Hardware-assisted Security Enhanced Linux in Embedded Systems: a Proposal

Leandro Fiorin Alberto Ferrante Konstantinos Padarnitsas Stefano Carucci

ALaRI - Faculty of Informatics - University of Lugano
Via Buffi 13, 6904, Lugano, Switzerland

name.surname@usi.ch

ABSTRACT

As computing and communications increasingly pervade our lives, security and protection of sensitive data and systems are emerging as extremely important issues. This is especially true for embedded systems, often operating in non-secure environments, and with limited amount of computational, storage, and communication resources available. In servers and desktop systems, Security Enhanced Linux (SELinux) is currently used as a method to enhance security by enforcing a security control based on policies that confine user programs, or processes, to the minimum amount of privileges that they require for their execution. While providing a powerful mean for enhancing security in UNIX-like systems, SELinux still remains a feature that is too heavy to be fully supported by constrained devices. In this paper, we propose a hardware architecture for enhancing security and accelerating retrieval and applications of SELinux policies in embedded processors. We describe the general ideas behind our work, discussing motivations, advantages, and limits of the solution proposed, while suggesting the main steps needed to implement the described architecture on common embedded processors.

Categories and Subject Descriptors

C.0 [**General**]: Hardware/software interfaces; C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems; D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*

General Terms

Design, Performance, Security

Keywords

SELinux, Embedded Systems, Access Controls

WESS '10 Scottsdale, Aritzona, USA

1. INTRODUCTION

Embedded system complexity is being increased constantly: the increasing computational capabilities available at low price allows developers to add new functionalities to these devices. In-vehicle embedded systems have become very complex providing network connection, multimedia, and complex vehicle control functionalities. The latest generations of mobile phones have huge computational capabilities and, therefore, they provide the ability to run complex applications. In particular, in most modern embedded systems users have the ability to install new applications, and, thus, add new functionalities. These new capabilities are exposing embedded systems to security risks similar to those of personal computers. In fact, the capability of installing software, and the fact that embedded systems are now network-enabled, expose them to an increasing number of possible attacks. In particular, viruses, malwares, and other malicious pieces of software represent nowadays a real danger.

Common methods for contrasting these threats, such as for instance the use of anti-viruses, are not compatible with embedded systems, or too heavy to be supported by constrained devices. Furthermore, these methods have already proved to be only partly effective even on personal computers. One of the methods that proved, instead, to be effective in protecting personal computers and servers is the implementation of process isolation. The access of processes to resources and to the other processes running in the system can be controlled by the operating system and denied when necessary. Process isolation can be obtained in different ways; some operating systems provide possibilities for implementing process isolation natively.

In the Linux kernel a technology named Security Enhanced Linux (SELinux) was introduced back in 2003. SELinux is a flexible mandatory access control method that can be used for controlling the access of all resources by processes. It is completely managed at kernel level and, thus, it cannot be easily broken by user-level processes. SELinux is extremely flexible and it can be configured by customizable policies; these policies define the operations that each *object* in the system (e.g., a process) can perform on *subjects* (files, processes, resources, ...). By defining suitable policies, non-safe processes can be run without the risk of influencing safe processes. For example, a proper policy should provide such processes the minimal rights to run, but it should provide no right to access certain system resources and other processes in the system. SELinux is presently widely used especially on servers. For example, it is presently enabled by default on all Red Hat and Fedora Linux distributions.

In these distributions pre-made policies are provided along with tools for controlling the behavior of SELinux and for customizing the policies [1].

It would be desirable to be able to provide such a fine grained isolation for processes in embedded systems. Unfortunately, though, SELinux has not been designed for such systems as it requires to check the access rights of objects for each operation performed on subjects. This introduces an overhead that is not compatible with the resources (computational resources and energy) available in embedded systems [2]. In the present work, we propose to optimize the execution of SELinux in embedded systems by using custom designed hardware accelerators. These hardware accelerators, accessed from the operating system by using proper Application Programming Interfaces (APIs), should provide the ability to overcome performance limitations of SELinux in embedded systems. The hardware can also be used, along with proper policies, to further enhance security.

