Ensuring Secure Program Execution in Multiprocessor Embedded Systems: A Case Study

Krutartha Patel, Sri Parameswaran, Seng Lin Shee
School of Computer Science and Engineering, The University of New South Wales, Sydney, Australia
{kpatel, sridevan, senglin}@cse.unsw.edu.au

ABSTRACT

Multiprocessor SoCs are increasingly deployed in embedded systems with little or no security features built in. *Code Injection* attacks are one of the most commonly encountered security threats. Most solutions to this problem in the single processor domain are purely software based and have high overheads. A few hardware solutions have been provided for the single processor case, which significantly reduce overheads. In this paper, for the first time, we propose a methodology addressing code injection attacks in a multiprocessor domain. A dedicated security *(monitor)* processor is used to oversee the application at runtime. Each processor communicates with the monitor processor through a FIFO queue, and is continuously checked.

Static analysis of program map and timing profile are used to obtain program information at compile time, which is utilized by the monitor processor at runtime. This information is encrypted using a secure key and stored in the monitor processor. A copy of this secure key is built into the processor's hardware and is used for decryption by the monitor processor. Each basic block of the program is also instrumented with security information that uniquely identifies itself at runtime. The information from static analysis thus allows the monitor processor to supervise the proceedings on each processor at runtime.

Our approach uses a combination of hardware and software techniques to keep overheads to a minimum. We implemented our methodology on a commercial extensible processor (Xtensa LX). Our approach successfully detects the execution of injected code when tested on a JPEG multiprocessor benchmark. The results show a small increase of 6.6% in application processors' runtime (clock cycle count) and 35.2% in code size for the JPEG encoder benchmark.

Categories and Subject Descriptors

B.8.1 [Performance and Reliability]: Reliability, Testing and Fault-Tolerance

General Terms

Security, Measurement, Design

Keywords

Code Injection Attacks, Multiprocessors, Tensilica, Security, Embedded System Processors

CODES+ISSS'07, September 30–October 3, 2007, Salzburg, Austria.

1. INTRODUCTION

The design of embedded systems with multiprocessor system on chip architecture (MPSoC) has gained a lot of momentum in recent times. The miniaturization of transistors has allowed chip manufacturers to increase chip density and incorporate multiple cores. Thus recent portable devices like PDAs, cellphones, gaming consoles, music players *etc.* tend to shrink in size while increase in functionality [32]. MPSoCs are seen as the pathway to taking the communications, multimedia and networking products to the next level [31].

The complexity of a multiprocessor system along with constraints of power and area make implementation of security features a challenge for embedded system designers [25]. Some of the most common devices today like cell phones and PDAs are starting to employ MPSoC architectures. With the ability of cellphones and PDAs to contain private and confidential information, the security of such systems is of utmost importance. Security is the largest concern that is still holding back some users of mobile technology from welcoming the use of next-generation mobile features such as m-commerce, and secure messaging [2, 17]. Extensive research available for securing for general purpose computers does not scale well to embedded systems. This is because embedded systems are a lot more resource constrained in terms of their size restrictions, processing capabilities and energy consumption [25, 17]. Hence there is a definite need for making MPSoC architectures secure, to encourage the use of embedded systems for secure applications.

In the single processor domain, *code injection* attacks are the most predominant. A publication by CERT highlighted that on average 47% of vulnerabilities reported from 1994-2004 were associated with buffer overflows, a type of code injection attack [27, 28]. This highlights that along with the theoretical improvements in cryptography and security protocols, *secure implementation* is important, because security attacks take advantage of weaknesses in system implementations [9]. One of the most often exploited weakness in the C programming language is an out-of-bound array access that causes a buffer overflow. Because there are no built in checks in C, even experienced programmers can easily write code with bugs in it. As shown in [22] the SPECINT95 *compress*, *go* and *ijpeg* benchmark programs contained several bugs in the form of out-of-bounds array accesses.

The intent of the attacker is to dynamically alter or insert instructions in the program. This modification of instructions violates the code integrity and hence disrupts the program flow causing erroneous program behavior. The code injection attacks come in the form of stack and heap based buffer overflows, dangling pointer references, format string vulnerabilities and integer errors. A detailed explanation of these attacks for single processors can be found in [33, 34].

