LOCS: A Low Overhead Profiler-Driven Design Flow for Security of MPSoCs

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

ABSTRACT

Security is a growing concern in processor based systems and hence requires immediate attention. New paradigms in the design of MPSoCs must be found, with security as one of the primary objectives. Software attacks like *Code Injection Attacks* exploit vulnerabilities in "trusted" code. Previous countermeasures addressing code injection attacks in MPSoCs have significant performance overheads and do not check every single line of code. The work described in this paper has reduced performance overhead and ensures that all the lines in the program code are checked.

We propose an MPSoC system where one processor (which we call a MONITOR processor) is responsible for supervising all other application processors. Our design flow, LOCS, instruments and profiles the execution of basic blocks in the program. LOCS subsequently uses the profiler output to re-instrument the source files to minimize runtime overheads. LOCS also aids in the design of hardware customizations required by the MONITOR. At runtime, the MONITOR checks the validity of the control flow transitions and the execution time of basic blocks.

We implemented our system on a commercial extensible processor, Xtensa LX2, and tested it on three multimedia benchmarks. The experiments show that our system has the worst-case performance degradation of about 24% and an area overhead of approximately 40%. LOCS has smaller performance, area and code size overheads than all previous code injection countermeasures for MPSoCs.

Categories and Subject Descriptors

B.8.2 [Performance and Reliability]: Performance Analysis and Design Aids

General Terms

Design, Measurement, Performance, Security

Keywords

Architecture, Code Injection, Execution Profile, MPSoC, Tensilica

1. INTRODUCTION

The increasing complexity and functionality of designs in embedded systems necessitates the exploration of new design paradigms with multiple processors on a single chip. Consumer devices such as cellular handsets already employ dual processor chips (DSP and RISC) [6]. Future multimedia devices will embed many processors on a single chip to tackle pipelining and parallelism, required to support a range of functionalities in a single embedded system [6, 20].

With the leap ahead in technology and newer design models, security attacks are imminent. Security is usually an afterthought in the design of general purpose embedded systems and single processor system on chips; and hence we see a wide range of attacks which systems today cannot confront [15]. We believe that security should be one of the design objectives and in this paper we show how designers can incorporate that when designing MPSoCs for multimedia applications.

Software attacks (and particularly code injection attacks) are the most commonly encountered attacks in embedded systems [13]. The US-CERT vulnerability reports show that, on average, nearly 11% of all vulnerabilities reported in 2007 referred to buffer overflow attacks, which is only one of the ways in which to induce a code injection attack. The statistics for the percentage of vulnerabilities pertaining to buffer overflow attacks reported in 2007 is shown in Figure 1. A variety of other common software attacks are discussed in [12, 19, 22].

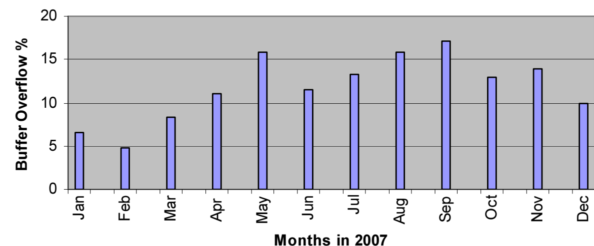


Figure 1: Buffer Overflow Vulnerabilities from US-CERT

Code injection attacks pose significant risk in multiprocessor systems as they are inherently more complicated than single processor systems. Security always adds overheads but it is imperative that security measures are included as part of the design process in MPSoCs with a goal to minimize overheads [15].

In this paper, we propose an MPSoC with an additional processor (MONITOR processor) for security. Our system is briefly described here. The basic blocks in each of the application processors are instrumented to allow checking at runtime by the MONITOR. *A key feature of our design flow LOCS is that the use of simulation trace analyzer and execution profiler tool allows the MPSoC designer to adjust the frequency of how often the security checks happen for each basic block.* The MONITOR has a record of execution times (a min and max time is given for each basic block - to allow for cache misses) for each basic block in each of the application processors. In addition, it also contains the control flow map of the programs in the application processors. Thus, if the application program takes more time than it should, or if it takes a path in the program which was not intended, then an interrupt is raised from the MONITOR which alerts the application processors.

The remainder of the paper is organized as follows. A summary of related work is in Section 2. Section 3 describes the methodology our design flow LOCS employs to secure an MPSoC. The software and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES+ISSS'08, October 19–24, 2008, Atlanta, Georgia, USA.
Copyright 2008 ACM 978-1-60558-470-6/08/10 ...$5.00.

hardware design flows are described in Section 4. Section 5 presents the results followed by discussion of key issues in Section 6. Finally the paper is concluded in Section 7.