The remaining part of this paper is organized as follows: a description of SELinux is provided in Section 2; Section 3 presents a comparison of our work with related works. In Section 4, a description of the proposed solution is provided; in particular, the hardware accelerator for SELinux is described along with its interface with the operating system. In Section 5, the resources required for implementing the proposed solution are estimated and discussed.

2. SECURITY ENHANCED LINUX

Security-Enhanced Linux (SELinux) [3] was presented in 2001 by the National Security Agency, and it is a security module for fine-grained mandatory access control which was integrated in the Linux kernel since its 2.6.0 release. Since Red Hat Enterprise 4, SELinux is enabled by default in Red Hat distributions. The protection provided by SELinux aims at preventing malicious processes from reading data and programs, tampering with data and programs, bypassing application security mechanisms, executing untrustworthy programs, or interfering with other processes in violation of the system security policies. SELinux is an implementation of the Mandatory Access Control (MAC). Depending on the security policy type, SELinux implements either Type Enforcement (TE), Roles Based Access Control (RBAC) or Bell-La Padula Model Multi-Level Security (MLS). SELinux defines the access and transition rights of every user, application, process, and file on the system, and then governs the interactions of these entities using a security policy that specifies how strict or lenient a given Linux system should be [4].

Access control is specified by policies. A policy is a set of rules that put constrains on the system behavior. Access to a system resource is granted only if a corresponding allow rule exists in the currently loaded policies; by default, no access is granted by SElinux; all required accesses must be explicitly described in the associated policy. A rule is composed of the triple *subject*, *object*, *object class*, and of an associated access vector specifying the type of access or operation allowed for them. The elements of the triple represent, respectively, the running process, the system resource to be accessed, and its class (i.e., the kind of resource considered). Table 1 shows a selected set of object/rules classes, providing some example of permissions; an exhaustive classification of all rules and permissions can be found in [5].

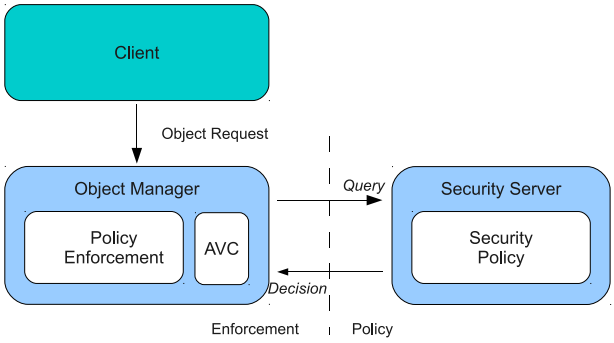

Figure 1: Overview of the Flask architecture. *Object Managers* enforce security policy decisions, while the *Security Server* provides security decisions to the Object Managers [6]

As mentioned earlier the policy is queried and enforced prior to each access requested by objects on subjects. This implies that each access must be authorized by the kernel after a query of the policy. Most recently used rules are cached to speed up their lookup. SELinux is based on the Flask architecture [6], which provides flexible support for different mandatory policies. Figure 1 shows the Flask architecture, which is composed of three main components: Object Managers, the Security Server, and the Access Vector Cache (AVC).

2.1 Object Managers

When SELinux is adopted, object managers enforce the policy decisions of the security server for the set of resources that they manage. Standard object managers are modified to obtain security policy decisions from the security server and to apply these decisions to label and to control access to their objects [5]. Examples of Object Managers include kernel subsystems such as the filesystem, the process management, the socket Inter-Process Communication (IPC), and the System V IPC. Depending on their functionality, they are implemented either at system- or at user-space. Three primary elements are provided for Object Managers by the Flask architecture. First, the architecture provides interfaces for whether a particular permission is granted between two entities, for specifying the security attributes to be assigned to an object, and for specifying which member of a set of resources should be accessed for a particular request. Second, the architecture provides an access vector cache (AVC) module that allows the object manager to cache access decisions to minimize the performance overhead. Third, Object Managers are given the possibility to receive and register notifications of changes to the security policy.

SELinux is integrated into the kernel through the Linux Security Modules (LSM) framework [7]. In the LSM architecture, the object managers are represented by the LSM hooks; these hooks are distributed throughout the kernel subsystems and call the SELinux LSM module for access decisions. The LSM hooks then enforce those decisions by allowing or denying access to the kernel resource.

