Architectural Frameworks for Security and Reliability of MPSoCs

Krutartha Patel, *Member, IEEE,* Sri Parameswaran, *Member, IEEE,* and Roshan Ragel, *Member, IEEE*

***Abstract*—Multiprocessor System on Chip (MPSoC) architectures are increasingly used in modern embedded systems. MPSoCs are used for confidential and critical applications and hence need strong security and reliability features.**

***Software attacks* exploit vulnerabilities in the software on MPSoCs. In this paper we propose two MPSoC architectural frameworks, tCUFFS and iCUFFS, for an Application Specific Instruction set Processor (ASIP) design. Both tCUFFS and iCUFFS employ a dedicated security processor for detecting software attacks.**

iCUFFS relies on the exact number of instructions in the basic block to determine an attack and tCUFFS relies on time-frame based measures. In addition to software attacks, reliability concerns of bit flip errors in the control flow instructions (CFIs) are also addressed. Additional method is proposed to the iCUFFS framework to ensure reliable inter-processor communication.

The results for the implementation on Xtensa processor from Tensilica showed, worst case runtime penalty of 38% for tCUFFS and 44% for iCUFFS, and worst case area overhead of 33% for tCUFFS and 40% for iCUFFS. The existing iCUFFS framework was able to detect approximately 70% of bit flip errors in the CFIs. The modified iCUFFS framework proposed for reliable inter-processor communication was at most 4% slower than the existing iCUFFS framework.

***Index Terms*—Architecture, Code Injection, Reliability, Instruction Count, MPSoC, Tensilica**

I. Introduction

In systems design, Multiprocessor System on Chips (MPSoCs) are emerging as the pre-eminent design solution to increasing demands in functional requirements, low power needs, and programmability [1]. The multimedia devices such as portable music players and cell-phones already deploy MPSoCs to exploit data processing parallelism and provide multiple functionalities [2, 3]. With increased functionalities the complexity of the design increases, and therefore the susceptibility of the system to attacks from adversaries. The small form factor for aesthetics of the devices and deeper pipelines to increase clock frequency for faster throughput have also been responsible for reliability errors [4].

Embedded systems designers rarely include security in their design objectives. The short design turnaround times, due to competitive pressure of getting a system out in the market, is often soaked up by getting the functionality, performance and energy requirements correct [5]. Weaknesses in system implementation inevitably remain and are often exploited by the attackers in the form of either physical, software or side-channel attacks. Software attacks that exploit vulnerabilities in software code or weaknesses in the system design are the most common type of attacks [6]. A reprieve from an attack still does not guarantee correct execution of the software because there could be reliability errors. Reliability errors may further hinder correct execution of the program due to, for example, bit flips errors [7].

Recent literature suggests that newer security threats targeting portable electronics like mobile phones and music players may pose significant risks [8, 9]. Given that such devices already employ MPSoC architectures, it is imperative that security is considered at design time rather than be employed as a reactive measure. Incorporating security in the design definitely increases overheads, but given the ability of attacks to cause fraud, disrupt activity or threaten the confidentiality of data, the overheads are worth the cost [6, 10].

Software attacks in systems usually aim to execute malicious code that is either already present in the system or is injected. Stack and heap based buffer overflows are the most common type of software attacks [11]. The buffer overflow vulnerabilities in application programs have been exploited since 1988 [12] and still continue to be exploited. On average nearly 11% of the vulnerabilities reported by the US-CERT vulnerability reports over the last three years pertain to buffer overflow attacks. Figure 1 shows the percentage of buffer overflow attacks in each month of 2006, 2007 and 2008.

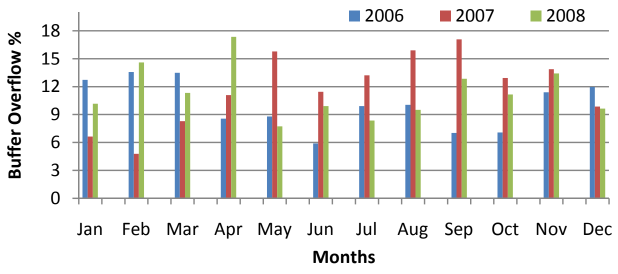


Fig. 1. US-CERT reported buffer overflow vulnerabilities

Embedded devices are moving towards miniaturization to achieve a small form factor [2, 3]. The improvement in nanometer technology is helping to achieve miniaturization. However, along with these advancements, there are several challenges brought about in terms of device reliability. The work in this paper targets *Soft Errors*, which normally arise from Single Event Upsets (SEUs). SEU is a change of state that can occur when ionizing radiation strikes a microelectronic device like a microprocessor, semiconductor memory or power transistors [13]. An SEU can result in a signal or datum being garbled or wrong.

Soft errors are a type of a transient fault that occur due to random events. Researchers have predicted an increase in soft errors due to advances in low power and low voltage technologies and increased clock frequencies [14, 15]. The reduced voltage level of the current microprocessors make them susceptible to corruption. For example, if small voltage levels with a small difference are used to represent bits 0 and 1, then exposure to ionizing radiation may easily alter the voltages and hence the bits [16]. Decreasing voltages and the miniaturization of devices has consequently brought about an increase in soft-error-rates (SERs) [17].

A. Paper Overview

In this paper, we draw a comparison between two architectural framework for detection of software attacks. One of the frameworks (iCUFFS) is based on ensuring that the correct number of instructions

K. Patel and S. Parameswaran are with the School of Computer Science and Engineering, University of New South Wales (UNSW), Sydney, NSW 2052 AUSTRALIA e-mail: (kpatel45@gmail.com, sridevan@cse.unsw.edu.au). R. Ragel is with the University of Peradeniya, SRI LANKA e-mail: (roshanr@ce.pdn.ac.lk)

Manuscript received August xx, 2009; revised XXXX xx, 2009.

are executed between "check-points" and the other (tCUFFS) is based on ensuring that the number of clock cycles between two checkpoints is within some pre-analyzed limits. In addition both tCUFFS and iCUFFS frameworks perform control flow checks of the program execution that help detect both security and reliability errors. We design an MPSoC with a dedicated security processor called the MONITOR. We tackle the issue of adding security to MPSoC systems at the processor design level because the overheads are lower when compared to the overheads incurred when addressing security at the software level [18, 19].

Each basic block in the application processors of the MPSoC is instrumented with one or two check-points. These check-points represent a special instruction that reports to the MONITOR at runtime. For the tCUFFS framework, the special instruction reports the cycle count time and for the iCUFFS framework, the special instruction reports the instruction count. Static analysis on the program is performed at compile time to extract the control flow of the program. For the tCUFFS approach the minimum and maximum execution time between check-points is determined by simulation analysis and for the iCUFFS approach the number of instructions between the check-points are determined using static analysis. For the tCUFFS approach, the control flow, and the minimum and maximum execution times are stored inside the hardware tables. For the iCUFFS approach, the control flow, and the number of instructions are recorded inside hardware tables of the MONITOR.

At runtime, the application processors report to the MONITOR using the special instructions as to which basic block they are executing and the value of the processor's Cycle Count (CC) register (for tCUFFS) or the Instruction Counter (IC) register (for iCUFFS). The MONITOR uses the communicated information to check that the control flow is correct and that the number of clock cycles or instructions between two check-points is in accordance with the information stored in its tables. However, if the MONITOR finds that the control flow is incorrect or that the number of clock cycles or instructions between two check-points mismatch with the value in its hardware tables, it sends an interrupt to all the processors to abort execution.

One of the novel contributions of the iCUFFS framework is the "active" MONITOR processor. By "active" we mean that it checks the value of the IC register of the application processors rather than just relying on only the information communicated to it from the application processors. By reading the value from the IC, the MONITOR determines whether or not an application processor has missed reporting at a check-point. If the MONITOR finds that a check-point has been passed through without reporting, an attack is inferred and the application processors' execution on the MPSoC is interrupted. Therefore the framework allows detection of attacks even when the application processors do not communicate with the MONITOR.

The iCUFFS framework proposed in this paper, is also applied to test for reliability errors in the control flow instructions of the application processors. Moreover, a checksum based variation of the iCUFFS framework is also proposed for reliable inter-processor communication on MPSoCs.

The frameworks tCUFFS and iCUFFS, have differing strengths and weaknesses. This allows one framework to be more suitable to a designer's requirements than the other. The tCUFFS framework has a lower code size and therefore performance overhead compared to the iCUFFS framework. However, iCUFFS has lesser area overhead compared to tCUFFS, therefore iCUFFS is more suitable to an MPSoC design with a tighter area constraint.

The remainder of the paper is organized as follows. Related Work is presented in Section II. The motivation, problem statement and assumptions of our work are in Section III. The architectural design for tCUFFS and iCUFFS is shown and contrasted in Section IV. Section V and Section VI explain as well as differentiate the software and hardware design flows respectively for both our designs. Section VII presents scenarios of how our two designs will protect an MPSoC against attacks. Experimental results of both the designs are presented in Section VIII. Section IX shows the use of iCUFFS framework for reliability analysis. Discussion and conclusions are presented in Section X and Section XI respectively.

II. Related Work

The countermeasures to software attacks can be broadly classified into either software based or architectural (hardware) based. Software based countermeasures consist of either static or dynamic techniques. Static analysis tools help in removing possible vulnerabilities in the code at compile time. Various static analysis techniques have been proposed in the literature [20–24]. Dynamic analysis techniques aim to detect errors or attacks at runtime. A well-known dynamic analysis tool CCured uses both static analysis and efficient runtime checks to ensure that the pointers are used safely in C programs [18].