2. RELATED WORK

In this section, we describe the existing countermeasures for MPSoCs and also evaluate the scalability issues of some of the countermeasures from the single processor domain to the multiprocessor domain.

Software based countermeasures can be classified into either static or dynamic techniques. Static techniques try to eradicate the vulnerabilities in the code during compilation while the dynamic techniques detect attacks at runtime. A number of static analysis techniques have been proposed in [2, 3, 4, 19].

A dynamic code analysis tool proposed in [5] helps in protection against invalid array accesses. CCured proposed in [9] uses both static and dynamic analysis for checking that the pointers are safe and would not cause memory errors which could potentially be used for code injection attacks.

In terms of hardware based countermeasures, Milenkovic et al. in [8] proposed a method to fetch instructions from the memory using a signature verification unit. Ragel et al. in [14] also proposed a method for basic block validation using microinstructions.

The hardware techniques in [1] proposed architectural modifications to detect code injection attacks via runtime monitoring. Another hardware based method in [7] proposes to use a separate secure return address stack (SRAS) to prevent buffer overflow attacks.

Static analysis tools like Stack Guard in [2], only aim to solve buffer overflow problems, and may not scale for other types of code injection attacks. Static analysis may also require the code to be written in a specific manner causing issues of portability. Static analyzers by themselves are not enough as they do not protect against attacks that occur at runtime.

The dynamic code analysis tools proposed in [5] and [9] incur runtime overheads of up to 220% and up to 150% respectively. The extremely high runtime overhead for [5, 9] makes them difficult to scale for multiprocessor applications.

A lot of the solutions proposed using hardware assisted techniques require major hardware modifications which are not possible in all commercial processors such as Tensilica's Xtensa LX2. Xtensa LX2 can be extended only to a certain extent using Tensilica's Instruction Extension (TIE) language which allows the users to define custom hardware. The approach in [8] requires architecture modification to allow interception of fetched instructions while the approach in [14] requires modification of the microinstructions. Both the features of interception of fetched instructions and modifications of microinstructions are unavailable in Xtensa LX2. The hardware techniques proposed in [1, 7] also face similar limitations of needing significant architectural modifications.

Thus the existing single processor hardware techniques cannot be directly applied for the MPSoC domain on commercial processors like Xtensa LX2 and similar where the processor hardware description is unavailable.

In the multiprocessor domain, the work in [11] uses a software based approach and the work in [10] uses a hardware based approach for detecting code injection attacks on MPSoCs. For the remainder of this paper, we refer the work in [11] as SW_MON and the work in [10] as SHIELD.

In contrast to the case study in SW_MON and in SHIELD, the work described in this paper is a profiler driven design flow for addressing security aspects of MPSoCs. Our approach in this paper monitors every single line of the application program code unlike the approaches in SW_MON and SHIELD where the control flow instructions were not monitored. In addition, our design flow also has half as much increase in code size compared to the approaches in SW_MON and SHIELD.

The profiler driven design flow in this paper provides the designer with an insight of the application's hot-spots at the basic block level. When designing for security, this provides the designer the flexibility to trade-off between performance and the granularity level of monitoring. The approaches in SW_MON and SHIELD did not have any profile information available to the designer and hence failed to provide the mechanisms for the trade-off discussed above. Finally our approach achieves the lowest performance and code-size overheads compared to the approaches in SW_MON and SHIELD whilst still being automated and just as easy to implement. Our approach in this paper has been thoroughly tested on three multimedia benchmarks whereas the ones in SW_MON and SHIELD were case studies.

2.1 Attack Model and Assumptions

We target code injection attacks that can take place on an MPSoC. Examples of such attacks include stack and heap based buffer overflows, spurious control flows within the program, and run-time code corruption. It must be noted that we are not trying to handle physical attacks and hence attacks such as erasure of data or instruction memory through physical access to the device are not covered by our technique.

We assume that each processor has its own separate instruction and data memories and that it is safe to use the system library functions. We also assume that our MONITOR is totally secure and cannot be attacked. This is a reasonable assumption given that the MONITOR only has a few instructions which initiate the monitoring hardware and these can be placed in a ROM.

2.2 Contributions

The contributions of this paper are as follows:

1. For the first time a systematic methodology is proposed that checks every single line of program code. This methodology provides a fully automated approach and an intelligent static analyzer to achieve an instrumented binary for the MPSoC architecture.
2. A novel profiler driven design flow is proposed for incorporating security measures in an MPSoC system. Our design flow shows the program hot-spots (at the basic block level), and suggests the designs with minimal performance overhead. Our novel methodology of using the program hot-spots information at the basic block level allows the designer to achieve a balance between performance and security.
3. This is the first time a methodology for detecting code injection attacks on MPSoCs has been tested with three multimedia multiprocessor benchmarks.