2.2 Security Server

The *security server* mainly consists of a database where the policy rules are stored and retrieved, and a decision mak-

Table 1: Rules classification in SELinux

Rule Class	Object Class	Permissions Example
Process Management	Process	Signals (sigkill, sigstop, sigchld), fork, transition, get_attribute.
File System	Pipe, file, directory	create, read, write, append, lock, link, get_attribute.
Networking	Socket	start, shutdown, bind a connection, receive, send a msg.

ing logic able to take a decision upon an access check, according to the corresponding access vector. It also provides functionality for loading and changing policies both at boot and at run-time. In SELinux, the policy database is implemented as a common single linked list hash-table [8] (the Access Vector Table *avtab*), which is composed of a pointer to the head nodes, the number of slots, and a mask for the hash function computation. Through the hash, a set of rules, grouped in a list, are found. A further search in the list is required to identify the rule. In fact, the rules contained in the list are the ones having colliding hashes. The key used for retrieving the permission is the hash of the concatenation of the *subject identifier* (SSID), the *target identifier* (TSID), and the *object class* (OBJC). An access vector stores the permission for that specific key.

2.3 Access Vector Cache

The *AVC* caches decisions made by the security server for subsequent access checks and, thus, it provides significant performance improvements. Whenever a resource is requested, the AVC is first searched for a decision previously taken. In the case of a miss, the query is forwarded to the policy database into the security server, and the decision of the security server is finally stored in the AVC for future occurrences. The information stored in the AVC is replaced following a Least Recently Used (LRU) policy, therefore exploiting access temporal locality. Its size is small to improve the search procedure performance and ease memory requirements. The AVC also provides the SELinux interfaces with the Linux security modules (LSM) hooks and with the kernel object managers. Similarly to the policy database, in SELinux the AVC is implemented as a common single linked list hash-table, and the key used for retrieving the permission is the hash of the concatenation of the SSID, the TSID, and the OBJC.

3. MOTIVATIONS AND RELATED WORK

SELinux is a powerful tool to overcome intrinsic security weaknesses of the Linux kernel access control mechanism. Its usage has been proven to be efficient and reliable in servers and personal computers. Being focused on this type of systems, many features were added during its development, and its resource consumption and overhead in term of CPU usage, file size, and memory, has become unacceptable for embedded computing [2]. While the overhead due to the security checks in system calls may appear insignificant for PCs and servers[5], it is not for embedded systems. A SELinux overhead of more than 100% has been reported, especially for read/write operations, which are amongst the ones executed the most [2, 9]. Moreover, the size of the operating system image (composed by the kernel and the userland) is increased by 2MBytes when SELinux is included. This size increase may be a problem for most embedded devices which are equipped with at most 32MBytes of read only memory. Similarly, SELinux increases memory consumption due to the necessity of storing policies.

In [2], the authors proposed to tune SELinux for reducing the resource consumption on consumer electronic devices. The tuning consists of two parts, i.e., the reduction of the policy size by utilizing a specific policy writing tool [10], and the tuning of kernel and userland by removing unneeded functions and redundant permission checks from the kernel, by reducing file size by removing features unnecessary for embedded devices, and by reducing memory usage by removing unnecessary data structures from the kernel. On the porting of SELinux on embedded devices, [9] describes the integration of Security-Enhanced Linux LSM with Android-powered devices, evaluating its ability to improve Android security. Despite these optimizations, notable slowdowns in the operations requiring SELinux check were reported.

In this work, to the best of our knowledge, we propose for the first time the use of hardware accelerators for enhancing SELinux performances on embedded devices. Our solution is complementary to the ones proposed in the related work and it can be applied together with them for limiting the overhead due to SELinux. In particular, we propose to trade off on-chip area and some flexibility for performance (and reduced power consumption), a simpler memory management for the policies, and an increased security. Our work can be considered complementary also to [11], where a mandatory access control mechanism alternative to SELinux designed for mobile devices is proposed and evaluated.

