending code).

Indexed mode loads and stores are verified by checking that indexing is carried out through a dedicated handle register, and that all the indices (which are by definition embedded in instructions) are within the fixed block size. As the runtime environment guarantees that a loaded handle is necessarily correct, all indexing operations will therefore necessarily be within the block. Similarly, indexed operations through a frame handle are verified to be within the allocated frame (cf. the *s* argument to salloc).

Branches/calls within a module are not problematic as the register allocation policy is the same throughout a module. However, when control is transferred between two modules with different partitioning policies, then the dedicated registers need to be reset to ensure correct isolation. There are two cases. If the two modules are within the same domain, only registers whose usage changes from general purpose to dedicated need to be reset. Otherwise, for cross-domain calls all dedicated registers need to be reset (this is done by the runtime environment within the emulated *call* instruction).

Note finally that the integrity of the system does *not* rely on the correctness of the partitioning policy: if accidental or malicious manipulation of a policy makes it inconsistent with the associated code, the latter will simply be rejected (or if the manipulated policy *is* still consistent with the code then execution will anyway be confined to the respective domain). Therefore, we do not rely on cryptographic primitives to ensure code integrity.

2.7 Phase 4: Execution

The task of the runtime environment is to execute the emulated instructions specified in Section 2.3. It also implements the following additional routines to deal with handle and pointer register spills (calls to these are inserted by the transformation phase using native call instructions):

loadh reg_id, handle_id	*load a handle*
loadp reg_id, pointer_id	*load a pointer*
storep reg_id, pointer_id	*store a pointer*

The loading of a handle/pointer involves searching for the id in a list maintained by the runtime environment, and then loading the handle/pointer into the specified register.

The runtime also handles stack management through *salloc*. Stacks reside in—possibly non-contagious—block(s), and if a stack frame cannot fit within an existing block allocated for the stack, it will be allocated in a new block. The emulated *ret* instruction resets the frame handle.

There is a key issue around the calling of these runtime environment routines. In particular, it is necessary to ensure that the verifier can validate critical arguments passed to these routines (in particular, it is crucial to validate the *reg_id* argument to the above functions and the *target* argument to *call*). However, if we were to simply pass the arguments on the stack, there would be nothing to prevent code elsewhere from pushing some malicious arguments onto the stack and then jumping to a runtime environment entry point. The verifier could not detect such a loophole. To prevent this we must explicitly associate each call with its associated argument in such a way that the binding between the two can be verified statically. To achieve this, the transformation phase embeds the arguments in the word(s) immediately following the *call* instruction (this is feasible because the critical arguments are known at compile time. Since the native 'call' instruction is assumed to automatically push the PC on the stack, the runtime routine can then reliably access the arguments to which the PC is pointing (and then increment the PC to the next instruction within the routine). This, together with the fact that code is not re-writable, forms a reliable basis on which the verifier can treat the call and its arguments as one indivisible unit.

The above-described mechanisms, together with the action of the verifier, are sufficient to secure the required isolation semantics as set out at the end of Section 2.2.

3. Design implications and costs

Our design is currently under implementation and is not yet sufficiently mature for a comprehensive performance evaluation. In the meantime, we briefly analyse the overhead implications of the design. We focus on runtime overheads and ignore the offline costs of compilation and transformation: these latter rely on well understood compilation principles and are anyway not time critical.

Runtime overheads comprise verifier and execution environment overheads. *Verifier overheads* are limited to making a single pass over the code, carrying out pattern matching to detect relevant instructions (i.e. those involving direct, indirect and indexed addressing) and checking them against the register allocation policy as described in Section 2.6. This involves a small and fixed per-instruction overhead which is well within the capabilities of an embedded node.

In terms of *execution overheads*, the main factors are the cost of VIS instruction emulation and the effects of the register partitioning policy (in terms of spillage). The emulated instructions with the highest overheads are *load*, *crtp* and *call*. The former two instructions save two registers, locate *handle_id* in a list, and determine *reg_id* from the code block. *Crtp* additionally involves a logical AND of *index* and *handle_id*. The cost of these steps on an MSP430 MCU is corresponds to ~20 instructions assuming a list of 5 handles. The overhead for *call* involves inspecting the stack pointer to ensure that there is enough room for another call before current call returns. In addition, all cross-module calls need to zero the handle and pointer registers to ensure isolation. Again, these operations involve only modest overhead.

The effects of the register partitioning policy are less easy to quantify as they are heavily dependent on peculiarities of a particular piece of code. The selection and evaluation of an optimal register algorithm remains an area for future work.

There are a number of other smaller sources of overhead: in particular, there is a degree of memory wastage due to internal fragmentation caused by the use of fixed-sized blocks and there will be a small increase in executable size due to dealing with dedicated register spillage. Note, however, that this memory overhead is likely to be tiny in comparison to other systems (see Section 4).

Finally, as most VIS instructions map directly to native equivalents, most code executes natively without any execution overhead whatsoever. Memory access incurs no additional cost as all the verification has been carried out prior to execution, and native instructions/ addressing modes are used.

4. Related work

As mentioned in the introduction, enforcing module boundaries in a single address space has been extensively studied in traditional architectures. For example, Software Fault Isolation [2] achieves isolation by binary code rewriting: first the code is statically checked to ensure that all direct jumps are within the monolithic code segment assigned to the module; then dynamic checks are inserted into the code at each indirect jump and store to ensure that these are also 'safe'. Unfortunately, in sensor nodes with little memory, allocating continuous regions for each module is not practical. Also, fully-general reliable binary rewriting is a complex task that is not practical on tiny embedded nodes. Furthermore, facilitating offline-rewriting would require an underlying cryptographic infrastructure (i.e. to enable the embedded node to verify the integrity of the rewritten code). This would carry undesirable overhead for embedded systems environments.