The threat of code injection attacks for MPSoC architectures is far greater than for single processor systems. We identify two distinct ways in which code injection attacks are a threat to MPSoC architectures. The first threat involves one of the processors cor-

rupting sections of the instruction memory that is used by other processor(s). The other way is by corrupting the communications between two processors. Communication between two processors generally allows shared memory locations or inter-processor communication buffers where data can be written to or read from. If the communicated data from the producer processor is corrupted before the consumer processor reads it, the instructions reliant on the read data would result in an invalid result. It is not hard to foresee the above mentioned threats given the fact that similar threats exist in general purpose computing systems [35]. Bus communications can be tampered with, e.g., *modchips* have previously been known to tamper a bus on Sony Playstation and Microsoft Xbox gaming consoles such that illegal games may be played [3, 18]. Thus, ensuring the integrity of the executed instructions in an MPSoC environment becomes an extremely important task at runtime.

One of the best ways to ensure secure execution of programs is to integrate security features in the hardware design of embedded processors. The distinct advantage of incorporating security features at the processor level is that there is a lot less overhead compared to handling security at a higher level. An approach that allows security features implemented at reduced overhead is a boon for energy stringent embedded systems. Another challenge in keeping the overheads low is the amount of access the implementation platform allows.

Extensible processors are gaining popularity in the industry as the existing base instruction set architecture (ISA) allows rapid development of multiprocessor applications [1, 4]. These being commercial processors, the user is unable to change the way the base instructions function (i.e., it is not possible to change the microinstructions of those base instructions to add security functions as was done in [24]. Hence the users are limited in the amount of direct information they are able to obtain (e.g., access to special registers like PC, IR, access to micro instructions of the base instruction set etc.). Nevertheless, it is important to come up with a scheme to implement security on these commercial platforms in order to understand the applicability of the research to platforms with limited modifiability.

In this paper, we propose a novel approach for adding security to MPSoc systems. We employ a dedicated security processor to monitor a multiprocessor application on an MPSoC architecture, built using Tensilica's Xtensa processing system. We show that we are able to implement security features on an MPSoC with a small percentage of runtime and area overheads. To the best of our knowledge this is the first time that a comprehensive approach addressing code injection attacks in programs is presented for an MPSoC architecture on an extensible commercial processor.

The remainder of the paper is organized as follows. A summary of related work is presented in Section 2. Section 3 describes the proposed architecture of an MPSoC. Section 4 describes the systematic methodology for equipping a given application with security features. The analysis on the tests and the results obtained from the case study on JPEG encoder benchmark are in Section 5 and the paper is concluded in Section 7.

2. RELATED WORK

The security threats to program from software attacks like code injection attacks have not been examined before on an MPSoC architecture. Arora et al. mapped the code and data of a commercial security processing library onto a multiprocessor SoC to achieve a speed up in cryptographic operations [8]. Zhang et al. proposes a shared bus encryption protocol for multiprocessors [35]. To the best of our knowledge however, there is no work on detecting software attacks in the multiprocessor domain. In this section, we compare our work with similar work in the single processor domain. We evaluate and discuss the feasibility of these approaches for an MPSoC where appropriate.

The techniques to counter code injection attacks can be broadly classified into software and hardware based. Software based techniques can either be static or dynamic which try to detect vulnerabilities in the code at compile time and runtime respectively. Hardware based techniques require architectural support to detect code injection attacks.

A number of researchers in [13, 19, 29] propose tools and techniques for static analysis of code that allows detection of buffer overflow which is a type of code injection attack. The problem with static analysis is that they raise a number of false positives. Also these approaches of static analysis would not scale well for MPSoCs because in MPSoCs, there is a possibility of memory corruption at runtime. Hence a technique that allows dynamic checking would be more suited to detection of code injection for multiprocessors. The dynamic checking methods use software constructs to monitor program behavior at runtime as used in Stack Guard[12]. Stack Guard specifically targets buffer overflow attacks only and may not work for other code injection attacks.

CCured used a dynamic checking approach to make C programs type safe [22]. It analyses a given C program at static time inferring the pointers that are statically safe and others that need to be checked at runtime. The problem with this approach is that the runtime checking of these vulnerable pointers causes a performance degradation of up to 150% in runtime. Arora et al. implemented the CCured framework by architectural modification to do runtime checking in hardware [7]. This allowed a speed up of up to 4.6% compared to the situation where the checking is done purely in software. The overall runtime for single processor benchmarks is still quite significant making the CCured approach hard to scale for MPSoC architecture.

The work by Zhang et al. in [35] highlights concerns that exists in symmetric shared multiprocessor environments (high performance multiprocessor servers). Zhang et al. indicate that memory corruption, memory-to-cache and also cache-to-cache communication via buses are threats that exist in multiprocessor environments.