Regarding related work addressing the isolations of processes and resources in embedded platforms, ARM [12] introduced the concept of the TrustZone Platform [13]. A TrustZone architecture is divided into secure and non-secure regions. A Non-Secure indicator bit determines the security operation state of the various components and it can only be accessed through a *Secure Monitor* processor mode. This secure mode is accessible only through a limited set of entry points. This mode is allowed to switch the system between secure and non-secure state, allowing a core in the secure state to gain higher levels of privilege. Support exists for secure interconnection transactions, and for secure memory accesses. Similarly, Sonics [14] provides in its SMART Interconnect solutions an on-chip programmable security "firewall" to protect system integrity and media content processed. The access mechanism aims at protecting memory regions from illegal accesses, and at supporting secure transactions between the System-on-Chip cores.

Compared to the aforementioned types of hardware protection, our solution provides a higher level access mechanism, targeting the system objects instead of low-level memory transactions. Our solution provides more flexibility to the system designer, exploiting the possibilities in terms of different types of permissions foreseen by the Flask architecture.

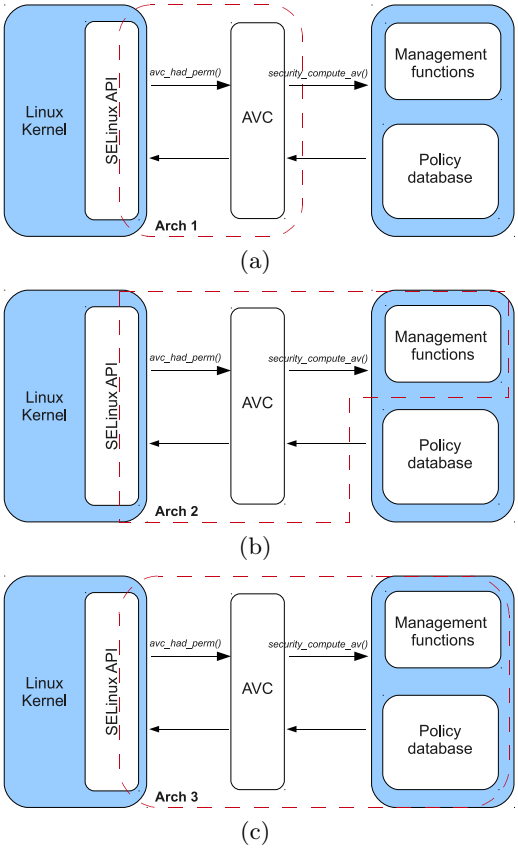

Figure 2: Possible alternative architectures for implementing hardware support for SELinux: (a) AVC; (b) AVC and main security server operations; (c) AVC and security server

Regarding the use of hardware accelerators for Operating Systems (OSs), some previous work can be found in the literature. In [15], tasks management in Multiprocessor Systems-on-Chip (MPSoCs) is supported through an ASIP-based solution, by adopting a processor core specifically designed to provide OS support to the multiprocessor system. Similarly, in [16] a coprocessor is proposed for performing scheduling in a homogeneous cluster of simplified RISC processors, with a low context switch overhead. In [17] and [18], programming support is also provided. In particular, [18] proposes a hardware real time operating system (HW-RTOS) that implements the OS layer in a dual-processor SMP architecture, and that allows specifying inter-task communication by means of dedicated APIs. The HW-RTOS takes care of the communication requirements of the application and also implements the task scheduling algorithm, with smaller footprints, and exploiting efficiently task migration features provided by an SMP architecture.

Our solution adopts the idea of using hardware accelerators for improving performances of operations performed by the OS. While related work focus on providing hardware support for OS operations mainly for improving real-time capabilities of the system, our work focuses on improving performance in the application of the security operations needed when running SELinux.

Table 2: Qualitative comparison between the three discussed alternative hardware architectures for supporting the execution of SELinux

Characteristic	Arch 1	Arch 2	Arch 3
Flexibility	high	medium	low
Area overhead	low	medium	high
Expected performance improvement over cost	high	low	low

4. HARDWARE-ACCELERATED SELINUX

In the present paper we propose to implement some parts of SELinux in hardware, with the goal of speeding up the policy enforcement when any action is executed. In particular, our hardware accelerator aims at speeding up the most time consuming operation in policy enforcement: the lookup of rules in the policy. As discussed in Section 2, rules lookup is performed through a multiple-layer structure composed of the AVC and the policy database; the lookup is the cause of the high performance overhead reported when SELinux is used in embedded devices [2]. Several alternative hardware implementations can be derived from the Flask architecture shown in Figure 1, as expression of different design trade offs between the performance improvement obtained and the additional costs in area of the hardware accelerator.

