CUFFS: An Instruction Count Based Architectural Framework for Security of MPSoCs

Krutarth Patel†, Sri Parameswaran†, Roshan G. Ragel‡
†School of Computer Science and Engineering, University of New South Wales, Sydney, Australia.
‡Department of Computer Engineering, University of Peradeniya, Sri Lanka.
†{kpate1, sridevan}@ece.unsw.edu.au ‡roshanr@ce.pdn.ac.lk

Abstract—Multiprocessor System on Chip (MPSoC) architecture is rapidly gaining momentum for modern embedded devices. The vulnerabilities in software on MPSoCs are often exploited to cause software attacks, which are the most common type of attacks on embedded systems. Therefore, we propose an MPSoC architectural framework, CUFFS, for an Application Specific Instruction set Processor (ASIP) design that has a dedicated security processor called iGuard for detecting software attacks. The CUFFS framework instruments the source code in the application processors at the basic block (BB) level with special instructions that allow communication with iGuard at runtime. The framework also analyzes the code in each application processor at compile time to determine the program control flow graph and the number of instructions in each basic block, which are then stored in the hardware tables of iGuard. The iGuard uses its hardware tables to verify the applications’ execution at runtime. For the first time, we propose a framework that proverbs the application processors to obtain their Instruction Count and employs an actively engaging security processor that can detect attacks even when an application processor does not communicate with iGuard. CUFFS relies on the exact number of instructions in the basic block to determine an attack which is superior to other time-frame based measures proposed in the literature. We present a systematic analysis on how CUFFS can thwart common software attacks. Our implementation of CUFFS on the Xtensa LX2 processor from Tensilica Inc. had a worst case runtime penalty of 44% and an area overhead of about 28%.

Categories and Subject Descriptors
B.8.2 [Performance and Reliability]: Performance Analysis and Design Aids
General Terms
Design, Performance, Security
Keywords
Architecture, Instruction Count, MPSoC, Attacks, Tensilica

I. INTRODUCTION

A Multiprocessor System-on-a-Chip (MPSoC) has been widely accepted as an architecture for high performance embedded systems [15]. The multimedia devices such as portable music players and cell-phones already deploy MPSoCs to exploit data processing parallelism and provide multiple functionalities [8, 27]. With increased functionalities the complexity of the design increases, and therefore the susceptibility of the system to attacks from adversaries.

Embedded systems designers often do not 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 [22]. 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 [2].

Stack and heap based buffer overflows are the most common type of software attacks [19]. The buffer overflow vulnerabilities in application programs have been exploited since 1988 [14] and still continue to be exploited. On average nearly 10.7% of the vulnerabilities reported by the US-CERT vulnerability reports pertain to buffer overflow attacks. Figure 1 shows the percentage of buffer overflow attacks in each month of 2006 and 2007.

Figure 2 shows an example of a stack buffer overflow attack. 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.

Recent literature suggests that newer security threats targeting portable electronics like mobile phones and music players may pose significant risks [4, 6]. 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 [2, 21].

In this paper, we propose an architectural framework CUFFS for detection of software attacks. We design an MPSoC with a dedicated security processor called iGuard. Each basic block in the application processors of the MPSoC has one or two check-points which are instrumented with a special instruction that reports to iGuard. Our static analysis tool extracts the control flow of the program and the number of instructions between two sequential check-points. Both the control flow, and the number of instructions between two sequential check-points are recorded inside hardware tables of the iGuard. This information is created at compile time, and recorded in the hardware tables at load time.

At runtime, the application processors report to the iGuard using the special instructions as to which basic block they are executing and the value of the processor’s Instruction Counter (IC) register. The iGuard uses the communicated information to check that the control flow is correct and that the number of instructions that were executed from one check-point to the other is in accordance with the information stored in its tables. However, if the iGuard finds that the control flow is incorrect or that the number of 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 this paper is the “active” iGuard processor in our architectural framework. By “active” we mean that...
it checks 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 IC register, the iGuard determines whether or not an application processor has missed reporting at a check-point. If the iGuard 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.

The remainder of the paper is organized as follows. Related Work is presented in Section II. The architectural framework CUFFS is shown in Section III. Sections IV and V explain the software and hardware design flows respectively, with Section VI presenting scenarios of how the framework will protect the MPSoC against attacks. Experimental results are presented in Section VII and discussion and conclusions are presented in Section VIII and Section IX 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 [3, 5, 7, 23]. Dynamic analysis techniques like CCured, proposed in [13] aim to detect errors or attacks at runtime.