Hardware techniques for detecting attacks usually use customized hardware blocks for runtime checks. McGregor et al. proposed a special return address stack (SRAS) [25] for protecting against buffer overflow attacks while Arora et al. [26] proposed a hardware monitor that uses the trace of the executing instructions and program addresses for detecting common software and physical attacks. Milenkovic et al. [27] proposed a signature verification unit that checks the instructions that are fetched from the memory. Ragel et al. [28] proposed a basic block validation scheme by modifying the processor's microinstructions. Nakka et al. [29] proposed a processor pipeline modification framework for detecting a process crash or hang. Wang et al. [30] proposed checking the instruction counter register at the function level for detecting incorrect execution paths in programs.

Static analysis techniques do not capture all the vulnerabilities in the code and often raise a number of false positives. Some, like the Stack Guard [20], aim to solve specific problems like the buffer overflow attacks and may not work for other types of software attacks. Dynamic code analysis techniques often incur high runtime overheads due to extra processing at runtime. For example, CCured [18] incurs performance overhead of up to 150%.

A majority of the proposed hardware based methods need significant architectural modifications which is a major limitation for commercial and extensible processors like Tensilica's Xtensa LX2. Xtensa LX2 provides a base processor implementation which can be extended using custom instructions defined using TIE (Tensilica Instruction Extension). Furthermore, the hardware description of the base processor is unavailable, which restricts major modifications to the processor.

The SRAS [25] and the hardware monitor [26] are not scalable for commercial processors like Xtensa LX2 due to unavailability of a special stack required for SRAS and access to the executed instructions (at runtime) required by the hardware monitor. Access to the instruction register (IR) is also unavailable in Xtensa LX2 and hence signature verification [27] is not possible. The microinstructions modification [28] and the pipeline modification [29] are also not possible due to the unavailability of the base processor's hardware description. The approach proposed by Wang et al. [30] needs various training data sets to build the instruction count values for program path patterns. New program paths encountered at runtime which are not in the training set result in false positives.

Therefore the existing single processor software and hardware solutions discussed above are not quite scalable or need significant architectural modifications which is unrealizable for extensible commercial processors like Xtensa LX2.

Two approaches, a software solution [19] and a hardware-based solution [31] have been previously proposed for detecting software attacks in the multiprocessor domain.

Our work differs from the previous work for detecting software attacks on an MPSoC architecture in the following ways. The tCUFFS uses only one special instruction per basic block as opposed to two special instructions used by [19, 31]. Therefore the code overhead in tCUFFS is half compared to [19] and [31]. Moreover tCUFFS also checks every single line of program code compared to [19, 31]. The code overhead in iCUFFS will always be less than or equal to the code overhead in [19] and [31] because unlike two special instructions per basic block in [19] and [31], iCUFFS uses either one or two special instructions per basic block.

The iCUFFS framework uses the **number of executed instructions** compared to the use of **execution time in clock cyles** in [19, 31] to verify correct execution between two check-points in an application program. The iCUFFS framework therefore knows the **exact** number of instructions that must be executed from one check-point to the next compared to the time reliant methodology in [19, 31], which employs a range of execution times.

The approaches in [19, 31] proposed a dedicated processor for security which was “passive”; i.e., the security processor would only perform timing or control flow checks when the application processors communicated. In contrast, our iCUFFS framework proposes an “active” processor that probes all the application processors on the MPSoC by regularly reading their IC for security checks. Hence iCUFFS even detects attacks that can hijack the processor for executing malicious code and never communicate with the security processor whereas neither of the approaches proposed in [19, 31] could detect such attacks.

The work proposed in [19, 31] requires the program’s execution trace to find the range of execution times a basic block can take. Furthermore, the basic blocks that do not fall on the execution path have their execution times estimated using the processor’s instruction set architecture (ISA). The iCUFFS framework only needs to know the **exact** number of instructions in each basic block which is available by static analysis of the assembly code and hence iCUFFS neither needs any execution trace analysis nor does it need to resort to estimation.

Both the tCUFFS and iCUFFS frameworks we propose, can be used to detect soft errors in the control flow instructions (CFIs). Additionally, we propose for the first time, a modified iCUFFS framework to ensure reliable inter-processor communication for an MPSoC framework.

A. Reliability

The advent of advanced fabrication technologies provides faster and powerful functionality but at the same time brings about significant reliability concerns [32, 33]. We target *Soft Errors*, also known as SEUs (Single Event Upsets) that result in a signal or datum being garbled or wrong. An explanation of how the soft errors may happen is detailed below.

Transient faults are one of the reliability concerns and a study by Siewiorek et al. in [34, 35] revealed that more than 90% of the system faults are caused by transient faults. Transient faults occur due to many reasons that include electromagnetic interference, power fluctuations, interconnect noise and soft errors. Soft errors are a major concern due to technological advances like deep pipelines, device scaling, lower power consumption and supply voltage [4, 15, 36–39].

Some reliable designs to protect against fault tolerances and Soft Errors, proposed in the literature [40–42], rely on aggressive redundancies. This category of techniques is normally referred to as *Modular Redundancy* techniques. Reis et al. rely on duplications of some important registers like the stack pointer and a flag register. Hopkins et al. provide a Fault Tolerant Multiprocessor (FTMP) architecture for aerospace applications [41]. In the FTMP approach by Hopkins et al., the information is processed and transmitted in triplicates so that errors can be corrected. Avizienis uses a multiple computation approach (by N-fold where $N \geq 2$) that performs computations in three domains: time(repetition), space(hardware) and information (software).

The work proposed by Bagchi et al. proposes a preemptive control based signature checking (PECOS) mechanism for single processors [43]. PECOS employs a software based methodology that uses embedded assertions in the assembly code, which are triggered at runtime [43].

The techniques proposed by Ragel et al. [7] involve architectural modification for checking control flow errors. The technique involves duplicating the control flow instruction fetch, then performing hardware checks to detect the bit flips in the instruction memory.

Ramamurthy et al. propose a watchdog processor based concurrent error detection mechanism and error recovery [44]. The approach uses signature analysis and is used to detect bit as well as control flow errors. The watchdog processor presented by Ramamurty et al. is add-on hardware and hence it would require integration with an existing processor.

Another watchdog monitoring approach using a watchdog processor is suggested by Michel et al. [45]. This allows control flow checking without the need to modify the program. The watchdog processor has two tasks. The first is to compute the signature of the executed instruction sequence. The second is the detection of the nodes reached by the main processor.

Modular Redundancy techniques are expensive due to the massive amount of redundancy involved. Generally, Modular Redundancy techniques are not plausible in embedded systems which have tight space and speed requirements. The approaches mentioned above [40–42] face high overheads due to redundancy.

PECOS has a significant code overhead of between 50% 150%. Additionally, it also has a significant program storage overhead of greater than 100% in average cases. The program storage overhead is a result of storing the reference signatures of basic blocks and checking code [43]. PECOS detects around 87% of control flow errors.

The approach in Ragel et al. [7] relies on micro-instruction modification of the instruction set architecture. It also requires implementation of a *Shadow PC* to overcome the problem of bit flips or a burst in the **program counter** (PC) register. To implement a *Shadow PC* as well as to modify the micro-instructions, a designer needs access to the hardware implementation of the processor. Commercial processors like Tensilica do not allow access to the hardware implementation of the processor.

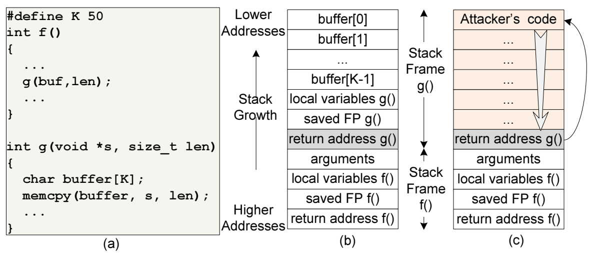
The approaches by Ramamurthy et al. and Michel et al. [44, 45] use a watchdog processor, but they are limited to checking errors in only one processor. Hence the approaches by Ramamurthy et al. and Michel et al. [44, 45] can’t be used for MPSoCs.

Both the tCUFFS and iCUFFS frameworks we propose, can be used to detect soft errors in the control flow instructions (CFIs). Additionally, a modified iCUFFS framework is proposed to ensure reliable inter-processor communication.

This paper proposes a framework for incorporating security and reliability features on an MPSoC. One of the novel contributions of this paper to the literature is that it only uses the existing design flow of commercial processors for an MPSoC design. There is no extra hardware added to the commercial processor beyond the minor hardware extensions that are allowed by the commercial processor. Therefore, not only do we explore the unknown territory of MPSoCs, but we also propose a security and reliability solution for MPSoCs. Previous hardware approaches were on single processors and relied heavily on significant hardware modifications. Newer attacks like side-channel attacks rely on monitoring bus traffic on an MPSoC. Although, side-channel attacks are beyond the scope of this paper, we identify the threat of monitoring bus traffic to reliable communication on an MPSoC. Therefore, we provide measures to protect the inter-processor and inter-chip communication by encrypting them.

III. Motivation, Problem Statement & Assumptions

Figure 2 shows an example of a stack buffer overflow attack. Figure 2(a) shows a snippet of vulnerable C code, Figure 2(b) shows the layout of the stack when function **g** is called from function **f**. As part of writing data to the array **buffer** in **g**, the attacker may supply malicious code in array **buf** before making a call to **g**. Passing a sufficiently higher value than **K** (which in this case is 50), in **len**, would ensure that the stack overflows and the return address is overwritten as shown in Figure 2(c). Thus the control flow of the program is changed to execute malicious code. This change in behavior disrupts the code integrity and causes fallacious program behavior.



```
#define K 50
int f()
{
  ...
  g(buf,len);
  ...
}

int g(void *s, size_t len)
{
  char buffer[K];
  memcpy(buffer, s, len);
  ...
}
```

Fig. 2. A stack based buffer overflow attack.

As discussed earlier in Section I, the software attacks are the most common types of attacks. Such attacks do not require any special equipment or sophisticated techniques, unlike physical attacks or side-channel attacks. A detailed explanation of common software attacks (heap attacks, format string vulnerabilities, arc injection, etc.) can be found in the literature [11, 46, 47].