Most of the hardware assisted techniques tend to be *attack specific* [12, 20] or concentrate on tamper resistance [14] or cryptography [14]. Attack specific techniques generally tend to only cover buffer overflow attacks which are a subset of code injection attacks. From a multiprocessor point of view, for the attack specific hardware technique, depending on what features in hardware are required, they may or may not be implementable in extensible processors. If they are implementable, unless the hardware can be shared among all the processors, each processor on an MPSoC might need its own hardware to be able to perform its security functions at runtime. This would significantly increase the area overhead on the processor.

Various obfuscation techniques including instruction set randomization have been proposed in [10, 16] to counter code injection attacks. These techniques rely on a randomized processor that encrypts (at compile time) and decrypts (at runtime) the instructions using a randomized key. An attack would cause a runtime exception and would not allow the injected code to execute. Instruction set randomization incurs a high amount of overhead and may also require processor modification. The overhead is mainly due to the decryption at runtime for each instruction. We also employ encryption and decryption to specific fields of our special instructions using a secure key stored in hardware. Hence our overhead from decryption at runtime is much lower. The storage of the key in hardware ensures that it secured from software attacks.

The use of non-executable stacks and heaps is another approach that has been suggested in [5, 6] to prevent code injection attacks. Non-executable stacks and heaps generally require operating systems modification or extra features in hardware to be added. This can prevent a great variety of code-injection attacks. However some programs actually require executable stacks or heaps for correct functionality. For example in a Java Virtual Machine (JVM), code is generated at runtime from byte-code and placed on to the heap.

Milenkovic et al. in [21] and Ragel et al. [24] propose using signatures for validating each basic block in the program. Ragel

et al. use a hardware based approach by modifying microinstructions. This approach generally requires significant modification (large number of instructions have to be instrumented) to the architecture that is only possible in some extensible processors. The hardware-software technique described in [9] requires additional co-processors and hardware tables for runtime monitoring. Their method produces code like ours, which is not easily relocatable because the hardware monitor has to be configured every time a different application executes on the embedded system. In addition it also needs significant modification to the architecture to be able to extract the properties of the code at runtime which may not be accessible in all extensible processors.

Oyama et al. in [23] and Chew et al. in [11] propose a method of ensuring secure *system call* execution. Our method is not designed to protect against corrupt system calls. However our system will be alerted if the execution time of a particular *basic block* of code is outside the range predicted by the profiler.

2.1 Contributions

The contributions of this paper are as follows:

1. For the first time we propose a hardware/software based approach to address code injection attacks in multiprocessor applications.
2. We demonstrate how this structure can be implemented with the proposed techniques using *Xtensa LX*, a commercial extensible processor design system from Tensilica Inc. using a multiprocessor embedded application (JPEG encoder).
3. In case of code injection, we are able to detect the exact basic block in which the attack occurred in a particular processor.

2.2 Limitations

The limitations of our approach are as follows:

1. Our approach does not monitor system calls and hence attacks via malicious system calls cannot be detected and prevented. However methods proposed in [15, 30] may be used in conjunction with our method for a stronger security implementation.
2. Our approach does not detect data corruption. For example if an attacker wants to simply corrupt the immediate value in an instruction, it would not be detected by our approach.
3. During static analysis, some of the sections of the code may be left unprofiled as the input data values determine the path of the program. But this is a common problem that embedded systems designers face, i.e. finding a data set that covers all the cases during the testing phase of a design. One of the approaches to solving this problem could be to use a data set that would cover most of the code. Then, manually estimating the *min* and *max* time values for those blocks. Also every time there are code changes, it must be re-profiled to get the control flow map and *min* and *max* times.

3. SYSTEM ARCHITECTURE

We use Xtensa LX, an extensible commercial microprocessor core generated by the toolset from Tensilica Inc. It allows the designers to configure each processor core by adding features on top of the base architecture making each processor core a superset of the baseline processor. These cores can then be synthesized. The base architecture contains up to 64 general purpose registers, 6 special purpose registers and 80 instructions including compact 16 and 24 bit RISC instruction encoding [26].

Xtensa LX allows extension to the processor through Tensilica Instruction Extension (TIE) language which is similar to VHDL. It supports new instructions and registers, execution units and I/O ports.

Because this is a multiprocessor application, the key feature that was employed for inter-processor communication was a FIFO queue interface. This feature in the Xtensa processor supports external communications at a much wider bandwidth than existing interconnects. The queue interfaces on the processors can be described using TIE language and can be used to push data to an outgoing queue and pop data from an incoming queue. The logic to stall the processor when writing to a full queue and reading from an empty queue is generated automatically via the Xtensa toolset.