Hardware techniques for detecting attacks usually use customized hardware blocks for runtime checks. McGregor et al. proposed a special return address stack (SRAS) in [10] for protecting against buffer overflow attacks while Arora et al. proposed a hardware monitor in [1] that uses the trace of the executing instructions and program addresses for detecting common software and physical attacks. Milenkovic et al. proposed a signature verification unit in [11] that checks the instructions that are fetched from the memory. Ragel et al. proposed a basic block validation scheme in [20] by modifying the processor’s microinstructions. Nakka et al. proposed a processor pipeline modification framework in [12] for detecting a process crash or hang. Wang et al. proposed checking the instruction counter register at the function level in [26] for detecting incorrect execution paths in programs. Mao et al. proposed a hash-based monitoring approach using a hardware monitor in [9].

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 in [3], 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 in [13] 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 in [10] and the hardware monitor in [1] are not scalable for commercial processors like Xtensa LX2 due to unavailability of a special stack required for [10] and access to the executed instructions (at runtime) required for [1]. Access to the instruction register (IR) is also unavailable in Xtensa LX2 and hence signature verification proposed in [11] is not possible. The microinstructions modification required for [20] and the pipeline modification required for [12] are also not possible due to the unavailability of the base processor’s hardware description. The approach proposed in [26] 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. The approach in [9] is proposed only for monitoring a single processor and may have additional overheads while monitoring multiple processors.

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.

A software solution and two hardware-based solutions for detecting software attacks in the multiprocessor domain are discussed in [18], [16] and [17] respectively.

Our work differs from the previous work proposed in the multiprocessor domain in the following ways. Our work uses the number of executed instructions compared to the use of execution time in clock cycles as proposed in [16–18], to verify correct execution between two check-points in an application program. We therefore know the exact number of instructions that must be executed from one check-point to the next compared to the time reliant methodology proposed in [16–18], which employed a range of execution times.

The approaches in [16–18], all 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 work proposes an “active” processor that probes all the application processors on the MPSoC by regularly reading their IC for security checks. Hence our work even detects attacks that can hijack the processor for executing malicious code and never communicate with the security processor whereas none of the approaches proposed in [16–18] could detect such attacks.

The work proposed in [16–18] 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). Our work 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 our work neither needs any execution trace analysis nor does it need to resort to any estimation.

Our work targets software attacks on an MPSoC architecture that may aim to subvert the control flow of the user’s application and instead 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 we target in this work.

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 our design tool. We also assume that iGuard can be completely secured. This is a reasonable assumption given that iGuard is a dedicated processor for security and only runs a loop that executes the customized hardware instructions. These small number of instructions can be easily placed in a ROM as the instructions need not change.

A. Contributions

The contributions of this paper are as follows:

1) For the first time, an architectural framework is proposed that employs a dedicated processor iGuard that “actively” monitors the application processors to detect software attacks on an MPSoC. Therefore the framework allows detection of attacks even when the application processors do not communicate with the iGuard.

2) Our framework uses the exact number of instructions in a basic block (which can be statically determined), for security checking at runtime. Hence the iGuard can determine with certainty whether or not an attack has taken place. The framework does not rely on the execution profile of the program to gather security information for runtime checks.

B. Limitations

The limitations of our approach 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.

III. ARCHITECTURAL FRAMEWORK

We use Xtensa LX2 processor from Tensilica Inc. for implementing the proposed framework. 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. and also 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. It also allows defining custom register files and storage tables for constants.

The layout of the CUFFS architectural framework for an MPSoC is shown in Figure 3. Every basic block in all the application processors on the MPSoC is instrumented with at least one special instruction that allow them to communicate with a dedicated security processor called iGuard. The special instructions are FIFO queue instructions that allow the application processors to communicate to iGuard through INS_FIFO (designed using Xtensa LX2’s ports and queue interface) as shown in Figure 3. Using the communicated information, the iGuard at runtime, checks whether the applications’ control flow is correct and also the number of instructions that were executed in each basic block. Any application processor’s failure in either of these checks causes the iGuard to send an interrupt to all the application processors to abort execution which is shown using the $I_N$ signals in Figure 3.