As shown in Figure 2, different levels of acceleration may be identified:

- **Arch 1 - acceleration of AVC**: the AVC module is substituted by an hardware module that implements its functionalities (Figure 2(a)). The block is mainly composed of a lookup table for storing the rules cached and of the logic for retrieving them, as well as of all the logic for interfacing the block to the processor and the function calls of the Object Managers in the Linux kernel;
- **Arch 2 - acceleration of AVC and security server operations**: in addition to the previous module, also a selected set of operations performed by the security server is implemented in hardware (Figure 2(b)). Possible operations are, for instance, the Least Recently Used (LRU) function for the replacement of the policy in the AVC, the retrieval and decoding of the information rules information stored in the binary policy file. In this architecture, the management of the policies in the policy database, and its memory allocation is still left to the software implementation;
- **Arch 3 - acceleration of AVC and security server**: as shown in Figure 2(c), both the AVC and the security server are implemented as hardware modules. The policies are stored in a separate external memory, and (almost) all the SELinux operations are supported by equivalent hardware modules.

Table 2 compares the different implementations and shows preliminary qualitative evaluations about the three types of architectural solutions. The evaluations on performance improvements are based on the fact that, with a sufficient amount of AVC cache the number of misses should be limited. To support this hypothesis we profiled the behavior of the AVC on a personal computer equipped with a 2.67GHz Intel core i7 M60, 4GByte of RAM and *Kubuntu* 10.04. In

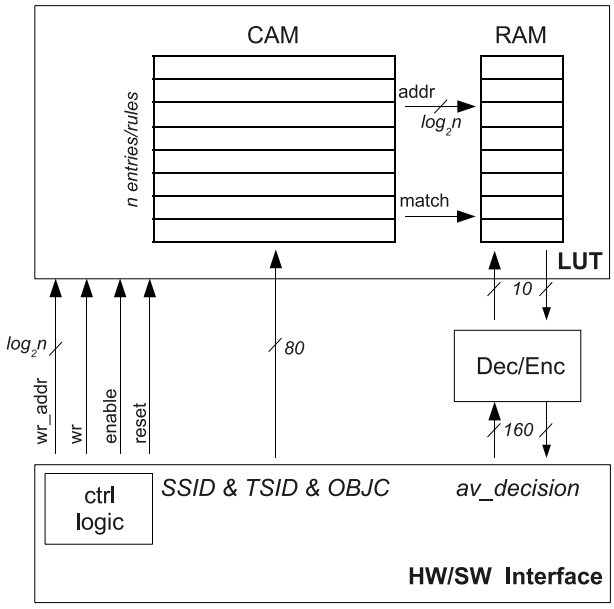

Figure 3: Overview of the Hardware Access Vector Cache (HAVC) architecture

8 hours of experiments, running different applications, the number of misses in a 512 entries AVC was of 0.0005%. We expect that, in an embedded device, we can obtain similar results even with much smaller AVCs, due to the limited number of applications usually run in such systems.

In the present work we have chosen to study the first proposed hardware architecture, i.e., the implementation of a hardware module for substituting the operations performed by the AVC. As reported by [2] and [9], the access to the AVC is in fact the main source of performance overhead in the SELinux, and the architecture represents a good trade-off between the expected performance improvement and the implementation costs.

4.1 Hardware implementation of the AVC

In SELinux, the AVC stores the most recently used rules and it is designed to speed up the policy search by exploiting the temporal locality of rule access; the policy database stores the full set of rules that compose the policy. As mentioned in Section 2, both modules are implemented in the kernel as hash-tables with a non-perfect hash function. The rule lookup task, implemented both in the security server and in the access vector cache, can be efficiently implemented in hardware. Therefore, it provides a clear opportunity for speeding up SELinux with relatively low hardware costs.