2.3 Limitations

The limitations of our approach are as follows:

1. The program trace profiler in our design flow, LOCS, calculates the minimum (min) and maximum (max) execution times for each basic block. However the min and max times need to be *estimated* for the basic blocks that do not fall on the execution path using the processor's instruction set architecture.
2. A very small number of basic blocks that are involved in interprocessor communication often violate their min or max times raising false alarms. The range of time these basic blocks take cannot be accurately determined and hence they are only checked for the correctness of their control flow.
3. For a particular basic block that represents a loop, LOCS performs the timing check on that basic block only when the loop execution finishes, whereas the approach in SHIELD performs a timing check on every iteration of the loop. The approach in SHIELD may detect the attack faster if it happens in the first few iterations, otherwise our design flow LOCS would detect it faster because the security processor in SHIELD would have a lot of communication backlog to process.
4. Our approach does not cover data corruption or bit flips in the data memory.

3. PROPOSED METHODOLOGY

The proposed methodology for designing a secure MPSoC consists of three main components, which are static analysis, simulation

and profiling, and profiler driven security tuning. Finally, the allowable program behavior in the presence of the MONITOR is given.

The architectural framework is shown in Figure 2 where a dedicated processor called MONITOR is employed, for supervision of the application processors. The application processors can be configured to execute an application program by themselves or even combine together in a pipelined fashion to execute multiprocessor applications. For example, the application processors can communicate amongst themselves using a FIFO buffer as shown in Figure 2. Each processor is assigned a unique id **pId**. Each application processor talks to the MONITOR using a special **TFIFO** queue that *timestamps* every entry into the queue. The use of the *timestamp* is discussed later in this section.

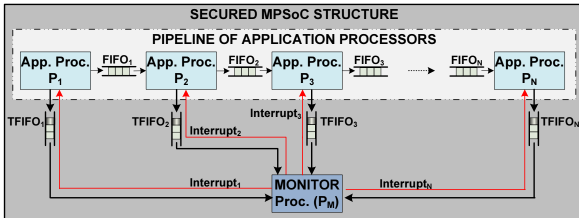


Figure 2: The architectural framework of the proposed design

3.1 Static Analysis

A program is a collection of basic blocks, and we reassert the claim in [14] that ensuring the integrity of the basic blocks suffices to ensure the integrity of the program. The source assembly program is statically analyzed to enable its instrumentation. First, the assembly program in the application processors is divided into basic blocks and instrumented with a special hardware instruction called CHKBLK. Each basic block is assigned a unique block id **bId**. The CHKBLK instruction is inserted at the start of each basic block. Figure 3(a) shows an extract of the assembly program from an application processor and Figure 3(b) shows how the basic block is instrumented with the CHKBLK instruction. The role of the CHKBLK instruction is discussed later in this section.

In the instrumentation process however, if a particular basic block represents a loop, it is to be instrumented slightly differently. If the control flow instruction at the end of a basic block has a possible transition to the start of the same basic block, then that basic block represents a loop. Figure 3(d) shows the instrumentation of the basic block representing a loop in Figure 3(c). Along with the CHKBLK instruction at the start of the basic block in Figure 3(d), a special dummy label **iL6** is also inserted after the CHKBLK instruction. The label **iL6** is inserted and the target of the branch instruction at the end of the basic block is changed to **iL6** to ensure that the CHKBLK instruction is not repeatedly executed in the loop. These basic blocks which represent loops are recorded and the designer is notified in the security tuning process.

The CHKBLK instruction is a hardware instruction that is used to push information into the TFIFO buffer connecting an application processor to the MONITOR. The number in the CHKBLK instruction represents an encrypted block Id and processor Id. Each CHKBLK instruction that is pushed into the TFIFO buffer is also time-stamped with the clock cycle count time (CCOUNT). The timestamp is used by the MONITOR for runtime checking.

Finally, the instrumented assembly file is analyzed for control flow. Since the file is divided into basic blocks, and each basic block has an

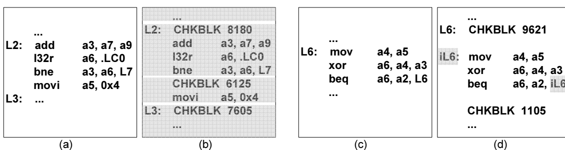


Figure 3: (a) Assembly Extract (b) Instrumentation with CHKBLK instruction (c) Assembly Extract (d) Editing a basic block with loop structure

id, the transitions between basic blocks can be numerically mapped. The control flow generation strategies mentioned in SW_MON and SHIELD had to rely on theoretical estimates or make an exception for the control flow checking hardware for an indirect jump or an indirect call instruction. Our work here uses the output from the profiler to obtain the possible transitions of these indirect control flows. Hence, given that the test data covers the nature of data to be used in the system, the profiler output can be used to resolve the indirect transitions at design time.