Actually, our design can be considered as a generalised form of SFI. However there are 4 major differences: i) no binary rewriting at the target node is necessary (conceptually, this is moved to the transformation phase); ii) our scheme allows multiple blocks of code and data in each domain (allocating single monolithic areas is infeasible on small embedded nodes); iii) because of the use of fixed sized blocks we eliminate most of the dynamic checks required in a SFI system; and iv) our use of dedicated registers is different: while the classic SFI scheme requires 3 dedicated registers (segment start, length (shift), formed address) for forming an address within a block, we need only a single register, the one that actually contains the formed address. This reduced register overhead is particularly important for MCUs with small register files.

T-kernel [5] is a good example of other work on adapting the SFI approach to the sensor network domain. However, t-kernel introduces large code size overheads (on average a factor of 8) due to the extensive rewriting employed. Also, t-kernel incurs high runtime overhead—for example, heap access can take 180,857 cycles. Execution cost in our scheme is minimal because memory accesses are mapped to native instructions, and handle loads/pointer-initialisations cost only around 20 instructions.

Proof Carrying Code [3] achieves isolation by requiring the construction of a 'proof' which verifies the software's compliance with the target host's 'safety policy'. This approach is superficially attractive for the sensor node context as the complex task of proof-generation is done offline, and the node performs only a 'simple' validation. Unfortunately, proofs tend to be much larger than the code itself (up to 8 times the code size [3]), and validating the proof, written using the Edinburgh Logic Framework (LF), is again a task that is beyond tiny embedded nodes.

SPIN [1] approaches the issue of isolation by leveraging high-level language semantics. However, there is an explosion of different languages and programming paradigms in use in the network embedded systems field (e.g. declarative, agent-based [13], global-programming [12]) and mandating a single type-safe language such as Modula, or even small set of such languages is therefore infeasible or at least undesirable. Also, code for embedded devices is often written in type-unsafe languages such as C, and hand-tuned assembler code is still common in the field. Furthermore, the same requirement for a cryptographic infrastructure that we noted above for SFI would equally apply to language based approaches.

Another approach to implementing modularity and protection is through high-level application specific virtual machines, such as Maté [4]. But the cost of full interpretation as used in such designs is costly and soon becomes a limiting factor for computationally intensive applications (e.g., Maté's emulated add instruction costs a few hundred cycles).

5. Conclusions

Module isolation in embedded nodes is an increasingly important issue. We have described a language, and largely architecture-independent, solution that attempts to move most of the overhead of enforcing isolation to a pre-deployment stage, while retaining a minimal but sufficient enforcement mechanism at the end-nodes.

Essentially, our approach employs the notion of dedicated registers coupled with per-module register allocation policies to restrict addressing modes in such a way that all instructions can be verified prior to runtime to access only fixed ranges of addresses (i.e. within blocks). Apart from this sandboxing, normal addressing semantics are maintained and, furthermore, considerable flexibility is maintained at runtime—i.e. dynamic code loading, memory allocation and block relocation are all supported.

Currently, we are completing our implementation and are also writing a formal model of the design. Beyond that, the issue of determining optimal register allocation policies based on code analysis is a key area for further investigation. This will build on existing compiler technology (which of course already needs to appropriately allocate registers to minimise spillage) but will need to be refined to work at the finer granularity of our 4 register categories.

6. References

[1] B.N. Bershad, et al., Extensibility, safety and performance in the SPIN operating system, Proc. 15th ACM SOSP, December 1995.

[2] R. Wahbe et al., Efficient software-based fault isolation, Proc. 14th ACM SOSP, December 1993.

[3] G. Necula et al., Safe kernel extensions without run-time checking, Proc. OSDI, Seattle, USA, October 1996.

[4] P. Levis and D. Culler., Maté: A tiny virtual machine for sensor networks, Proc. ASPLOS-X, October 2002.

[5] L. Gu et al., t-kernel: Providing Reliable OS Support to Wireless Sensor Networks, Proc. 4th ACM SenSys, 2006.

[6] Ram Kumar, et al., Software-based memory protection in sensor nodes, Proc. 3rd EmNets, 2006.

[7] MSP 430 specification, http://focus.ti.com/mcu/docs/.

[8] Vikram Adve, et al., LLVA: A Low-level Virtual Instruction Set Architecture, Proc. ACM/IEEE MICRO, 2003.

[9] David Gay et al., Memory management with explicit regions, Proc. PLDI, 1998.

[10] Grossman, et al., Region-based memory management in cyclone, Proc. PLDI, 2002.

[11] M. Tofte, et al., Region-based memory management. Information and Computation,132(2):109–176, 1997.

[12] R. Newton, et al., The Regiment Macroprogramming System, Proc. IPSN'07, 2007.

[13] Chien-Liang Fok, et al., Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications, Proc ICDCS, 2005.

[14] J. Hill, et al. System architecture directions for networked sensors, Proc. ASPLOS-IX, 2000.

[15] Steffan, A. et al. Towards Multi-Purpose Wireless Sensor Networks, Proc. Systems Communications, 14(17):336–341, August 2005.

[16] ARM7 specification, http://www.arm.com/products/ CPUs/ ARM7TDMI.html.