Figure 3 shows the proposed architecture, the Hardware Access Vector Cache (HAVC). HAVC is a hardware implementation of the AVC cache found in the software implementation of SELinux. HAVC is queried by the CPU at every security check. The main component of the HAVC is a relatively small-sized lookup table, implemented as a Content-Addressable memory (CAM) [19] and an associated RAM. The CAM stores the keys that are looked up for retrieving the associated access rules, while the RAM stores the associated permissions. The number of entries of the lookup table depends on the implementation and on the

```
struct av_decision {
        u32 allowed;
        u32 auditallow;
        u32 auditdeny;
        u32 seqno;
        u32 flags;
};
```

Figure 4: Data structure returned by the *avc_has_perm()* and containing the permissions for the desired input key

applications running on the system. A design exploration is required to select the ideal size of the look-up table, as trade-off between the area of the module and the number of misses in the AVC. An approach similar to the one followed in [20] can be used in this design phase.

Input of the HAVC during the lookup phase are the parameters characterizing the specific rules, i.e., the *subject identifier* (SSID), the *target identifier* (TSID), and the *object class* (OBJC). These three values are equivalent to those passed as argument of the original SELinux function *avc_has_perm()*, which is called in the software implementation for requesting the lookup of the permission in the AVC. In hardware hash tables are not useful anymore. The lookup can be implemented more efficiently by using a concatenation of the aforementioned parameters (*SSID*, *TSID*, and *OBJC*) as research key in the CAM. As shown in Figure 3, the lines of the CAM are designed to be 80-bit wide, in order to store the whole key. Each entry in the CAM structure indexes a RAM line containing the access rights for the specific rule.

In the software implementation, the information returned after a rule lookup is the *av_decision* struct that is composed of 5 32-bit integers (Figure 4). The member *allowed* contains the identifier permission stored for the associated key (an example is reported in Table 1). *auditallow* and *auditdeny* are employed by the kernel for auditing purposes, in order to derive statistics about SELinux performances, such as the number of the AVC hits and misses. The member *seqno* specifies the sequence number of the policy file which is used (useful in the case of run-time modifications of the policies to distinguish between old and new ones), while *flags* is used to specify whether SELinux is in a *enforcing* or a *permissive* mode [1]. In the hardware implementation, we encoded the *av_decision* with 10 bits, which also gives the width of the RAM lines. 5 bits are in fact used for encoding the maximum number of different types of permission that it is possible to have for an Object Class, while 1 bit is needed for the *auditallow*, the *auditdeny*, and the *seqno* members. Two bits are instead needed for encoding the content of the *flags* field. As mentioned, the SELinux permission data are represented in different ways in the software implementation and in the RAM of the HAVC. Thus, an encoding/decoding hardware module is required to interface the RAM with the SELinux processes.

Further optimizations can be performed on the RAM size. In fact, only the *allow* field can be considered mandatory for applying the basic SELinux functions. By excluding the use of logs and assuming a permanent *enforcing* operating mode of SELinux and a fixed policy file, the width of the RAM can be reduced to 5, thus decreasing the overhead associated to the HAVC.

When a miss occurs in AVC (i.e., the lookup fails), the module makes a call to the *Security Server* for retrieving the required rule from the policy database. The rule is copied as last used in the AVC, and passed to the calling function of the kernel. As already partially discussed, a part from the lookup table, the HAVC implements also all the interfaces and the conversion logic from the data types passed as arguments in the SELinux functions to the logic bits used for storing the associated information, and vice-versa. Moreover, it implements all the control logic needed for performing the lookup efficiently.

4.2 Integration With the Operating System

The hardware implementation of the AVC requires modifications to SELinux to work. In particular, proper interfaces for the hardware accelerators should be added and the related function calls in the SELinux code need to be modified accordingly. The goal of our hardware design have been not only to improve performance and security, but also to minimize the changes required in the Linux kernel code. A SELinux hardware API (Application Programming Interface) is being added to the kernel. The goal of an API is, in general, to provide an uniform access to hardware and software functionalities. In this specific case, the API will provide access to the functionalities implemented in hardware that will be substituting corresponding software functions.

To make use of the HAVC implementation presented above, the AVC query function will need to be modified. The AVC works as a policy cache; in a pure software implementation when an object wants to access a subject (e.g., an application requires to access a resource), the AVC is queried first by using the *avc_has_perm()* function. If the policy corresponding to the required access has been stored in the AVC previously, the permissions are returned; the security server needs to be queried instead. The security server is directly queried by the AVC module that stores the results before sending them to the Linux kernel to allow or deny the required operation. When an AVC accelerator is adopted, the *avc_has_perm()* has to be modified as follows:

- the AVC query (*avc_lookup()* function) should be substituted by a call to the API.
- If the hardware returns no valid result (i.e., the entry is not present in the HAVC) the function will query the security server. After the query to the security server, the result is sent to the HAVC for being stored through a call to the API that substitutes the *avc_insert()* function. The result is also sent to the operating system to evaluate if the required operation is legitimate or not.