3.2 Simulation Analysis and Profiling

After the instrumentation of the assembly source in each application processor, the program execution in each processor is traced and analyzed using an automated script. Each execution of the CHKBLK instruction from the trace is analyzed in a sequential order. The analysis in a sequential order of CHKBLK instructions gives us the basic block to basic block transitions of the application program in each processor.

These transitions are then merged with the transitions obtained from the theoretical analysis in Section 3.1 and this resolves any ambiguity resulting from indirect control flow instructions. Thus we finally obtain a transition table that shows all the possible transitions for each basic block in the application program. The transitions table (also called the control flow table) is stored in the MONITOR for runtime checking.

Since each CHKBLK instruction is time-stamped as described in Section 3.1, the time difference between two consecutive CHKBLK instructions in the trace file gives us the basic block execution time.

A lot of these basic blocks will be executed more than once and it is possible that they have different execution times due to cache hits or misses. Hence we obtain a range of timings for each basic block from the execution trace and select the minimum (min) and maximum (max) times. These min and max times are then stored in the MONITOR for runtime checking.

3.3 Profiler Driven Security Tuning

Our profiler tool analyzes the simulation trace file and extracts the number of times each of the basic blocks was executed and monitored. The number of times a particular basic block was executed and monitored is generally the same, except in the case of basic blocks that are loops. For example, a case where a basic block that is a loop is executed two times; the first time for 10 iterations and the second time for 7 iterations; would be monitored only twice. This is due to the software instrumentation explained in Section 3.1 using Figure 3(c) and (d). Figure 4 shows the monitoring frequency of most of the executed basic blocks plotted on a log scale graph from one of the application cores while the system was executing a JPEG Decoder program. Note that only the basic blocks that were executed more than 10 times are shown.

Looking at the information presented in Figure 4, the designer can clearly see how frequently particular basic blocks are executed and monitored and which of the basic blocks represent a loop. This information can be used to tune the application's security requirements. For example, Figure 4 shows that basic block number 5 is a loop and is executed and monitored about 1600 times. But the individual iterations in each of the 1600 executions are not monitored. If the designer considers that block 5 is extremely important and all the it-

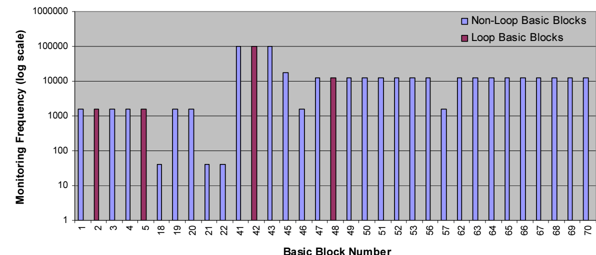


Figure 4: Monitoring frequencies (on log scale graph) of some executed basic blocks in an application processor

erations must be monitored, this can be specified to our design flow, LOCS, which would instrument the basic block 5 such that every iteration is monitored. On the other hand some of the basic blocks which are executed and monitored many times, may not need to be monitored at all. The designer can again control and exclude some blocks from being monitored. It should be noted that LOCS does not need any user intervention and provides a default solution in which every basic block will be checked, but it does offer the option of designer intervention if required.

3.4 Runtime Checks

The runtime checks are performed by the MONITOR using customized hardware. The MONITOR must however initiate the checking using the software instruction VERIFY. The algorithm used by the MONITOR is shown in Algorithm 1.

Algorithm 1 Algorithm in the MONITOR processor

Initialize $finished = 0$;
while (($finished == 0$) AND ($error == 0$)) **do**
  for $j = 1$ to N **do**
    if (TFIFO$_{P_j}$ not EMPTY) **then**
      Read and Decrypt TFIFO$_{P_j}$ Information
  VERIFY(error, finished);

As described in Algorithm 1, all the TFIFOs are checked for data. If the data is available in any of the TFIFOs, it is read and decrypted through a hardware instruction. The VERIFY instruction, which is a *Single Instruction Multiple Data* (SIMD) instruction, performs the timing and control flow checks in hardware. It updates the *error* to 1 if any of the processors fails any of the checks and also updates *finish* to 1 if the application has ended.