Our work targets software attacks on an MPSoC architecture that aim to subvert the control flow of the user's application to execute malicious code. Stack and heap based buffer overflows (code injection attacks), pointer subterfuge attacks and arc injection attacks are prime examples of software attacks that are targeted in this work. We do not target physical attacks, such as damaging the MPSoC by force or erasure of data/instruction memory through physical access, to the device. Our work also does not target side-channel attacks on embedded systems which typically involves the use of power measurements (or other signals which emanate from the device) to find crucial information from the application program, such as encryption keys.

We assume that the system calls are safe and hence need not be supervised. If needed however, the functions in the system library can also be easily instrumented using the tCUFFS and iCUFFS frameworks discussed in this paper. We also assume that a secure "loader" is used for loading the programs in the processor. The "loader" is trusted so that it cannot compromise the program code.

We assume that the programs that execute on each application processor are fixed at design time and also that the MONITOR can be completely secured. This is a reasonable assumption given that MONITOR is a dedicated processor for security and only runs a loop that executes the customized hardware instructions. This small number of instructions can be easily placed in a ROM as the instructions need not change.

The storage tables are built in hardware, keeping in mind the speed/performance impacts. There may be memory access latencies involved which reduces the performance, if the tables were not built in hardware. However, if the problem was done in other platforms like ASIPMeister (where hardware description of processor implementation was available unlike Xtensa LX2), a loadable table may be considered. Such loadable tables would be updated when software is updated. However, our work hasn't explored this option of loadable tables yet.

IV. MPSoC Architectural Design

We have implemented the proposed frameworks of tCUFFS and iCUFFS using the Xtensa LX2 processor from Tensilica Inc. The Xtensa LX2 processor provides a base core implementation that contains 80 instructions. The base core can be further customized from Xtensa's existing resource pool by adding co-processors, multiplier units, Boolean registers, local memories, etc. It is also possible to customize the processor by changing features such as the pipeline length and instruction fetch widths. Besides the customizations from the existing resource pool, user-defined hardware instructions can be created using Tensilica Instruction Extension (TIE) language. Xtensa LX2 also provides implementation for ports and queues which we use in our architectural framework. The Xtensa LX2 processor also allows defining custom register files and storage tables for constants.

The common MPSoC architecture layout we propose (for both tCUFFS and iCUFFS) in this paper is shown in Figure 3(a). The extension to the common architecture layout which is necessary for iCUFFS is not shown in Figure 3(a), and will be shown later. In the MPSoC layout in Figure 3(a), there are N application processors and one additional MONITOR processor that supervises the N application processors. The N application processors can execute in any arbitrary fashion. For example, the programs can execute independently as shown in Figure 3(a) or as a pipeline of processors communicating amongst themselves.

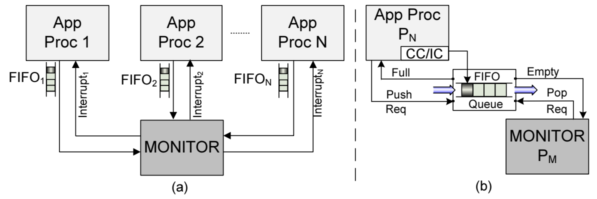


Fig. 3. (a) An MPSoC system with a MONITOR (b) Communications via FIFO

On the MPSoC system, one of the key features we employ is a FIFO queue for inter-processor communication. The FIFO queue allows communication at runtime between an application processor and the MONITOR processor. The FIFO shown in Figure 3(b) ensures that every time an input is received, a **CC** (from Cycle Count Register) or **IC** (from Instruction Count Register) reading is attached to the input. The **CC** is utilized by the tCUFFS approach where as the IC is utilized by the iCUFFS approach for ensuring correct program execution. The FIFO queue stalls when attempting to read from an empty queue and write to a full queue using the Empty and Full signals shown in Figure 3(b).

The architectural layout for the iCUFFS approach is shown in Figure 4. It is an extension to the tCUFFS layout shown earlier in Figure 3(a). The iCUFFS design is equipped with a hardware unit called **CHK_IC** that allows the MONITOR to probe all the application processors to obtain their **IC** reading through a shared memory interface as shown in Figure 4. The **CHK_IC** allows the MONITOR to detect an attack even in the case of an application processor being hijacked by an attacker. The MONITOR's active probing of the application processors allows it to foil an attack even if the attacker prevents any communication from the application processors using the special FIFO instructions. The methodology is described in detail as a combination of software and hardware design flows in the following two sections.

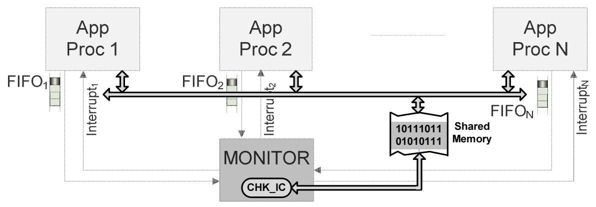


Fig. 4. The design of the iCUFFS architectural framework

V. Software Design Flow

The software design flow used by both tCUFFS and iCUFFS is shown in Figure 5 and discussed in the following two subsections.

A. Software Design Flow - tCUFFS

Firstly, the application program's source code in C/C++ is compiled to obtain the source code in assembly. The assembly source code is then divided into basic blocks (BBs) as shown in Figure 6(a) by the dotted lines. A BB is defined as a set of sequential instructions that end in a control flow instruction like a **branch**, **jump**, **system call** or **function call** instruction. Once the program is divided into BBs, static analysis is performed to yield a **control flow graph** of the program at the BB level which is shown in Figure 6(b).

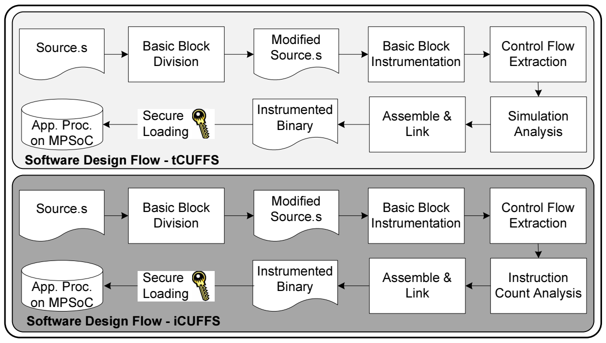
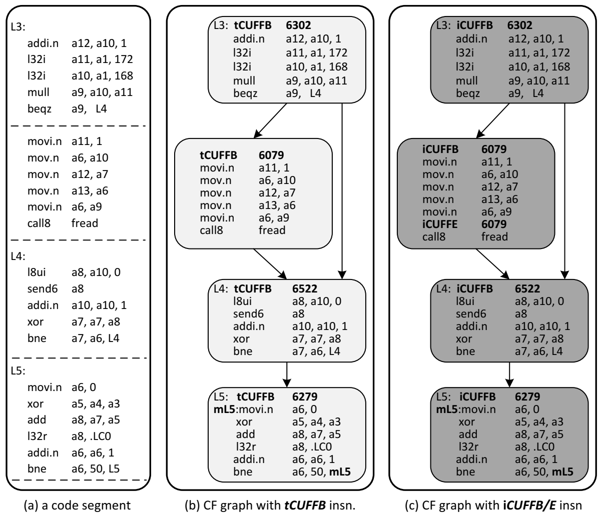


Fig. 5. The software design in the proposed framework

Dividing a program into basic blocks allows low level monitoring of a program running on an application processor in an MPSoC. The advantage of monitoring a program at the granularity of BB allows a rapid stop of the system on compromise, and an attack can be narrowed to a small chunk of instructions. Commercial processors do not allow access to processor implementation or special registers like program counter and instruction register, hence monitoring at a lower granularity than BB is not possible. Monitoring at a function level is possible (higher granularity than BB), but functions can comprise of several control flow instructions and it is difficult to isolate the place of attack inside a function.



```
L3:
    addi.n   a12, a10, 1
    l32i     a11, a1, 172
    l32i     a10, a1, 168
    mull     a9, a10, a11
    beqz     a9,  L4
- - - - - - - - - - - - - -
    movi.n   a11, 1
    mov.n    a6, a10
    mov.n    a12, a7
    mov.n    a13, a6
    movi.n   a6, a9
    call8    fread
- - - - - - - - - - - - - -
L4:
    l8ui     a8, a10, 0
    send6    a8
    addi.n   a10, a10, 1
    xor      a7, a7, a8
    bne      a7, a6, L4
- - - - - - - - - - - - - -
L5:
    movi.n   a6, 0
    xor      a5, a4, a3
    add      a8, a7, a5
    l32r     a8, .LC0
    addi.n   a6, a6, 1
    bne      a6, 50, L5
- - - - - - - - - - - - - -
```

```
L3:  tCUFFB    6302
     addi.n    a12, a10, 1
     l32i      a11, a1, 172
     l32i      a10, a1, 168
     mull      a9, a10, a11
     beqz      a9,  L4

     tCUFFB    6079
     movi.n    a11, 1
     mov.n     a6, a10
     mov.n     a12, a7
     mov.n     a13, a6
     movi.n    a6, a9
     call8     fread

L4:  tCUFFB    6522
     l8ui      a8, a10, 0
     send6     a8
     addi.n    a10, a10, 1
     xor       a7, a7, a8
     bne       a7, a6, L4

L5:  tCUFFB    6279
mL5:movi.n     a6, 0
     xor       a5, a4, a3
     add       a8, a7, a5
     l32r      a8, .LC0
     addi.n    a6, a6, 1
     bne       a6, 50, mL5
```

```
L3:  iCUFFB    6302
     addi.n    a12, a10, 1
     l32i      a11, a1, 172
     l32i      a10, a1, 168
     mull      a9, a10, a11
     beqz      a9,  L4

     iCUFFB    6079
     movi.n    a11, 1
     mov.n     a6, a10
     mov.n     a12, a7
     mov.n     a13, a6
     movi.n    a6, a9
     iCUFFE    6079
     call8     fread

L4:  iCUFFB    6522
     l8ui      a8, a10, 0
     send6     a8
     addi.n    a10, a10, 1
     xor       a7, a7, a8
     bne       a7, a6, L4

L5:  iCUFFB    6279
mL5:movi.n     a6, 0
     xor       a5, a4, a3
     add       a8, a7, a5
     l32r      a8, .LC0
     addi.n    a6, a6, 1
     bne       a6, 50, mL5
```

(a) a code segment
(b) CF graph with ***tCUFFB*** insn.
(c) CF graph with ***iCUFFB/E*** insn

Fig. 6. Basic block division and control flow extraction

Each processor is assigned a unique processor ID and each BB of the program in the processor is assigned a unique block ID. Using the processor ID and the block ID, a special ID called *SID* is created for each BB. We assume that this *SID* is encrypted using a distinct encryption key (based upon physical uncloneable functions (PUF), proposed in [48], to acquire an encryption key using the physical properties of integrated circuits in the MPSoCs) at load time by the secure "loader". An exact copy of the encryption key is also stored in hardware as shown in Figure 7 to decrypt the *SID* at runtime. The importance of encrypting the *SID* is further discussed in Section VII.