Although we use the Xtensa LX processor to implement our security solution, the solution is scalable to other processors. As long as we can build a multiprocessor application and have some way of communicating between processors, for example using FIFOs, our solution can be implemented on that processor.

4. SYSTEM DESIGN

In order to design a multiprocessor system we first needed a benchmark program. Since the research community does not have ready access to already partitioned benchmarks, we produced our own benchmark application based on a single processor benchmark. A freeware compression algorithm JPEG, was chosen for this purpose. The benchmark program, available as a C program was manually partitioned into various pipelines/data flow stages, adhering to the JPEG standard. The partitioned stages were then mapped to individual processors in Xtensa LX. The JPEG encoder application partitioned into six simultaneous tasks that ran on six different processors. Thus a multiprocessor benchmark of JPEG was obtained.

In this pipeline of processors, we allow the first processor to execute the first part of the JPEG algorithm (say DCT), while the second processor works on the quantization of the previous frame and so forth. Such a pipeline increases performance without unduly increasing the area of the overall processor, since each of the pipelined processor can be customized for the reduced functionality.

A multiprocessor system was designed for the *JPEG* benchmark. The configurable features used in addition to the base ISA features for each application are shown in Table 1.

Features	JPEG
Speed	533 MHz
Process	90nm GT
Pipeline length	5
User Defined Registers	4
Size	63,843 gates
Core Size	0.32 mm^2
Core Power	74.35 mW
MUL16/MUL32	$\surd$
MIN/MAX/MINU/MAXU	$\surd$
Enable Density Instructions	$\surd$
Enable Boolean Registers	$\surd$
TIE Arbitrary Byte Enables	$\surd$
Enable TIE Wide Stores	$\surd$
Max Instruction Width	8 bytes
PIF Interface Width	128 bits

Table 1: Extending the base ISA of Xtensa LX processor for JPEG benchmark

4.1 Overview of Proposed Approach

The overall system block diagram of the architecture designed to implement security features for the multiprocessor applications is shown in Figure 1. It shows a multiprocessor application with an N-processor configuration. The communication between two processors is performed using a FIFO queue. For simplicity in Figure 1, FIFO communication is shown between processors P_k and P_{k+1}, $1 \leq k \leq (N-1)$ but there could be FIFO communication between any two processors P_a and P_b, $1 \leq a \leq N$ and $1 \leq b \leq N$, depending on the requirements of the application.

In a multiprocessor system of N processors, the required connections between the processors are made through Xtensa's modeling protocol called XTMP. An extra security processor called the **monitor processor** is used to monitor all the individual processors of the multiprocessor system. Each of the individual processors communicates through a FIFO queue structure to the monitor

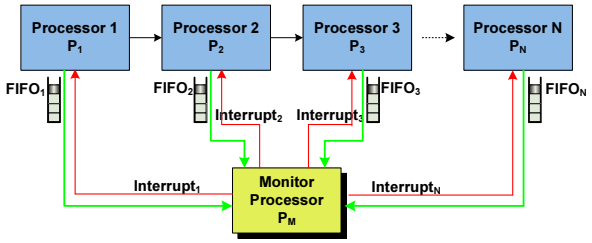

Figure 1: A layout of the multiprocessor system designed using Xtensa Modelling Protocol (XTMP).

processor. The monitor processor communicates with each of the individual processors through an external signal that evokes an interrupt in the individual processor.

The FIFOs that are used for communication between the processors P_1 and P_N are implemented using the FIFOs generated from the Xtensa toolset. But the FIFOs that are used for communication by each application processor with the monitor processor use a special FIFO shown in Figure 2. This FIFO has a *Timestamp Unit* that uses the clock cycle count (CCOUNT) register of its respective processor. Thus each entry into the FIFO is timestamped through the *Timestamp Unit* for use by the monitor processor. The signals *full* and *empty* are used for checking the queue before writing and reading respectively.

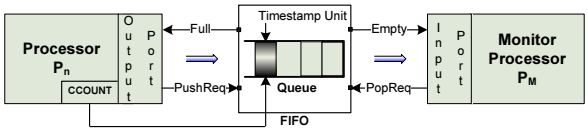

Figure 2: FIFO implementation an application processor and the monitor processor using XTMP.

Figure 3 gives a detailed overview of the hardware as well as the software flow for the implementation of the design which are further discussed in subsections 4.2 and 4.3.