The intrinsics of the VERIFY instruction are shown in Figure 5. The encrypted number in the CHKBLK instruction (communicated

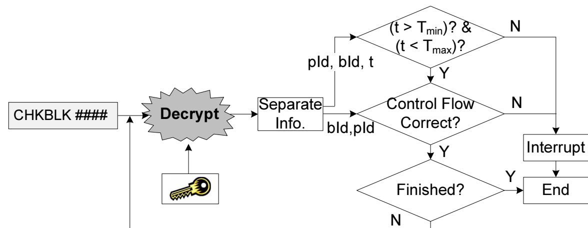


Figure 5: The runtime checks performed in hardware

to the MONITOR through TFIFO) is decrypted using the hardware key in the MONITOR. The time (t), processor Id (pId) and the block Id (bId) information are separated and used to check the validity of the timing and control flow against the stored information in the MONITOR. The time information refers to the execution time of the basic block identified by bId. If the execution time of the basic block bId is less than its stored minimum execution time (T_min) or greater than its stored maximum execution time (T_max), the appropriate application processors referred to by pId are interrupted. The control flow check ensures the transition to the current basic block from the previous basic block is valid. The system exits once the final processor has finished execution.

3.5 Application Behavioral Properties

We discuss in this section, the set of program properties that will be monitored by our system. We define the program behavior in different scenarios when the properties of the program are violated. As discussed in Section 3.4, our MONITOR checks for timing and control flow. The MONITOR generates SIGTI when the timing checks fail and SIGCF when the control flow check fails. We classify the branch, jump, call and return instructions as control flow instructions (CFIs). Also after the software instrumentation, each basic block starts with a CHKBLK instruction and ends with a CFI. We will refer to these instructions as bounding instructions (BI).

Table 1 shows the list of properties that are checked in our system. The first column lists the type of the property being checked. The

Attack Type	Original	Modified	Error
T1	non-BI	added more non-BI	SIGTI
T2	non-BI	CHKBLK	SIGCF/TI
T3	non-BI	CFI	SIGCF/TI
T4	CHKBLK	modify num	SIGCF/TI
T5	CHKBLK	CFI	SIGCF/TI
T6	CHKBLK	non-BI	SIGTI
T7	CFI	another CFI	SIGCF [a]
T8	CFI	modify target	SIGCF
T9	CFI	non-BI	SIGCF [b]
T10	CFI	CHKBLK	SIGTI/CF

Table 1: Different code integrity violations and error signals

[a] Attack not detected if only opcode changed.
[b] Attack not detected if branch instruction changed to non-BI.

second column shows the type of the original instruction in the program and the third column shows the type of the attack instruction. The final column shows the error signals generated by the MONITOR.

In the attack of type T1, where more non-BI instructions are added into the basic block, the timing of the basic block is violated and hence the MONITOR generates SIGTI. In T2, if a non-BI instruction is replaced with a CHKBLK instruction, the MONITOR will generate a SIGCF. The CHKBLK instruction replacing the non-BI instruction is unlikely to have the correct encrypted numbers for the processor Id (pId) and the block Id (bId) and hence generate SIGCF. In T3, the CFI replacing the non-BI instruction may cause a switch to a basic block, which is an incorrect control flow and hence generate SIGCF. In both T2 and T3, the premature end to the basic block means that its execution time is less than its minimum execution time causing the MONITOR to generate SIGTI.

In type T4 attack, the modification of the numbers in the CHKBLK instruction would reveal incorrect pId and bId numbers when decrypted in the MONITOR. This would fail the control flow and timing checks thus generating SIGCF and SIGTI. In T5, when the CHKBLK is changed to a CFI, in the case of an incorrect transition to another basic block, a SIGCF would be generated. But in the case of a continued execution (i.e., the branch is not taken and the execution continues to the next instruction in memory), a SIGTI will be generated as the maximum execution time of the basic block will be exceeded. The attack in T6 is similar to T5 (case of continued execution), causing a SIGTI due to the execution time being exceeded.

In T7, if the existing CFI is changed to another CFI, and the resulting transition is an incorrect control flow, the MONITOR will generate a SIGCF. However, if only the opcode is changed (e.g., branch not equal (bne) changed to branch if equal (beq)) and the targets are left unmodified, the control flow will not be violated and hence the attack would be undetected. Type T8 attacks causes the monitor to generate a SIGCF, if the modified target causes an incorrect control flow. In T9, the MONITOR generates a SIGCF as the original CFI was supposed to cause a change in the control flow to a basic block other than the basic block under the current basic block in memory (e.g., the jump CFI). However, if the original CFI was such that the basic block under the current basic block in memory is a valid transition, then the attack may not be detected (e.g., the branch CFI). In T10, the CHKBLK instruction causes a premature end to the current basic block which results in the execution time of the current basic block to be less than its minimum execution time generating a SIGTI from the MONITOR. Since the CFI in T10 is replaced with a CHKBLK instruction now, the next basic block executed will be the basic block following the current basic block in memory. Therefore, if the original CFI (e.g., jump CFI) was supposed to cause a forced transition to a basic block other than the one following the current basic block in memory, the MONITOR would generate SIGCF.