The BB instrumentation procedure is different for the two frameworks. For the tCUFFS framework, each BB is instrumented with only a single special instruction called *tCUFFB* as shown in top three boxes in Figure 6(b). The *tCUFFB* instruction is inserted at the start of each basic block. The number in the *tCUFFB* instruction is the encrypted *SID*. A BB representing a loop where the frequency of execution can be statically known is instrumented slightly differently by our static analyzer as shown in the last BB of Figure 6(b). An extra label (this case **mL5**) is inserted after the *tCUFFB* instruction and the target of the branch is changed to this extra label **mL5**. This type of instrumentation allows the *tCUFFB* instruction to be executed only once for each execution of the loop.

B. Software Design Flow - iCUFFS

The stages in the software design flow for iCUFFS are identical to tCUFFS except that tCUFFS uses simulation analysis where iCUFFS uses instruction count analysis.

For the iCUFFS approach, each BB is instrumented with one or two special instructions as shown in Figure 6(c). A special *iCUFFB* instruction is always added as the first instruction in each BB.

For a BB that ends in a system call, another instruction *iCUFFE* is added before the system call. An example of such a BB is shown in the second BB box in Figure 6(c). Therefore BBs ending in a system call contain two special instructions per BB.

The number in the *iCUFFB* and *iCUFFE* instruction is the encrypted *SID*. The instrumentation for a BB representing a loop (where the execution frequency can be statically known) is done in exactly the same manner as described above for tCUFFS. The only difference being that in the iCUFFS approach, the special instruction added is *iCUFFB* instead of *tCUFFB*.

C. Control Flow Extraction - tCUFFS and iCUFFS

The static analysis of the instrumented assembly file also yields a control flow map of the program which is shown using arrows in Figure 6(b) and (c) for tCUFFS and iCUFFS respectively. Since each of the BBs are given a *SID*, the control flow map can be expressed in terms of *SID*. The BBs with an indirect control flow, however may not be resolved at compile time. Hence simulation trace file analysis may be required. This analysis is described in the next subsection.

D. Simulation Analysis - tCUFFS

The ‘Simulation Analysis’ stage in the tCUFFS approach extracts the minimum and maximum execution time for each BB by analyzing the execution trace after simulation. 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 of a particular basic block, we get the execution time of that basic block. It is likely that some basic 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 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 does 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 by using the instruction set simulator’s (ISS’s) general guide. A history of the same operation instruction can also be looked at in the tracefile to get an estimate on the min and max time for the instruction.

Once each instruction’s min and max times are estimated in this unexecuted basic block, we can sum up these estimated values and get the estimated execution time for the basic block. The minimum and the maximum execution time of all the basic blocks are recorded and stored in the MONITOR processor for the tCUFFS approach.

The simulation analysis can also be used to resolve indirect branches in the code. The tracefile shows the control flow transitions from the BB containing the indirect control flow instruction (CFI) to other BB. Thus the control flow graphs generated in the previous subsection can be further reinforced using the analysis of the tracefile. However, not all the indirect CFIs may have been executed and therefore not all the indirect CFIs can be resolved through tracefile analysis. Hence the tCUFFS and iCUFFS approaches have a limitation that an indirect CFI may not be monitored unless, there exists a set of particular basic blocks that the indirect CFI can have a transition to.

E. Instruction Count Analysis - iCUFFS

For the iCUFFS approach, the ‘Instruction Count Analysis’ stage simply refers to gathering information regarding the number of instructions in each BB, which can be obtained by going through the instrumented assembly file. Thus the relevant information about the number of instructions for iCUFFS is stored in the hardware tables of the MONITOR processor. These tables will be used at runtime by the MONITOR processor for security checking.

For the iCUFFS approach, Simulation Analysis is only necessary for processors that need to resolve indirect control flow addresses. The resolution of indirect control flow addresses is done in the same way as it is done for tCUFFS as explained in the previous subsection.

F. Assemble, Link and Loading - tCUFFS and iCUFFS

Finally the instrumented application is assembled and the binary is loaded through a secure “loader” into the application processor of the MPSoC using a secure key (same random key that is built into the base architectural configuration as shown in Figure 7). Every basic architectural configuration as well as the secure loader that goes with it are built with a different random hardware key.

We also refer to places where special instructions *tCUFFB*, *iCUFFB* and *iCUFFE* are inserted as “check-points” in this paper. All the instructions, *tCUFFB*, *iCUFFB* and *iCUFFE* are hardware instructions that write to a FIFO queue when executed.

VI. Hardware Design Flow

The hardware design flow for both tCUFFS and iCUFFS is shown in Figure 7. Examining the tCUFFS approach, it can be seen that we start with a base processor configuration that is customized in two stages to finally obtain either an application processor or a MONITOR processor. The first stage involves the processor’s customization using the existing pool of resources labelled from A to F in Figure 7. There are many other pre-built resources available in the Xtensa toolset, which the base processor can be customized with [49]. However, we only show six, labelled from A to F in Figure 7 due to space limitations.

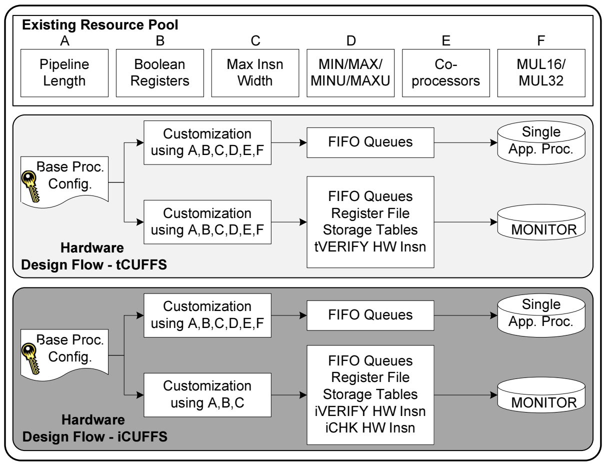

Existing Resource Pool
A
B
C
D
E
F
Pipeline Length
Boolean Registers
Max Insn Width
MIN/MAX/ MINU/MAXU
Co-processors
MUL16/ MUL32
Base Proc. Config.
Customization using A,B,C,D,E,F
FIFO Queues
Single App. Proc.
Customization using A,B,C,D,E,F
FIFO Queues Register File Storage Tables tVERIFY HW Insn
MONITOR
Hardware Design Flow - tCUFFS
Base Proc. Config.
Customization using A,B,C,D,E,F
FIFO Queues
Single App. Proc.
Customization using A,B,C
FIFO Queues Register File Storage Tables iVERIFY HW Insn iCHK HW Insn
MONITOR
Hardware Design Flow - iCUFFS

Fig. 7. The hardware design in the proposed framework

A. Hardware Customizations - tCUFFS

For the tCUFFS framework, the MONITOR processor is customized the same way as the application processors, i.e., it is homogenous to the application processors. The second stage of customization for tCUFFS involves defining custom hardware instructions. The Xtensa LX2 processor allows users to define custom hardware using TIE language. For the application processor this involves just designing a FIFO queue to be able to send traffic to the MONITOR. For the MONITOR, we define the implementation for the FIFO queues, a register file, storage tables and some hardware logic in the form of tVERIFY instruction. The FIFO queues are designed for communication between application processors and the MONITOR processor. A custom register file is designed to have fast access to data for the hardware instructions. The storage tables are used to store the control flow graphs and minimum and maximum execution time for each basic block for the programs executing in each of the application processors. The tVERIFY instruction is used to perform security checks which are discussed in Section VI-C.

B. Hardware Customizations - iCUFFS

The hardware design flow for iCUFFS is almost entirely similar to tCUFFS as described above except for some minor differences. For the iCUFFS framework, we propose a simple MONITOR processor which is different to the application processors, customized independently by selecting only the features that are needed. For example, as shown in Figure 7, hardware design flow for iCUFFS, we use only resources A, B and C for the MONITOR. Also iCUFFS has two hardware instructions in the form of iCHK and iVERIFY rather than the one in tCUFFS. The design of iCHK and iVERIFY instructions is further discussed in Section VI-C. Another minor difference in iCUFFS compared to tCUFFS is that the storage tables are used for storing the number of instructions rather than the execution time limits of BBs in the application processor.

C. Runtime Functionality

At runtime, the MONITOR processor in tCUFFS and iCUFFS performs checks using Algorithm 1 with the exception of one or two lines. The tCUFFS framework does not use lines 9 and 10 in Algorithm 1 and the iCUFFS framework does not use line 8 in Algorithm 1.