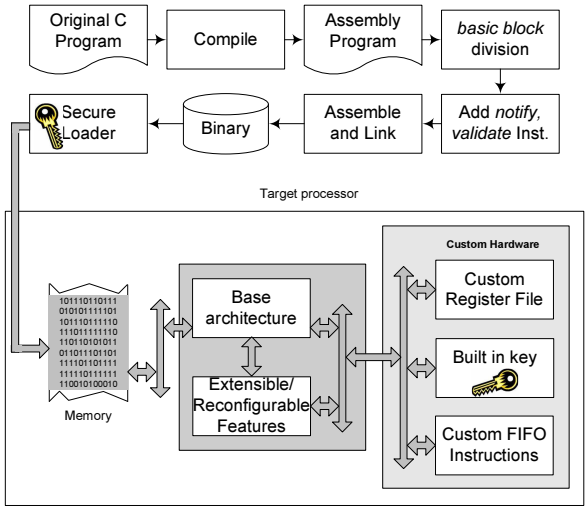

Figure 3: The hardware and software design flow of the proposed design.

4.2 Software Flow

The assembly code of each of the processors of the multiprocessor system is divided into *basic blocks*. We define a *basic block* as a collection of instructions which does not cause a change in the normal sequential flow of the program. The normal flow of the program is one instruction after another as they are arranged in the instruction memory. But there are a few instructions which can change the control flow of the program. A branch, jump, loop or a call instruction may change the control of the program such that the next executed instruction is not from the next address in instruction memory but from address(es) which are above or below the current address of the program counter (PC). Our definition of the basic block is in Figure 4(a).

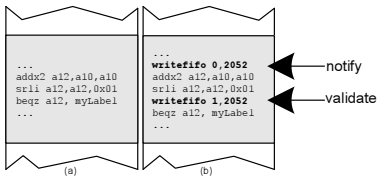

Figure 4: (a) Segment of assembly code in an Xtensa LX processor. (b) A *basic block* after software instrumentation

Software instrumentation is performed at compile time on the assembly file designed to run on each of the processors in a multiprocessor application. Each processor core is allocated a unique processor id *pId*. The assembly file is then broken down into basic blocks, and each basic block is instrumented with two FIFO instructions. The FIFO instruction called **writefifo** was defined in TIE language. The instrumented basic block is shown in Figure 4(b). The **writefifo** instructions at the start and end of the basic block are also referred to as notify and validate instructions respectively.

Each basic block in a particular processor is allocated a unique *blockId*. Thus, the **writefifo** instruction with which each basic block is instrumented contains three important fields. The first field is the interface code, the notify instruction has an interface code of 0, and the validate instruction has an interface code of 1. The number after the interface code is a combination of the *pId* and *blockId* which we call *pId_bId*. The *pId_bId* number for a particular block would be the same and hence it can be seen that both the **writefifo** instructions have the same number in the *pId_bId* field. The final field is for the timestamp which is placed at runtime when the instruction is executed. The timestamp on the outgoing FIFO instruction is the value of the cycle count register **CCOUNT** of the respective processor at runtime.

Hence at runtime, when the first FIFO instruction of a *basic block* is executed, it registers its *pId* and *blockId* with the monitor processor. The purpose of the first FIFO instruction is to inform the monitor processor that a particular processor identified by *pId* is about to execute block *bId*. The purpose of the second FIFO instruction at the end of the *basic block* is to again inform the monitor processor that a particular processor *pId* has finished executing the block.

The second part of the instrumentation involves static analysis of the assembly code. By analyzing the assembly code, we generate a map of all the possible paths that the program may take at runtime. The map is generated using the *bId* of the blocks that the task on each processor is divided into. These program maps are later loaded onto the monitor processor.

The application is then run without the monitor monitoring it for timing requirements and a trace file of the execution is generated. Using the trace file produced for each processor, we are able to find the time taken by each instruction that was executed. Adding up the execution time of each instruction in a particular block, we get the execution time of that block. It is likely that some blocks have been executed more than once and that their execution time has a range of values. The cache in the architecture also introduces variability in timing depending on whether or not the instruction was in the cache or had to be fetched from the memory.

It may also be possible that the execution path of the program do not include all possible sections of the code. The timing information for those blocks of code would therefore be unavailable through the tracefile analysis. It is likely that these sections of the code may not be used much except in corner cases. Thus another tool that estimates the time for these blocks is used. This tool estimates how much time each instruction in the block may take based

on the instruction set simulator's ge neral guide. A history of similar instructions can also be looked at in the tracefile to get a rough estimate on the time for the instruction. Once each instruction's time is estimated in this unexecuted block, we can sum up these estimated values and get the estimated execution time for the block. The minimum and the maximum execution time of all the blocks are recorded and stored in the monitor processor.