Any attack that relies on the corruption of the numbers in the CHKBLK instruction would be detected because those numbers are encrypted. It is impossible for an attacker to know the key that is built into the processor hardware. Physical attacks are beyond the scope of this paper but even if the attacker manages to get hold of the key, mass attacks would not be possible as each processor is built with

a random hardware key. Processor designs that exploit the physical characteristics of integrated circuits can be used to embed a key in hardware as was proposed in [18].

4. DESIGN FLOW

Our design flow LOCS, shown in Figure 6, involves a hardware-software co-design approach for implementing security features in an MPSoC architecture. The inputs to LOCS are the source assembly files and the architectural configuration of the design platform. The C/C++ source files are also valid inputs as they can easily be compiled through the target processor's cross-compiler. We assume that each architectural configuration has a unique secure random key built into the processor. As explained earlier in Section 3.5, processors can be designed with secure random keys using the physical characteristics of the integrated circuits. Our design flow, LOCS, automates the software instrumentation and identifies the required hardware customizations in the MPSoC design platform. The final outcome is the customized MPSoC with securely loaded binaries of the application software.

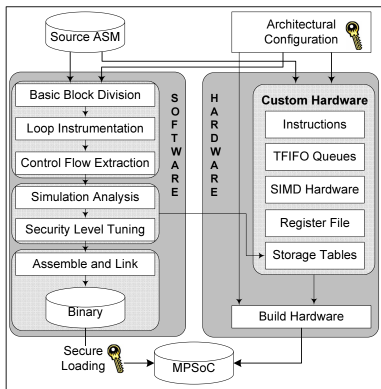


Figure 6: Hardware/Software flow of the proposed design

4.1 Software Design

The software design flow is shown in the left side of Figure 6. The C/C++ source files can be compiled to generate the assembly code for the target instruction set architecture (ISA). The source assembly files are first divided into basic blocks and the basic blocks that represent a loop are identified. Then the control flow map of the program is extracted through static analysis. After this, the programs are simulated and a trace file is generated. The trace file is analyzed to extract the minimum and maximum execution times of basic blocks. The control flow and the timing information is then loaded into the custom hardware tables for use by the MONITOR at runtime. The profile information for the basic blocks obtained from the simulation is used by the designer for modifying the granularity at which some of the basic blocks are monitored.

Finally the instrumented application is assembled and the binary is loaded through a "Secure Loader" into the MPSoC using a secure key (same random key that is built into the architectural configuration). Every basic architectural configuration as well as the secure loader that goes with it are built with a different random hardware key.

4.2 Architectural Design

The hardware design flow is shown in the right hand side of Figure 6. It involves the design of custom instructions, TFIFO queues, Single Instruction Multiple Data (SIMD) hardware units, custom register file and storage tables. The custom instructions are required for interfacing with the TFIFO queues and SIMD units. The SIMD units represent hardware that can handle operations on multiple data. We design the SIMD units for the implementation of the VERIFY instruction in hardware as detailed in Section 3.4. For N application processors, N SIMD units and TFIFO queues are needed. The customized storage tables in hardware are used to store the control flow and basic block timing information available from the software design phase.

5. EXPERIMENTAL SETUP AND RESULTS

In order to evaluate the effectiveness of our proposed design flow we used a commercial processor, Xtensa LX2, from Tensilica Inc. [16]. We extended the base processor's architecture (64 general purpose registers, 80 instructions) with additional custom hardware (register file, storage tables, instructions). We also used the processor's special feature called Ports and Queues to implement the FIFO buffers required for our design flow.

It is difficult to have free access to multiprocessor benchmarks partitioned for Tensilica's toolset but we could still obtain three multimedia benchmark systems which were designed using Tensilica's toolset for our experimentation. These benchmarks were; a six processor JPEG encoder benchmark and a five processor MP3 encoder benchmark produced by Shee et al. in [17], and a five processor JPEG decoder benchmark produced by Wong et al. in [21]. All these partitioned benchmark programs (pipeline of processors) were mapped into Xtensa LX2 using the design flow shown in Figure 6.

LOCS was designed using Python which is a high level programming and scripting language. A number of bash scripts have also been used for generating Makefiles for compilation. We implemented the algorithms described in SW_MON and SHIELD to compare and evaluate with our design flow LOCS. The comparisons and the analysis are presented in the following subsections.

5.1 Performance Metrics

The execution times of three different approaches on three multimedia benchmarks are laid out in Table 2. The JPEG Encoder benchmark was tested on five different benchmark image files. MP3 was tested on a small music file and the JPEG Decoder benchmark was tested on three different benchmark image files of varying sizes. The *App. Exec. time* in the second column refers to the time the application takes to finish execution without the MONITOR and security instructions. The *Sys. Exec. time* in the third to fifth column refers to the time taken by the entire MPSoC system with the MONITOR to finish execution. Table 2 clearly shows that our design flow in LOCS achieves the best system execution time amongst the three approaches and finishes soon after the application.