Algorithm 1 The algorithm employed by the MONITOR for tCUFFS and iCUFFS

```
1: Initialize $error = 0$, $done = 0$;
2: while (($error == 0$) AND ($done == 0$)) do
3:    for $j = 1$ to $N$ do
4:       if (FIFO$_{P_j}$ not EMPTY) then
5:          Read and Decrypt FIFO$_{P_j}$ Information
6:       end if
7:    end for
8:    tVERIFY($error$, $done$);          // tCUFFS only
9:    iVERIFY($error$, $done$);          // iCUFFS only
10:   iCHK();                            // iCUFFS only
11: end while
```

We have N identical hardware units representing tVERIFY instruction for tCUFFS and iVERIFY instruction for iCUFFS for each of the N application processors on the MPSoC. The N identical repeated hardware blocks allow fast computation of the $error_N$ and $done_N$ signals. The overall *error* signal is computed based on a logical **OR** operation of the individual $error_N$ signals and the *done* signal is changed to '1' when the final processor finishes execution.

1) Runtime Functionality - tCUFFS: As described in Algorithm 1, all the FIFOs are checked for data. If the data is available in any of the FIFOs, it is read and decrypted through the hardware instruction tVERIFY. The tVERIFY is a *Single Instruction Multiple Data* (SIMD) instructions, which updates *error* to 1 if any of the processors fail any of the checks and also updates *finish* to 1 if the application has ended. The tVERIFY instruction performs timing and control flow checks for the tCUFFS framework as shown in Figure 8.

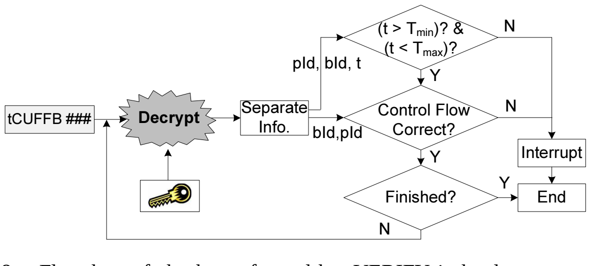


Fig. 8. Flowchart of checks performed by tVERIFY in hardware

The encrypted number in the tCUFFB instruction (communicated to the MONITOR through FIFO) 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 processor. The time information refers to the execution time of the basic block 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 and a timing error **tCUFFS_TIE** generated. The control flow check ensures the transition to the current basic block from the previous basic block is valid and if it isn't, a control flow error **tCUFFS_CFE** is generated. The system exits once the final processor has finished execution.

2) Runtime Functionality - iCUFFS: The iVERIFY instruction like the tVERIFY instruction is a SIMD hardware instruction, which updates the *error* to 1 if any of the processors fail any of the checks and also updates *finish* to 1 if the application has ended. The iVERIFY instruction performs checks for instruction count and control flow generating errors **iCUFFS_ICE** and **iCUFFS_CFE** respectively as shown in Figure 9.

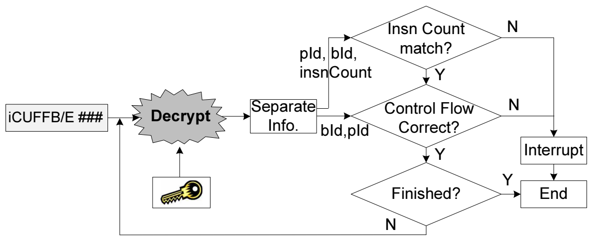


Fig. 9. Flowchart of checks performed by iVERIFY in hardware

The encrypted information in the iCUFFB instruction is decrypted using the hardware key in the MONITOR. The decryption results in the instruction count (insnCount), processor Id (pId) and the block Id (bId) information. The pId information is used to correctly index the hardware tables of the appropriate application processor. The insnCount information is used to check whether the instruction count reported by the iCUFFB/E instruction matches the record in the hardware table. The bId information is used to check the control flow of the application processor pId by matching against the hardware table. The mismatch in the insnCount generates an **iCUFFS_ICE** error. The violation in the control flow generates an **iCUFFS_CFE** error. Once the final processor has finished execution, the MONITOR processor ceases execution.

The iCUFFS framework employs another hardware instruction iCHK which makes the MONITOR processor "active". The internal architecture of iCHK hardware instruction is shown in Figure 10, which checks for timeout error of application processors. The iCHK instruction generates a time out error signal **iCUFFS_TOE** if any of the application processors missed reporting any of the check-points.

The iCHK hardware block sends out a signal *S* to all the application processors and obtains the value of the application processor's **IC**. The iCHK hardware is also aware of the last received **IC** which is available in **prevIC** and the last received BB information, available in **prevBB**. The **prevBB** is used to index into the instruction count hardware table and the table entry is compared to the difference of **IC** and **prevIC**. If the difference is greater than the table entry, a **iCUFFS_TOE=1** is generated indicating that the application processor has likely missed out reporting on a check-point due to an attack, otherwise **iCUFFS_TOE=0** is generated.

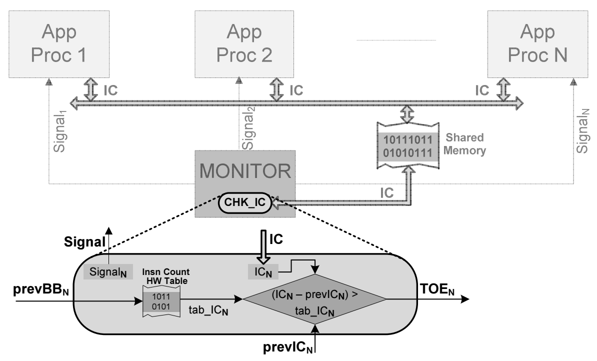
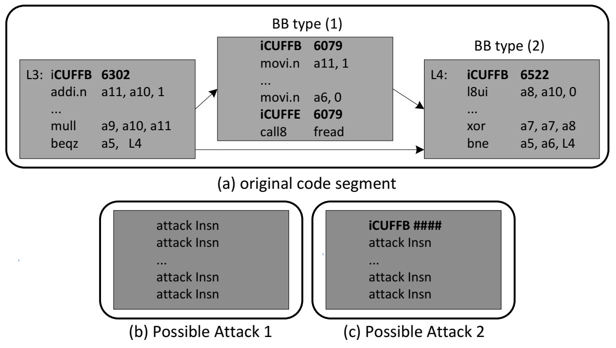


Fig. 10. The hardware logic for the iCHK instruction



```
L3: iCUFFB 6302
    addi.n a11, a10, 1
    ...
    mull   a9, a10, a11
    beqz   a5,  L4

    iCUFFB 6079
    movi.n a11, 1
    ...
    movi.n a6, 0
    iCUFFE 6079
    call8  fread

L4: iCUFFB 6522
    l8ui   a8, a10, 0
    ...
    xor    a7, a7, a8
    bne    a5, a6, L4
```

```
attack Insn
attack Insn
...
attack Insn
attack Insn
```

```
iCUFFB ####
attack Insn
...
attack Insn
attack Insn
```

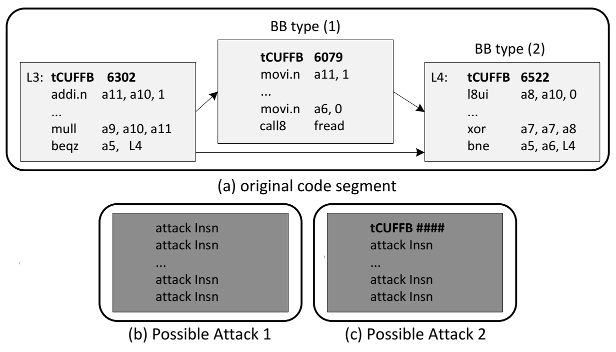
Fig. 12. Possible attacks on BBs in iCUFFS

VII. System Protection Mechanisms

In this section, we discuss how the tCUFFS and the iCUFFS architectural frameworks can be used for the protection of an MPSoC system. Errors are indicated using **tCUFFS_TIE** (Timing Error), **tCUFFS_CFE** (Control Flow Error) for tCUFFS and **iCUFFS_CFE** (Control Flow Error), **iCUFFS_ICE** (Instruction Count Error) and **iCUFFS_TOE** (Time Out Error) for iCUFFS. If any of these are active, the execution of all the application processors on the MPSoC is aborted. The MPSoC security is said to be compromised if any of the application processors is under attack.

The tCUFFS and the iCUFFS framework monitor for security at the BB level. If we can ensure that each BB execution was correct in terms of the properties for it is being checked, we can extrapolate that the entire program execution was correct for those properties. We classify the BBs into three types: (1) ending in a system call; (2) ending in a CFI; and (3) ending in neither a system call nor a CFI. We show in this section that tCUFFS and iCUFFS are able to detect the attacks for each of the three types of BBs and hence secure the application.

We now discuss how the tCUFFS and the iCUFFS frameworks would respond to attacks. Figure 11(a) and Figure 12(a) show an example of a code segment in tCUFFS and iCUFFS respectively. Figure 11(b) and Figure 12(b) show an attack where the attack code does not communicate to the MONITOR and Figure 11(c) and Figure 12(c) show an attack where the attack code does communicate to MONITOR using *tCUFFB* and *iCUFFB* instructions respectively.



```
L3: tCUFFB 6302
    addi.n a11, a10, 1
    ...
    mull   a9, a10, a11
    beqz   a5,  L4

    tCUFFB 6079
    movi.n a11, 1
    ...
    movi.n a6, 0
    call8  fread

L4: tCUFFB 6522
    l8ui   a8, a10, 0
    ...
    xor    a7, a7, a8
    bne    a5, a6, L4
```

```
attack Insn
attack Insn
...
attack Insn
attack Insn
```

```
tCUFFB ####
attack Insn
...
attack Insn
attack Insn
```

Fig. 11. Possible attacks on BBs in tCUFFS

Consider a scenario when BBs of type (1) in Figures 11(a) and 12(a) are attacked by attack code shown in Figures 11(b) and 12(b) respectively. The attack would only be detected in tCUFFS causing a **tCUFFS_TIE** error when the next communication to MONITOR takes place. However, for iCUFFS, there is a **iCUFFS_TOE** because the BB with *SID* 6522 was supposed to follow the BB with *SID* 6079. Since the attack code does not communicate to the MONITOR, the MONITOR detects through the iCHK unit that a "check-point" was missed. Similar result would be achieved for BBs of type (2) in Figures 11(a) and 12(a), when attacked with attack code shown in Figures 11(b) and 12(b) respectively.