The final part of the instrumentation process is the act of the secure loader. We assume that we have a secure loader that allows encryption using a unique secure key that is secretly known to the loader. The secure loader must only encrypt the *pId_bId* field of the **writefifo** instruction before it loads the program in the instruction memory of Xtensa LX.

Thus the monitor processor contains two pieces of vital information about the system. One of them being the program map and the other is the execution times of each block in each processor of the multiprocessor system. With this information and an algorithm for monitoring the processors in the system, run time security checks are performed on the system.

4.3 Architectural Framework

Figure 3 shows that the base architecture of the processor is extended using the extensible options available in Tensilica. The extended features for the JPEG benchmark are shown in Table 1. The processor was further extended by defining custom hardware. For example a small register file consisting of four registers, the secure key used by the monitor processor and also the FIFO instructions for communications between processors are defined in Xtensa's TIE language.

4.4 Runtime Security Checks

The monitor processor has two important tasks in terms of monitoring the security of the processors in the multiprocessor application. Firstly, ensuring that the execution of each of the processors follows a valid path by comparing with the program map. Secondly, it checks that the time taken by particular blocks of code are within the time limits obtained from the tracefile. The algorithm used by the monitor processor for doing security checks at runtime is shown in Algorithm 1.

Algorithm 1 The algorithm used by the monitor processor for security checks at runtime.

```
Initialise Last Processor P_k = Unfinished
while (P_k = Unfinished) do
   Read from incoming FIFO of processor P_j
   if read numbers from the FIFO then
      Unencrypt read numbers to code, pId, bId, time
      if (code = 0) then
         Check Path Validity
      else if (code = 1) then
         Check Time Validity
      else if (code = 3) then
         Check if P_k has finished
      end if
   end if
end while
```

If the execution time of a block of code is violated then the monitor processor sends an interrupt signal which causes all the processors to stop their execution and the multiprocessor application to exit.

It should be noted that the monitor processor needs to decrypt the map information, the minimum and maximum blocktime information and also the *pId_bId* information using the secure key in hardware. Thus the runtime checking involves both the hardware as well as the software phases of the design. There is obviously going to be extra overhead caused by decryption of the above mentioned items. The decryption engine being in hardware helps in reducing the runtime overhead for the application.

The encryption at compile time and decryption at runtime is an extra security measure. An attacker implementing a code injection attack, knowing how the basic blocks are instrumented may make educated guesses about the *pId_bId* number in the **writefifo** instructions. Encrypting the *pId_bId* field gives us another dimension of catching a code injection attack. Without knowing the key, it is highly unlikely that the *pId_bId* field in the **writefifo** instruction matches at runtime. The key being stored in hardware is not accessible without physical or side-channel attacks, both of which require physical access to the device and highly sophisticated equipment.

5. EXPERIMENTAL TESTS AND RESULTS

A multiprocessor system was designed for a case study on a multiprocessor benchmark JPEG with 7 processors including the monitor processor.

5.1 Performance Analysis

In order to test whether the system was able to detect security violations, the following types of tests were designed.

A. **Inserting/Adding** instruction(s) into an existing block of code
B. **Modifying** the *pId_bId* field of the *notify* or *validate* instruction
C. **Changing** the *notify* instruction to a *validate* instruction or vice-versa
D. **Modifying** a *branch/jump* instruction to go to a block of inserted malicious code

Type **A** tests were designed to test situations where there are some malicious instructions added in an existing block of code in order to corrupt the program. Type **B** instruction is a stepping stone for achieving a successful attack of type **D**. Type **C** attacks try to extend a block by changing a *notify* instruction to a *validate* instruction. The type **D** tests were designed to test a situation where a return address register could be overwritten to jump to some inserted malicious code. Hence by overwriting a *branch/jump* instruction to point to a foreign code, we are trying to test whether the security features are able to detect execution of an entire block of malicious code which is what the attackers are usually interested in.

Table 2 shows the results of the attacks carried out under each of the four categories. The term $\mathbf{BL}_O$ refers to the original block length and $\mathbf{BL}_N$ refers to the new block length after the security violation. Our approach is able to detect all the four kinds of attacks. These four categories of attacks test the timing constraints, encryption of *pId_bId*, program map as well as the runtime algorithm in the monitor processor for security checking.