Benchmark	**App. Exec. time ($\times 10^3$ cc)**	**Sys. Exec. time ($\times 10^3$ cc)**		
		SW_MON	**SHIELD**	**LOCS**
JPEG Encoder				
grandmom	4221.5	42350.1	4504.2	4238.3
mom	4221.7	47674.8	4504.6	4239.3
mom-daughter	4221.7	44170.1	4504.4	4238.5
flower garden	4221.2	85129.9	4516.9	4240.1
tennis	4221.0	67754.2	4507.2	4236.1
MP3	216571.4	6259514.7	274823.0	268628.2
JPEG Decoder				
galois	9266.0	378363.8	17975.9	10836.5
pattern	7587.0	418910.4	21632.8	10430.2
pip	299.6	1403.1	318.9	305.4

Table 2: Results from the tests comparing three approaches

The statistics in Table 2 clearly show that the execution time of the approach in SW_MON is significantly worse than the other two approaches. Figure 7 shows the percentage increase in the clock cycle runtime of LOCS as well as the approach in SHIELD when compared to the Application's Execution time without any security measures. Figure 7 shows that LOCS has significantly reduced overhead than the approach in SHIELD. LOCS has negligible overhead for the JPEG Encoder benchmark, around 25% overhead for the MP3 benchmark and no more than 20% overhead for the three images tested in the JPEG Decoder benchmark.

5.2 Area Overheads

The details of the MPSoCs generated for all three multimedia benchmarks are shown in Table 3. The first column lists the benchmark data files used for each of the multimedia benchmarks tested. The second column refers to the total area of the MPSoCs without the MONITOR and custom hardware. The third, fourth and fifth column gives the new area with the MONITOR and custom hardware for security for each of the three approaches. The fifth column is

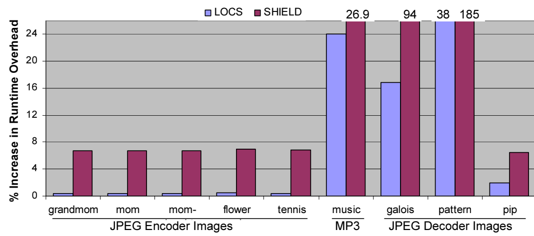
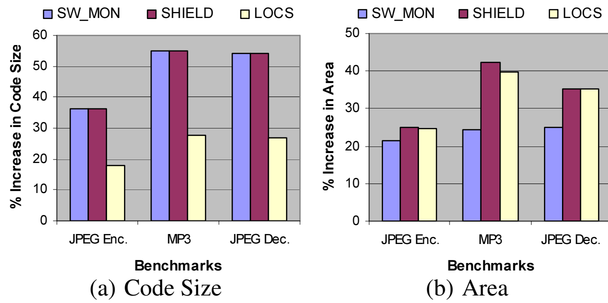


Figure 7: Perf. Overhead Percentage for LOCS and SHIELD



(a) Code Size (b) Area

Figure 8: Percentage Inc. in Code and Area for three approaches

for the power consumption of each core in the MPSoC without the custom hardware.

Benchmark	Orig. Area (mm^2)	Area with MONITOR (mm^2)			Power (mW)
		SW_MON	SHIELD	LOCS	
JPEG Enc.	3.363	4.805	4.208	4.196	55.51
MP3	1.962	2.441	2.792	2.739	134.61
JPEG Dec.	1.427	1.786	1.931	1.929	126.45

Table 3: Area overhead and Power from the experimental setup

Figure 8 shows the comparison between the three approaches in terms of percentage increase in code size and percentage increase in area. The approach in SW_MON is mainly software based and hence it can be seen that there is a larger increase in code size but a smaller increase in area compared to the other two approaches. However it has a significant performance penalty as discussed in Section 5.1.

The comparison of the two mainly-hardware approaches in SHIELD and LOCS clearly shows that LOCS has a smaller percentage increase in the code size and area. In fact, if the approach in SHIELD increases the code size by K instructions, LOCS increases the code size by only $\frac{K}{2}$ instructions. The relationship exists because LOCS adds only one special instruction per basic block whereas the approach in SHIELD adds two.

It should be noted that in both LOCS and the approach in SHIELD, $\frac{100}{N}$% of the area overhead is due to an additional processor, where N is number of processors in the MPSoC excluding the MONITOR. For example, since N = 5 for MP3, 20% of the total 42.3% area overhead refers to the MONITOR; and the other 22.3% is due to the custom hardware. Chip area and designing for flexibility is one of the trade-offs that a designer has to consider during the MPSoC design process. Trade-offs are discussed further in Section 6.