Now we consider the type of attack shown in Figures 11(c) and 12(c) on BBs of types (1) and (2) in Figures 11(a) and 12(a), where the attack code does communicate to the MONITOR. We can see that the attack block contains the correct *tCUFFB* and *iCUFFB* instruction types to be able to attack the basic blocks with *SID* 6079 and 6522 in Figures 11(a) and 12(a) respectively. However, because the *SID* number is encrypted, the MONITOR will cause a **tCUFFS_CFE** for tCUFFS and **iCUFFS_CFE** for iCUFFS when the *SID* is decrypted to an unknown value that will cause the control flow check to fail.

When a BB of type 3 faces the attack scenarios mentioned above, it would behave similarly to that shown by a BB of type 2. Both type 2 and type 3 BBs have only one *tCUFFB* or *iCUFFB* instruction per BB, which is the first instruction in that BB.

In an MPSoC architecture, there is often inter-processor communication because of an application executing on multiple processors. An encrypted *SID*, comprising of a processor ID and a basic block ID was discussed earlier in Section V. Encryption of SID is important to prevent information leakage through bus monitoring (one of the types of side-channel attacks) during inter-processor communication. Using bus monitoring, an attacker may be able to reconstruct the control flow of the program on different application processors. With an encrypted SID, the attacker would not be able to decipher the processor number and the basic block number and hence reconstruct the control flow. Moreover, the same bus used by all the application processors to communicate to the MONITOR further complicates deciphering the processor number and basic block number from the encrypted SID obtained from bus monitoring.

VIII. Experimental Results

We tested the tCUFFS and the iCUFFS architectural frameworks using Xtensa LX2 processor from Tensilica Inc. The framework was tested using three multiprocessor multimedia benchmarks (JPEG Encoder, MP3 and JPEG Decoder) of varying complexities. These multiprocessor benchmarks were obtained from the authors of [50] and [51] who had previously partitioned these benchmarks using Tensilica toolset. The details of the processor cores designed for testing each of the three benchmarks is shown in Table I.

The first column of Table I shows the benchmark that was tested. The second column states the type of processor, either Application (App) or MONITOR (MON). The third column states the number of application processors or MONITOR processors in the MPSoC system. The fourth column lists the type of technology used for each of the processor cores. The fifth and the sixth columns state the individual core speed for tCUFFS and iCUFFS respectively. The

TABLE I
PROCESSOR CORE CONFIGURATIONS FOR MPSOCS

Bench-mark	Proc. type	No. of Proc.	Tech-nology (nm)	Speed (MHz)		Power (mW)	
				tCUFFS	iCUFFS	tCUFFS	iCUFFS
JPEG Enc.	App.	6	130	303	303	333.06	335.58
	MON	1	130	303	332	55.51	40.52
MP3	App.	5	90	533	533	673.05	678.35
	MON	1	90	533	585	134.61	93.34
JPEG Dec.	App.	5	90	533	533	632.25	637.55
	MON	1	90	533	585	126.45	93.34

seventh and the eight columns outline the power statistics for tCUFFS and iCUFFS respectively. In the case of application type processors the power figures are a collective statistic for all the application processors on the MPSoC, whereas for the MONITOR processor the power refers only to the one MONITOR on the core.

We investigated two types of designs for the MONITOR in tCUFFS and iCUFFS frameworks. In the tCUFFS framework, we used a MONITOR that was similar to the application processors on the MPSoC, where as in iCUFFS framework we used a MONITOR that was heterogeneous to the application processors by using only the minimal required features. Table I shows that this saves power because the total power for iCUFFS for all the benchmarks is lower than the total power for tCUFFS. For example the JPEG Encoder benchmark, total power consumption of MPSoC, for tCUFFS is 388.57 mW and for iCUFFS is 376.1 mW. The iCUFFS framework also has a higher frequency for the MONITOR compared to the tCUFFS framework and this is again a result of using a simpler MONITOR. This would allow the FIFO communication from the application processors to be processed at a faster rate, as long as the MONITOR processor was clocked separately.

A. Performance Impact

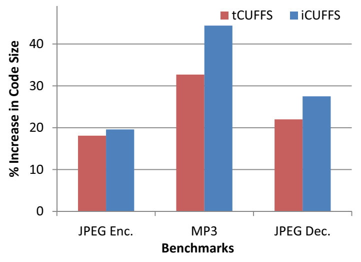
The performance overheads for tCUFFS and iCUFFS resulting from the tests on the multimedia benchmarks are shown in Figure 13. The JPEG encoder benchmark has performance overheads of less than 1% whereas the MP3 and the JPEG decoder which are more complicated benchmarks have higher performance overheads.

Figure 13 clearly shows that the performance overheads resulting from the iCUFFS framework are slightly higher than the tCUFFS (which has the least performance overhead among previously proposed methods in [19] and [31] for detecting software attacks on MPSoCs). However, for slightly higher performance overheads, iCUFFS provides a security framework that checks every single line of code and has an "active" MONITOR to be able to detect a greater range of attacks than the tCUFFS framework as described in Section VII.

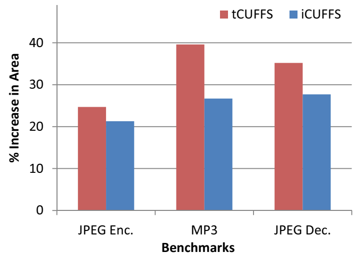
B. Area and Code Size Overheads

The code and area overheads incurred in the MPSoC system due to the tCUFFS and iCUFFS frameworks are shown in Figure 14(a) and Figure 14(b) respectively. iCUFFS has a higher percentage of code overhead compared to tCUFFS (which again has the least amount of code overhead among previously proposed methods in [19] and [31] for detecting software attacks on MPSoCs) as shown in Figure 14(a). The higher code overhead in iCUFFS is a result of employing two special instructions per basic block when a basic block ends in a system call compared to just one in tCUFFS.

The iCUFFS framework, however, has a lower percentage of area overhead than tCUFFS as shown in Figure 14(b). A lower percentage



(a) Code Size



(b) Area

Fig. 14. Percentage Inc. in Code & Area for tCUFFS & iCUFFS

of area is achieved mainly because iCUFFS employs a simpler MONITOR processor with only the required features compared to tCUFFS. It should be noted that we have not accounted for the communication between the application processors and the MONITOR in our area estimation. It is difficult to estimate the area for the communication channels at this high level of abstraction without resorting to place and route methods.

IX. RELIABILITY ANALYSIS

According to the studies on reliability by Schutte et al. in [52] and Ohlsson et al. in [53], between 33% and 77% of all transient faults (soft errors) correspond to CFEs which may be caused due to transient bit flips. The iCUFFS framework was tested for detecting soft errors in the CFIs. Section IX-A details the analysis method and results achieved by the iCUFFS framework. A modified version of the iCUFFS framework is proposed in Section IX-B to ensure reliable inter-processor communication.

A. Fault Injection Analysis

We tested the tCUFFS and iCUFFS frameworks for detecting bit flip errors in the CFIs. We test for bit flips in the CFIs that may occur in the instruction memory of an MPSoC system. Every processor on an MPSoC for a particular benchmark was injected with a set number of faults. The faults in the form of bit flips were injected in the CFIs and it was noted whether the bit flip was detected or not by the MONITOR processor.

Table II shows the analysis of the fault injection tests on each of the three multimedia benchmarks. The first column shows the benchmark and the processor. The second, third and fourth columns show respectively the number of function/library call, branch and jump instructions selected for injecting faults. The fifth column shows the total number of control flow instructions that the faults were injected into. The sixth column shows how many of the faults in the fifth column were detected and the seventh column shows the percentage of faults detected.

Each benchmark was injected with 100 faults or errors. For example, in the JPEG Encoder benchmark, four processors were injected with 17 faults and two processors were injected with 16

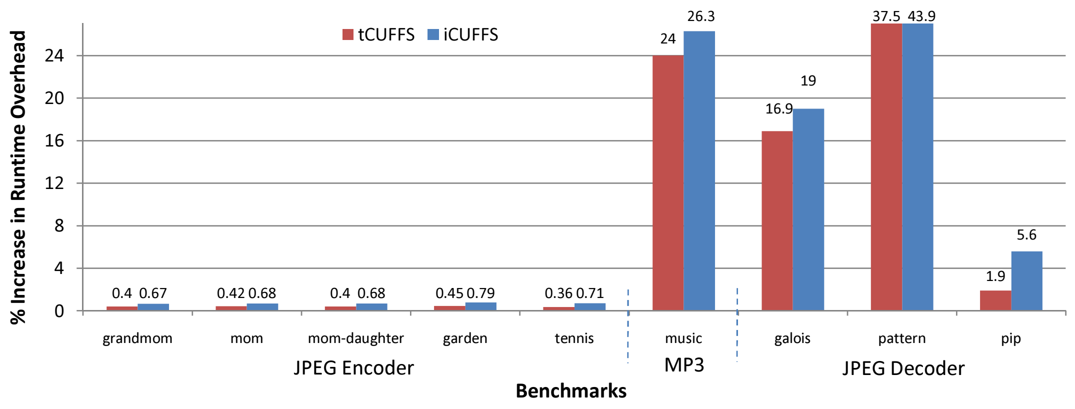


Fig. 13. Performance Overheads for tCUFFS and iCUFFS

faults to get a total of 100 faults. In each of the benchmarks, the CFI and the bit in the CFI, where the fault is to be injected was selected at random. Each processor k has a certain number of CFIs represented by CFI_k. So random numbers were generated between 1 and CFI_k for each k. These numbers were used to identify the CFI in which the fault was to be injected. To determine, which bit of the instruction was to be corrupted, again a random number was generated between 1 and 16 or 1 and 24 depending on whether the instruction was 16 or 24 bits long.