Test Type	$\mathbf{BL}_O$	Security Violation	$\mathbf{BL}_N$	JPEG	
				Detected	Detector
A	10	Inserted *xor*	11	√	Timer
B	7	change *pId_bId* in *notify*	7	√	Algorithm/ Encryption
C	5	changed *notify* to *validate*	5	√	Algorithm
D	0	Inserted a block	7	√	Timer/Mapper

Table 2: Results from security violations on JPEG multiprocessor benchmark

Table 3 details the breakdown of the increase in code size on each of the six processors used for the JPEG benchmark. There was an overall increase of 6.6% in application processors' runtime and 35.2% in the code size of the JPEG encoder benchmark. The increase in runtime is due to the communication instructions from each processor to the monitor processor.

The graph in Figure 5 shows the overhead values for 50 different frames (used as an input to the JPEG encoder) from five different benchmarks. It can be seen that the overhead is fairly constant at about 6.6% for all the benchmarks but *flowg*. This is expected because the *flowg* benchmark is computationally very expensive. It

Bench-mark	Original System		Secure System		% Increase	
	Code Len.	Runtime ($\times 10^6 cc$)	Code Len.	Runtime ($\times 10^6 cc$)	Code Len.	Run time
JPEG	1313	4.29	1775	4.58	35.2	6.6
P_1	440	N/A	587	N/A	33.4	N/A
P_2	56	N/A	81	N/A	44.6	N/A
P_3	171	N/A	197	N/A	15.2	N/A
P_4	144	N/A	198	N/A	37.5	N/A
P_5	435	N/A	621	N/A	42.8	N/A
P_6	67	N/A	91	N/A	35.8	N/A

Table 3: Code Length and Execution time overheads for a multiprocessor JPEG benchmarks

uses the computationally expensive functions that the other benchmarks do not use. The execution path for *flowg* unlike other benchmarks includes some small basic blocks. If the basic blocks are small, the relative percentage overhead for the particular blocks is higher. Hence it can be noticed that generally the overhead stays constant at 6.6% when the data uses similar functions in the program. It is worth noting that even when we have a benchmark like *flowg*, which is computationally very expensive compared to the other benchmarks, the overhead rises to only 9.3%.

The overhead in our approach comes from communication between application processors and the monitor processor. The monitor processor has to clear all of the monitoring information which is being rapidly sent to itself. Future work will look at ways to reduce this traffic of packets.

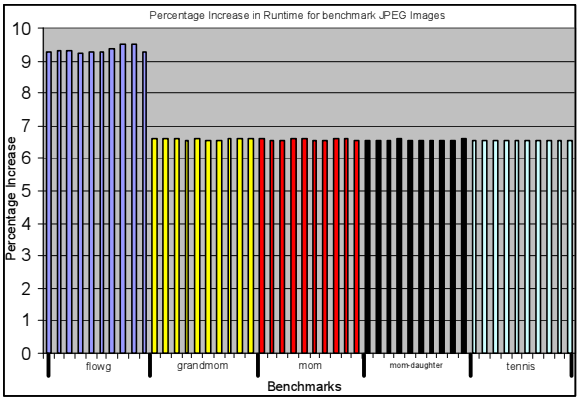

Figure 5: The distribution of runtime overhead for 50 different images used as an input to the jpeg encoder program.

5.2 Area Estimation

For the JPEG case study, the base system consists of six processors with only the FIFO connections between the processors as the custom hardware. After the security measures for secure execution are implemented, there is an additional (monitor) processor. Moreover, a custom FIFO using TIE instructions is added to each processor to communicate with the monitor processor. Overall this results in an area increase of 10.7% to $4.186mm^2$ architecture. This is an acceptable overhead given that a whole new processor is added which is entirely dedicated to monitoring the security of the JPEG encoder application.

6. ACKNOWLEDGEMENTS

The authors would like to thank Jorgen Peddersen, Jeremy Chan, Jude Ambrose and Roshan Ragel for their help with this work.

7. CONCLUSIONS

In this paper, we presented a novel architecture for allowing secure execution of multiprocessor programs. We combine the forces of program map, profiling analysis and encryption to come up with a strong design for detecting code injection attacks. We applied our methodology on a case study of a JPEG encoder. Our hardware/software approach for implementing security resulted in an acceptable overhead of 6.6% in the applications' runtime and an area overhead of 10.7%. The validation of the approach on inducing the system's response by conducting security attacks showed promising results. The system managed to detect most of the attacks by correctly identifying the processor Id and the block Id of the attacked processor.