As aforementioned, other hardware accelerators might also be considered. The adoption of a hardware implementation of the policy server (*Arch2* and *Arch 3* in Figure 2) will also provide the ability to simplify the procedure when the entry is not found in the HAVC. The AVC hardware implementation, in fact, will be able to directly query the hardware implementation of the policy server (that will implement fully the *security_compute_av()* function), store the results, and send them back to the query software. In practice, the *avc_has_perm()* function will be fully implemented by the hardware accelerators. Therefore, the API will substitute this function with a proper hardware call. The API will be also required to implement a *selinux_init_load_policy()* to load the policy in the hardware policy server.

Table 3: Area overhead (in mm^2) of the HAVC for full and reduced implementation, while varying the number of entry lines of the lookup table

# lines	Full	Optimized
64	0.0545	0.0527
128	0.0970	0.0948
256	0.1286	0.1247

Table 4: Area (in mm^2) of a full implementation of a 128 lines HAVC compared with the area of a ARM926EJ-S processor

HAVC	ARM926EJ-S
0.0970	1.40

In the future the modifications to the SELinux code can be further minimized by exploiting the Linux Security Module (LSM) [3] capabilities. LSM provides a general kernel framework that provides the infrastructure to support security modules. HAVC call may be inserted as alternate security function calls in LSM.

5. EVALUATION AND PRELIMINARY RESULTS

Table 3 shows the cost, in term of area (mm^2), of implementing the Hardware Access Vector Cache in ASIC by using a $90nm$ technology. The values reported were obtained by using CACTI 5.3 [21] for estimating the area of the RAM and the CAM in the HAVC. As it is possible to notice, the area overhead is proportional to the number of HAVC lines, being in fact the lookup table the larger component in the architecture (more than 99% of the total area in the 128 lines full configuration). The table shows the results obtained when considering two different cases:

- a system with full SELinux support, with run-time adaptation of policies (*Full* in Table 3);
- a minimal system, in which only the permissions capabilities are considered (*Optimized*).

The first system is more flexible and it provides the possibility to perform auditing; the second, instead, has limited flexibility and it does not support audit. The second system is less memory hungry then the first one as it requires to store less information; the difference in memory size can be quantified in 5 bits for each rule. Furthermore, the second solution introduces a smaller overhead than the first one due to the absence of auditing and to the smaller size of rules to be loaded. The first or the second solution (or any intermediate solution) can be adopted depending on the system considered. The best solution for a specific application should be chosen by considering the trade-off between costs and flexibility. Results are shown for different dimensions of the HAVC, i.e., with different capabilities in term of number of rules that the HAVC is able to store.

Table 4 compares the area occupied by the HAVC with the size of a commercial embedded processors. We selected for the comparison an ARM926EJ-S with 8KBytes of data and instruction cache, speed optimized, running at the maximum frequency of 470 MHz [12], and implemented by using $90nm$

technology. We considered a full implementation of a HAVC with 128 entry lines. As the table shows, the overhead introduced by the HAVC implementation is of about 7% of the processor area, which we believe to be an acceptable price to be paid for the advantages in terms of enhanced security and performance improvement that the proposed module could provide.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed a solution for reducing the performance overhead given by the adoption of SELinux in embedded systems. A hardware accelerator for key parts of SELinux has been proposed and an evaluation of its cost have been performed. SELinux will enhance security of Linux-based embedded systems, by providing, in particular, the ability to isolate non-secure processes. While the SELinux accelerator is being designed with embedded systems in mind, it might also be adopted on general purpose computer architectures and on servers: it might improve performance of high-load machines. The possibility to write policies in the hardware (e.g., in a flash memory) will definitely improve security by physically forbidding unauthorized changes of the policies.