TABLE II
RESULTS FROM FAULT INJECTION FOR ICUFFS ON BENCHMARKS

Processor	Func/ Lib Call	Bra-nch	Jump	To-tal	Dete-cted	% Det-ected
JPEG Enc.						
P1:Read File	9	7	1	17	11	64.7
P2:RGB Convert	0	14	3	17	16	94.1
P3:DCT2	2	15	0	17	17	100
P4:Quantization	8	8	1	17	13	76.5
P5:Huffman	0	14	2	16	9	56.3
P6:Output to File	11	5	0	16	8	50
Total	30	63	7	100	74	74
MP3						
P1:Read File	14	6	0	20	10	50
P2:Polyphase Filtering	10	9	1	20	18	90
P3:Polyphase Filtering	9	10	1	20	18	90
P4:MDCT	9	9	2	20	16	80
P5:Quantization, Encoding, Writeback	7	11	2	20	13	65
Total	49	45	6	100	75	75
JPEG Dec.						
P1:Read and Entropy Decoding	4	14	2	20	11	55
P2:Dequantiz, IDCT for Y,Cb,Cr and Level Shift	6	11	3	20	14	70
P3:Dequantiz, IDCT for Y,Cb,Cr and Level Shift	9	9	2	20	18	90
P4:Dequantiz, IDCT for Y,Cb,Cr and Level Shift	13	3	4	20	11	55
P5:Color Space Conversion, Write Back	9	9	2	20	15	75
Total	41	46	13	100	69	69

Table II shows that 74%, 75% and 69% of the injected single-bit errors in the JPEG Encoder, MP3 and JPEG Decoder benchmarks were detected in our MPSoC system by either the ISS (when the changed instruction does not exist in the instruction set) or the MONITOR at runtime. It should be noted that the CFIs in the system library functions were not tested for fault injection as they are not currently instrumented. System library can also be instrumented in the future to provide detection of software attacks and bit flip errors. In some cases, if the fault injected only changes the opcode of an instruction to another valid opcode, the fault would not be detected. The fault is not detected because the control flow remains valid although the execution is incorrect. For example, if a bit flip changes the opcode referring to **beq** instruction to **bne**, the next basic block executed, although incorrect, is still valid in terms of the control flow. A majority of the undetected faults simply do not lie on the execution path and hence go undetected.

B. Reliable Inter-Processor Communication

The architectural framework proposed earlier in Section IV can be used to achieve soft error detection to ensure reliability of inter-processor and inter-chip communications. However, a minor modification is necessary to facilitate this and achieve a robust framework that can detect soft errors or intentional tampering with the communicated information between two processors or chips.

1) Append an encrypted checksum to the encrypted information *SID* that is being communicated from one processor to the other or one chip to the other.
2) Decrypt the checksum and the information and then confirm that the checksum is valid.

We implemented the above modifications to the iCUFFS framework. The checksum is encrypted and appended to the information by the sender processor and also decrypted and extracted by the receiver processor in hardware. The area overheads shown in Figure 14(b) increased by less than 1% for all the benchmarks. The code overheads remained the same as shown in Figure 14(a). The code overheads do not change because no extra software instructions were added to facilitate the checksum calculation or communication.

The performance overheads further increased by negligible amounts in JPEG Encoder, approximately 0.5% in MP3 and around 4% in JPEG Decoder. The graph in Figure 15 shows the number of times the runtime of the modified iCUFFS (with reliability approach) increases when compared with the iCUFFS approach alone.

X. 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 processors functional unit would need to be changed if there were

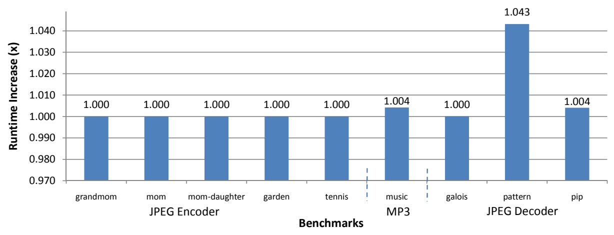


Fig. 15. Increase in Runtime Overhead for iCUFFS with reliability approach compared to iCUFFS only

any changes in the design. This is often an expensive exercise in an ASIP based design.

Related work like CCured in the single processor domain yields performance overheads of up to 150%. The frameworks tCUFFS and iCUFFS for multiprocessor domain, in comparison, achieve about a third of performance overheads of CCured. Therefore, such high performance overheads are common in the research area of security. A conscious effort of the research community is directed at lowering these overheads but it is likely to take some time.

Our frameworks can also handle the case when the execution of a basic block is interrupted in a program. A new signal called *deduct* can be used to identify whenever an interrupt occurs before a basic block has finished execution. The number of cycles or instructions executed in the interrupt service handler (ISH) can be recorded by getting the first and the last instructions of the ISH to record the value of the Cycle Count (CC) register for tCUFFS or Instruction Count (IC) register for iCUFFS. Hence the number of executed cycles or instructions in ISH can be calculated and stored e.g., in COUNT_ISH. Thus, in the case of *deduct* signal being active, the application processor communicates to MONITOR the (CC - COUNT_ISH) value for tCUFFS or (IC - COUNT_ISH) value for iCUFFS. In the normal case, when there is no interrupt, the *deduct* signal would be low and the CC or IC value would be communicated. Context saving on the stack is commonly used for nested interrupts. Therefore, in the case of nested interrupts, the deduct signal as well as the (CC - COUNT_ISH) or the (IC - COUNT_ISH), can be stored on the stack. Finally, the count value before the start of the nested interrupt; the count value during the execution of nested interrupt; and the count value after the execution of nested interrupt should be subtracted from CC (for tCUFFS) or IC (for ICUFFS).

Although we have implemented our framework on Xtensa LX2 processor from Tensilica Inc., the simplicity of algorithm in MONITOR and the simplicity of custom hardware means that the framework can be easily adapted for other MPSoC architectures. Our framework can be scaled to larger systems with many application processors by employing a greater number of MONITOR processors keeping in mind the performance and area constraints of the MPSoC design.

A. Limitations

The limitations of tCUFFS and iCUFFS approaches are as follows:

1) Since our approach provides security solution at the granularity of a basic block, the runtime penalty of the system is dependent on the size of the basic blocks.
2) The control flow transitions of the basic blocks with indirect addressing should be deterministic at compile time or from an execution profile analysis.
3) Our work does not cover data corruption, or any other form of attacks like physical or side-channel attacks.

B. Applications and Future Work

The work discussed in this paper has applicability in modern architectural designs. Although we presented a framework targeting security of MPSoCs in this paper, our work is also relevant in automotive and control systems industries, where hard and soft real time embedded systems are important.

Future work will look at ways to reduce the overheads in the framework. A better static analysis of the code can result in identifying potential point of attacks in the code and then just instrumenting them rather than the entire code. Another possibility of reducing the performance overhead is to pre-compute a hash function for the program counter (PC) and instruction memory (IR) and store the hash function for verification at runtime. However, to perform hashing at runtime, access to PC and IR may be required, which may be unavailable for commercial processors.

The significant area overhead on the MONITOR can be avoided by using one of the application processors as a MONITOR. For the tCUFFS framework, using one of the application processors as a MONITOR is straight forward because all the processors including the MONITOR possess a similar configuration. However, for the iCUFFS framework, the MONITOR must be designed keeping in mind the application that will run on the MONITOR.

Furthermore, a new "symbiosis" architecture can be devised such that two or three processors on the MPSoC can be grouped together. Each processor verifies the execution of one of its fellow processor in the group. The symbiosis architecture nullifies the need for a MONITOR hence reducing the area overhead.

XI. Conclusions

In this paper, we presented two architectural frameworks, tCUFFS and iCUFFS, for protecting against software attacks. Both the frameworks, tCUFFS and iCUFFS, used a dedicated processor for detecting software attacks to detect violations in control flow of the applications. Additionally tCUFFS and iCUFFS respectively used the execution time of basic blocks and instruction count of basic blocks to detect software attacks. We have presented an analysis that shows that our framework can ensure secure execution of programs.

Our results showed that iCUFFS had slightly higher runtime penalty and area overhead compared to tCUFFS. We showed that the iCUFFS framework can be used to detect bit flip errors in the control flow instructions and the tests indicated that approximately 70% of such errors are detected. Finally a modified version of the iCUFFS framework was proposed to ensure reliable inter-processor or inter-chip communication. We believe that our frameworks are scalable and general enough to be applied to other processors for detection of software attacks in MPSoCs.

References

[1] J. Park, H. Song, S. Cho, N. Han, K. Kim, and J. Park, "A real-time media framework for asymmetric mpsoc," in *ISORC '06*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 205–207.
[2] M. Loghi, M. Poncino, and L. Benini, "Cycle-accurate power analysis for multiprocessor systems-on-a-chip," in *GLSVLSI '04*, NY, USA, 2004, pp. 410–406.
[3] W. Wolf, "The future of multiprocessor systems-on-chips," in *DAC '04*, New York, NY, USA, 2004, pp. 681–685.
[4] K. Bhattacharya, S. Kim, and N. Ranganathan, "Improving the reliability of on-chip l2 cache using redundancy," Oct. 2007, pp. 224–229.
[5] S. Ravi *et al.*, "Security in embedded systems: Design challenges." *ACM Trans. Embedded Comput. Syst.*, vol. 3, no. 3, pp. 461–491, 2004.
[6] J. Coburn, S. Ravi, A. Raghunathan, and S. Chakradhar, "Seca: security-enhanced communication architecture," in *CASES '05*. New York, NY, USA: ACM, 2005, pp. 78–89.