8. REFERENCES

[1] ARC the leader in configurable processor technology. ARC International (http://www.arc.com).
[2] epaynews - mobile commerce statistics. (http://www.epaynews.com/statistics/mcommstats.html).
[3] What is a modchip? ModChip (http://www.modchip.com).
[4] Xtensa Processor. Tensilica Inc. (http://www.tensilica.com).
[5] Homepage of the pax team. PaX Team (http://pax.grsecurity.net/), 2007.
[6] Linux: Exec shield overflow protection. PaX Team (http://kerneltrap-.org/node/644), 2007.
[7] D. Arora, A. Raghunathan, S. Ravi, and N. K. Jha. Architectural support for safe software execution on embedded processors. In *CODES+ISSS '06*, pages 106–111, New York, NY, USA, 2006. ACM Press.
[8] D. Arora, A. Raghunathan, S. Ravi, M. Sankaradass, N. K. Jha, and S. T. Chakradhar. Software architecture exploration for high-performance security processing on a multiprocessor mobile soc. In *DAC '06*, pages 496–501, New York, NY, USA, 2006. ACM Press.
[9] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. Secure embedded processing through hardware-assisted run-time monitoring. In *DATE '05*, pages 178–183, Washington, DC, USA, 2005. IEEE Computer Society.
[10] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *CCS '03*, pages 281–289, New York, NY, USA, 2003. ACM Press.
[11] M. Chew and D. Song. Mitigating buffer overflows by operating system randomization. Technical Report CMU-CS-02-197, Department of Computer Science, Carnegie Mellon University, Dec. 2002.
[12] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, jan 1998.
[13] N. Dor, M. Rodeh, and M. Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. In *PLDI '03*, pages 155–167, New York, NY, USA, 2003. ACM Press.
[14] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the ibm 4758 secure coprocessor. *Computer*, 34(10):57–66, 2001.
[15] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.
[16] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *CCS '03*, pages 272–280, New York, NY, USA, 2003. ACM Press.
[17] P. Kocher, R. Lee, G. McGraw, and A. Raghunathan. Security as a new dimension in embedded system design. In *DAC '04*, pages 753–760, New York, NY, USA, 2004. ACM Press. Moderator-Srivaths Ravi.
[18] M. G. Kuhn. Cipher instruction search attack on the bus-encryption security microcontroller ds5002fp. *IEEE Transactions on Computers*, 47(10):1153–1157, 1998.
[19] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. pages 177–190, 2001.
[20] J. Mcgregor, D. Karig, Z. Shi, and R. Lee. A processor architecture defense against buffer overflow attacks. pages 243–250, 2003.
[21] M. Milenkovic, A. Milenkovic, and E. Jovanov. Hardware support for code integrity in embedded processors. In *CASES '05*, pages 55–65, New York, NY, USA, 2005. ACM Press.
[22] G. C. Necula, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy code. In *POPL '02*, pages 128–139, New York, NY, USA, 2002. ACM Press.
[23] Y. Oyama and A. Yonezawa. Prevention of code-injection attacks by encrypting system call arguments. Technical Report TR06-01, Department of Computer Science, The University of Tokyo, March 2006.
[24] R. G. Ragel and S. Parameswaran. Impres: integrated monitoring for processor reliability and security. In *DAC '06*, pages 502–505, New York, NY, USA, 2006. ACM Press.
[25] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady. Security in embedded systems: Design challenges. *ACM Trans. Embedded Comput. Syst.*, 3(3):461–491, 2004.
[26] C. Rowen and D. Maydan. Automated processor generation for system-on-chip. Technical report, Sept 2001.
[27] N. Tuck, B. Calder, and G. Varghese. Hardware and binary modification support for code pointer protection from buffer overflow. In *MICRO 37*, pages 209–220, Washington, DC, USA, 2004. IEEE Computer Society.
[28] US-CERT. US-CERT Vulnerability Notes Database. (http://www.kb.cert.org-/vuls), 2007.
[29] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.
[30] C. Warrender, S. Forrest, and B. A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.
[31] W. Wolf. The future of multiprocessor systems-on-chips. In *DAC '04*, pages 681–685, New York, NY, USA, 2004. ACM Press.
[32] W. Wolf. Multimedia applications of multiprocessor systems-on-chips. In *DATE '05*, pages 86–89, Washington, DC, USA, 2005. IEEE Computer Society.
[33] Y. Younan, W. Joosen, and F. Piessens. Code injection in C and C++: A survey of vulnerabilities and countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2004.
[34] Y. Younan, W. Joosen, and F. Piessens. A methodology for designing countermeasures against current and future code injection attacks. In *IWIA '05*, pages 3–20, Washington, DC, USA, 2005. IEEE Computer Society.
[35] Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta. Senss: Security enhancement to symmetric shared memory multiprocessors. In *HPCA '05*, pages 352–362, Washington, DC, USA, 2005. IEEE Computer Society.