Future work will first consist in completing the implementation of the proposed accelerator and on testing its performances. The accelerator is being implemented on a Xilinx development board which contains a PowerPC processor along with a FPGA module. The PowerPC runs Linux; the accelerator is being implemented in the FPGA. A study on optimal sizes of the HAVC will be conducted for different kinds of embedded systems. Furthermore, a SELinux policy targeted for embedded systems will also be developed. The policy will be designed to be as simple as possible and to implement, as a basic feature, process isolation. The policy will have the possibility of being customized for specific embedded applications. The aforementioned application of a hardware SELinux accelerator to servers and to general purpose computer architectures will also be studied.

7. REFERENCES

[1] "Red Hat SELinux Guide," http://www.redhat.com/docs/manuals/enterprise/RHEL-4-Manual/selinux-guide.

[2] Y. Nakamura and Y. Sameshima, "SELinux for Consumer Electronics Devices," *Proceedings of Linux Symposium*, pp. 125–133, 2008.

[3] S. Smalley, C. Vance, and W. Salamon, "Implementing SELinux as a Linux Security Module," US National Security Agency, Tech. Rep., 2001, http://www.nsa.gov/research/_files/publications/implementing_selinux.pdf.

[4] "Red Hat Enterprise Linux Deployment Guide," http://www.redhat.com/docs/manuals/enterprise/RHEL-5-manual/Deployment%_Guide-en-US/selg-overview.html.

[5] P. Loscocco and S. Smalley, "Integrating Flexible Support for Security Policies into the Linux Operating System," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 29–42.

[6] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies," in *in Proceedings of The Eighth USENIX Security Symposium*, 1999, pp. 123–139.

[7] C. Wright, C. Cowan, and J. Morris, "Linux Security Modules: General Security Support for the Linux Kernel," 2002.

[8] "Hash Table definition - Wikipedia," http://en.wikipedia.org/wiki/Hash_table.

[9] A. Shabtai, Y. Fledel, and Y. Elovici, "Securing Android-Powered Mobile Devices Using SELinux," *IEEE Security and Privacy*, vol. 8, pp. 36–44, 2010.

[10] "Selinux Policy Editor," http://seedit.sourceforge.net.

[11] E. Reshetova and J. erik Ekberg, "Mandatory Access Control for Mobile Devices," 2008.

[12] "Arm," http://www.arm.com/products/processors/classic/arm9/arm926.php.

[13] T. Alves and D. Felton, "TrustZone: Integrated hardware and software security," ARM, Tech. Rep., Jul. 2004.

[14] "Sonicsmx smart interconnect datasheet," Sonics, Tech. Rep.

[15] J. Castrillon, D. Zhang, T. Kempf, B. Vanthournout, R. Leupers, and G. Ascheid, "Task management in MPSoCs: an ASIP approach," in *ICCAD '09: Proceedings of the 2009 International Conference on Computer-Aided Design*. New York, NY, USA: ACM, 2009, pp. 587–594.

[16] S. Park, D. sun Hong, and S.-I. Chae, "A hardware operating system kernel for multi-processor systems," *IEICE Electronics Express*, vol. 5, no. 9, pp. 296–302, 2008.

[17] M. Lippett, "An IP Core Based Approach to the On-Chip Management of Heterogeneous SoCs," in *IPSOC '04: Proceedings of the 2004 IP Based SoC Design COnference & Exhibition*, 2004.

[18] A. C. Nácul, F. Regazzoni, and M. Lajolo, "Hardware scheduling support in SMP architectures," in *DATE '07: Proceedings of the conference on Design, automation and test in Europe*. San Jose, CA, USA: EDA Consortium, 2007, pp. 642–647.

[19] K. Pagiamtzis and A. Sheikholeslami, "Content-addressable memory (CAM) circuits and architectures: A tutorial and survey," *IEEE JOURNAL OF SOLID-STATE CIRCUITS*, vol. 41, no. 3, pp. 712–727, 2006.

[20] Alberto Ferrante, Giuseppe Piscopo, and Stefano Scaldaferri, "Application-driven Optimization of VLIW Architectures: a Hardware-Software Approach," in *Real-Time and Embedded Technology Applications*. San Francisco, CA, USA: IEEE Computer Society, 7 Mar. 2005, pp. 128–137.

[21] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "Cacti 5.1: Technical Report," Hewlett-Packard Development Company, L.P., Tech. Rep., 2008.