[7] R. G. Ragel and S. Parameswaran, “Hardware assisted pre-emptive control flow checking for embedded processors to improve reliability,” in *CODES+ISSS ’06*. New York, NY, USA: ACM, 2006, pp. 100–105.

[8] D. Dagon, T. Martin, and T. Starner, “Mobile phones as computing devices: the viruses are coming!” *IEEE Pervasive Computing*, vol. 03, no. 4, pp. 11–15, 2004.

[9] M. Hypponen, “Malware goes mobile,” *Scientific American*, vol. 295, no. 5, pp. 70–77, 2006.

[10] A. Raghunathan, S. Ravi, S. Hattangady, and J.-J. Quisquater, “Securing mobile appliances: new challenges for the system designer,” *DATE, 2003*, pp. 176–181, 2003.

[11] J. Pincus and B. Baker, “Beyond stack smashing: Recent advances in exploiting buffer overruns,” *IEEE Security and Privacy*, vol. 2, no. 4, pp. 20–27, 2004.

[12] J. Nelißen, “Buffer overflows for dummies,” (http://www.sans.org/reading_room/whitepapers/threats/481.php), 2002.

[13] R. G. Ragel, “Architectural support for security and reliability in embedded processors,” Ph.D. dissertation, School of CSE, UNSW, Sydney, Australia, 2006.

[14] K. Bhattacharya, S. Kim, and N. Ranganathan, “Improving the reliability of on-chip l2 cache using redundancy,” Oct. 2007, pp. 224–229.

[15] C. Constantinescu, “Trends and challenges in vlsi circuit reliability,” *IEEE Micro*, vol. 23, no. 4, pp. 14–19, 2003.

[16] S. K. Reinhardt and S. S. Mukherjee, “Transient fault detection via simultaneous multithreading,” in *Proceedings of the 27th annual international symposium on Computer architecture*. ACM Press, 2000, pp. 25–36.

[17] R. W. David Lammers, “Soft errors become hard truth for logic,” *EE Times*, 2004. [Online]. Available: http://www.eetimes.com/showArticle.jhtml?articleID=19400052

[18] G. C. Necula, S. McPeak, and W. Weimer, “Ccured: type-safe retrofitting of legacy code,” in *POPL ’02*, New York, NY, USA, 2002, pp. 128–139.

[19] K. Patel, S. Parameswaran, and S. L. Shee, “Ensuring secure program execution in multiprocessor embedded systems: a case study,” in *CODES+ISSS ’07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. New York, NY, USA: ACM, 2007, pp. 57–62.

[20] 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*, San Antonio, Texas, Jan 1998, pp. 63–78. [Online]. Available: citeseer.nj.nec.com/cowan98stackguard.html

[21] N. Dor, M. Rodeh, and M. Sagiv, “Cssv: towards a realistic tool for statically detecting all buffer overflows in c,” in *PLDI ’03*, NY, USA, 2003, pp. 155–167.

[22] D. Larochelle and D. Evans, “Statically detecting likely buffer overflow vulnerabilities,” 2001, pp. 177–190. [Online]. Available: http://www.usenix.org/events/sec01/larochelle.html

[23] R. Rugina and M. Rinard, “Symbolic bounds analysis of pointers, array indices, and accessed memory regions,” in *PLDI ’00*. New York, NY, USA: ACM, 2000, pp. 182–195.

[24] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, “Its4: A static vulnerability scanner for c and c++ code,” in *ACSAC ’00*. Washington, DC, USA: IEEE Computer Society, 2000, p. 257.

[25] J. Mcgregor *et al.*, “A processor architecture defense against buffer overflow attacks,” 2003, pp. 243–250. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1270612

[26] D. Arora *et al.*, “Secure embedded processing through hardware-assisted run-time monitoring,” in *DATE ’05*, Washington, DC, USA, 2005, pp. 178–183.

[27] M. Milenkovic, A. Milenkovic, and E. Jovanov, “Hardware support for code integrity in embedded processors,” in *CASES ’05*, NY, USA, 2005, pp. 55–65.

[28] R. G. Ragel and S. Parameswaran, “Impres: integrated monitoring for processor reliability and security,” in *DAC ’06*, New York, NY, USA, 2006, pp. 502–505.

[29] N. Nakka, G. P. Saggese, Z. Kalbarczyk, and R. K. Iyer, “An architectural framework for detecting process hangs/crashes.” in *EDCC*, ser. Lecture Notes in Computer Science, M. D. Cin, M. Kaniche, and A. Pataricza, Eds., vol. 3463. Springer, 2005, pp. 103–121. [Online]. Available: http://dblp.uni-trier.de/db/conf/edcc/edcc2005.html#NakkaSKI05

[30] L. Wang and R. K. Iyer, “Count&check: Counting instructions to detect incorrect paths,” in *Workshop on Compiler and Architectural Techniques for Application Reliability and Security (CATARS)*, 2008.

[31] K. Patel and S. Parameswaran, “Shield: A software hardware design methodology for security and reliability of mpsocs,” *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pp. 858–861, June 2008.

[32] V. Narayanan and Y. Xie, “Reliability concerns in embedded system designs,” *Computer*, vol. 39, no. 1, pp. 118–120, 2006.

[33] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, “Robust system design with built-in soft-error resilience,” *Computer*, vol. 38, no. 2, pp. 43–52, 2005.

[34] D. Siewiorek and L. K.-W. Lai, “Testing of digital systems,” in *Proceedings of the IEEE*, vol. 69, no. 10, 1981, pp. 1321–1333.

[35] D. Siewiorek and R. Swarz, *Theory and Practice of Reliable System Design*. Digital Press, 1982.

[36] P. Hazucha and C. Svensson, “Impact of cmos technology scaling on the atmospheric neutron soft error rate,” *Nuclear Science, IEEE Transactions on*, vol. 47, no. 6, pp. 2586–2594, Dec 2000.

[37] C. Hescott, D. Ness, and D. Lilja, “Scaling analytical models for soft error rate estimation under a multiple-fault environment,” Aug. 2007, pp. 641–648.

[38] iRoC Technologies, “New trends and solutions to combat soft error threats to nanometer semiconductors,” White paper, iRoC Technologies Ltd., February 2004.

[39] T. Semiconductor, “Soft errors in electronic memory - a white paper,” White paper, Tezzaron Semiconductor, January 2004, available online (7 pages).

[40] A. Avizienis, “The n-version approach to fault-tolerant software,” *IEEE Trans. Softw. Eng.*, vol. 11, no. 12, pp. 1491–1501, 1985.

[41] A. Hopkins, T. Smith, and J. Lala, “Ftmp - a highly reliable fault-tolerant multiprocessor for aircraft (1978),” *Proceedings of the IEEE*, vol. 66, pp. 1221–1239, October 1978.

[42] G. Reis, D. August, R. Cohn, , and S. Mukherjee, “Software fault detection using dynamic instrumentation,” in *BARC ’06: Proceedings of the Fourth Annual Boston Area Architecture Workshop*, February 2006.

[43] S. Bagchi, Y. Liu, K. Whisnant, Z. Kalbarczyk, R. K. Iyer, Y. Levendel, and L. Votta, “A framework for database audit and control flow checking for a wireless telephone network controller,” in *DSN ’01: Proceedings of the 2001 International Conference on Dependable Systems and Networks (formerly: FTCS)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 225–234.

[44] B. Ramamurthy and S. Upadhyaya, “Watchdog processor-assisted fast recovery in distributed systems,” in *Fifth IEEE Int’l Working Conf. Dependable Computing for Critical Applications*. IEEE Computer Society Press, September 1995, pp. 125–134.

[45] T. Michel, R. Leveugle, and G. Saucier, “A new approach to control flow checking without program modification,” in *Twenty-First International Symposium on Fault-Tolerant Computing. FTCS-21*, June 1991, pp. 334–341.

[46] 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*, San Diego, CA, February 2000, pp. 3–17. [Online]. Available: citeseer.nj.nec.com/wagner00first.html

[47] Y. Younan, W. Joosen, and F. Piessens, “Code injection in C and C++: A survey of vulnerabilities and countermeasures,” Departement Computerwetenschappen, Katholieke Universiteit Leuven, Tech. Rep. CW386, Jul. 2004. [Online]. Available: citeseer.ist.psu.edu/younan04code.html

[48] G. E. Suh and S. Devadas, “Physical unclonable functions for device authentication and secret key generation,” in *DAC ’07*. New York, USA: ACM, 2007, pp. 9–14.

[49] C. Rowen and D. Maydan, “Automated processor generation for system-on-chip,” Tensilica Inc., Tech. Rep., Sept 2001.

[50] S. L. Shee and S. Parameswaran, “Design methodology for pipelined heterogeneous multiprocessor system.” in *DAC*, 2007, pp. 811–816.

[51] J. Wong, A. Ignjatovic, and A. Janapsatya, “Multiprocessor implementation of image compression algorithms,” in *BE Thesis, School of CSE, The University of New South Wales*, 2007.

[52] M. A. Schuette and J. P. Shen, “Processor control flow monitoring using signatured instruction streams,” *IEEE Trans. Comput.*, vol. 36, no. 3, pp. 264–276, 1987.

[53] J. Ohlsson, M. Rimn, and U. Gunneflo, “A study of the effects of transient fault injection into a 32-bit risc with built-in watchdog.” in *FTCS*, 1992, pp. 316–325.