6. DISCUSSION

We employ a dedicated processor in an MPSoC architecture for security purposes. Such a configuration allows for flexibility in the design process, where security is often an after thought. The flexibility arises due to the fact that the MONITOR can be adapted independently of the application processors. If the custom hardware was designed as a functional unit for each processor then each processor's functional unit would need to be changed if there were any changes in the design. This is often an expensive exercise in an ASIP based design.

To address the timing issues in the case of an interrupt in the application processor, we have to employ a separate time counter rather than using the application processor's CCOUNT register. The counter register will be incremented on every clock cycle except when the application processor is in the interrupt subroutine (ISR). To address the limitation of not being able to perform timing checks on the blocks that are involved in an inter-processor communication, we propose to stall the appropriate counters when the TFIFOs are full or empty. Our approach can be readily scaled to larger systems by employing more than one MONITOR. A greater number of MONITORS would result in a faster detection of code corruption or faults but would also mean a higher area overhead. The designers must consider the area and speed requirements of the system being designed while deciding on multiple MONITORS.

In addition to detecting code injection attacks, our system can also tackle reliability issues caused due to control flow errors. Preliminary testing on using LOCS for detecting random bit flips in the control flow instructions yielded a promising detection rate of approximately 80%. Our future work will include results for detection of control flow errors induced due to reliability issues.

7. CONCLUSIONS

In this paper, we have presented a very low overhead design flow for detecting code injection attacks in an MPSoCs architecture. We have formulated a list of admissible application behavior for our MPSoC system. We have devised an automatic hardware-software methodology to design our solution using a commercial ASIP design tool from Tensilica Inc. We have tested our system on multimedia benchmarks and reported the performance, area and code size overheads. The results show that these overheads are much lower when compared to previously proposed methods. The applicability of LOCS for detecting control flow errors caused due to reliability errors were identified. We conclude that our technique is a low overhead solution for addressing concerns regarding code injection attacks in MPSoCs.

8. REFERENCES

[1] D. Arora et al. Secure embedded processing through hardware-assisted run-time monitoring. In *DATE '05*, pages 178–183, Washington, DC, USA, 2005.

[2] C. Cowan et al. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan 1998.

[3] 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, NY, USA, 2003.

[4] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. pages 177–190, 2001.

[5] E. Larson and T. Austin. High coverage detection of input-related security facults. In *SSYM'03*, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[6] M. Loghi, M. Poncino, and L. Benini. Cycle-accurate power analysis for multiprocessor systems-on-a-chip. In *GLSVLSI '04*, pages 410–406, NY, USA, 2004.

[7] J. Mcgregor et al. A processor architecture defense against buffer overflow attacks. pages 243–250, 2003.

[8] M. Milenkovic, A. Milenkovic, and E. Jovanov. Hardware support for code integrity in embedded processors. In *CASES '05*, pages 55–65, NY, USA, 2005.

[9] 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.

[10] K. Patel and S. Parameswaran. Shield: A software hardware design methodology for security and reliability of mpsocs. *DAC 2008*, pages 858–861, June 2008.

[11] K. Patel, S. Parameswaran, and S. L. Shee. Ensuring secure program execution in multiprocessor embedded systems: a case study. In *CODES+ISSS '07*, pages 57–62, New York, NY, USA, 2007.

[12] J. Pincus and B. Baker. Beyond stack smashing: Recent advances in exploiting buffer overruns. *IEEE Security and Privacy*, 2(4):20–27, 2004.

[13] R. G. Ragel. *Architectural Support for Security and Reliability in Embedded Processors*. PhD thesis, School of CSE, UNSW, Sydney, Australia, 2006.

[14] 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.

[15] S. Ravi et al. Security in embedded systems: Design challenges. *ACM Trans. Embedded Comput. Syst.*, 3(3):461–491, 2004.

[16] C. Rowen and D. Maydan. Automated processor generation for system-on-chip. Technical report, Sept 2001.

[17] S. L. Shee and S. Parameswaran. Design methodology for pipelined heterogeneous multiprocessor system. In *DAC*, pages 811–816, 2007.

[18] G. E. Suh and S. Devadas. Physical unclonable functions for device authentication and secret key generation. In *DAC '07*, pages 9–14, New York, USA, 2007. ACM.

[19] D. Wagner et al. 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.

[20] W. Wolf. The future of multiprocessor systems-on-chips. In *DAC '04*, pages 681–685, New York, NY, USA, 2004.

[21] J. Wong, A. Ignjatovic, and A. Janapsatya. Multiprocessor implementation of image compression algorithms. In *BE Thesis*, 2007.

[22] 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.