Besides these checks, the iGuard also uses a hardware unit called CHK_IC that probes all the application processors to obtain their Instruction Count through a shared memory interface as shown in Figure 3. The CHK_IC allows the iGuard to detect an attack even in the case of an application processor being hijacked by an attacker. The CHK_IC’s active probing of the application processors allows the iGuard 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.

### IV. Software Design

The software design flow used by our framework is shown in Figure 4. 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 5(a). Once the program is divided into BBs, static analysis is performed to yield a control flow graph (CFG) of the program at the BB level which is shown in Figure 5(b). The static analysis also generates the statistics about the number of instructions in each BB.

![Fig. 3. The design of the CUFFS architectural framework](image)

![Fig. 4. The software design in the proposed framework](image)

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. This SID is then encrypted using a distinct encryption key (based upon physical uncloneable functions (PUF), proposed in [25], to acquire an encryption key using the physical properties of integrated circuits in the MPSoCs). An exact copy of the encryption key is also stored in hardware to decrypt the SID at runtime.

Each BB is then instrumented with one or two special instructions as shown in Figure 5(c) by our automated static analyzer. A special iBeatB instruction is added as the first instruction in each BB. A BB that ends with a system call instruction is instrumented with two special instructions, iBeatB and iBeatE as shown in the second BB box in Figure 5(c). The number in the iBeatB and iBeatE 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 5(c). An extra label is inserted after the iBeatB instruction and the target of the branch is changed to this extra label. This type of instrumentation allows the iBeatB instruction to be executed only once per loop, thus reducing communication overhead.

The information regarding the control flow of the program and the number of instructions in each basic block are stored in hardware tables. These tables will be used at runtime by iGuard for security checking. We also refer to places where special instructions iBeatB and iBeatE are inserted as “check-points” in the paper. Both the iBeatB and iBeatE are hardware instructions that write to a FIFO queue when executed. The special features of the FIFO queue are detailed in Section V.

### V. Hardware Design

The hardware design flow employed by our framework is shown in Figure 6. We start with a base processor core configuration that is customized in two stages. It must be noted that our framework is concerned with integrating an iGuard processor on the MPSoC and that the customizations that lead to application processors on the MPSoC are according to the user’s specifications.
The second stage of customization involves defining custom hardware instructions. The Xtensa LX2 processor allows users to define custom hardware using TIE language. We define implementation for the INS_FIFO queues, a register file, storage tables and some hardware logic. The INS_FIFO queues are designed so that the application processors on the MPSoC can communicate to the iGuard and a custom register file is designed to have faster access to data for the hardware instructions. The storage tables are used to store the control flow graphs and number of instructions per basic block for the programs in each of the application processors. The CHK_IC and GUARD hardware instructions are used to design some runtime security checks which are further discussed in Section V-A. Finally, these customizations yield an iGuard processor that can be used for detection of software attacks in the application processors on an MPSoC as shown in Figure 6.

A. Runtime Functionality

The iGuard processor at runtime employs a fixed set of instructions to check the control flow (CF), the instruction count (IC) and a timeout (TO) for processors that have missed reporting at any of their check-points. The algorithm employed by the iGuard is shown in Algorithm 1.

Algorithm 1 The algorithm employed by iGuard for security

```plaintext
Initialize error = 0, done = 0;
while ((error == 0) AND (done == 0)) do
  for j = 1 to N do
    if (INS_FIFO[j] not EMPTY) then
      Read and Decrypt INS_FIFO[j] Information
    end if
  end for
  GUARD(error, done);
  CHK_IC;
end while
```

The Algorithm 1 does three things: (1) Reads the incoming INS_FIFO queue if they are not empty; (2) Executes the GUARD hardware instruction; and (3) Executes the CHK_IC hardware instruction.

If the INS_FIFO contains information, it is read and decrypted using the hardware key as shown in Figure 7. The resulting information is stored in a storage state FIFOINFO of iGuard.

Since both (2) and (3) are hardware instructions, they are executed in parallel. We discuss (2) first which is the GUARD instruction. The runtime logic that allows the GUARD instruction in Algorithm 1 to compute the error signal is shown in Figure 8. The iGuard uses the FIFOINFO to know the BBs that are currently executing (currBB) in the application processors, the current instruction count value (currIC) in the application processor and also the type of FIFO information (currCode). The iBeatB and iBeatE have currCode of 0 and 1 respectively. The last iBeatE instruction that an application processor sends has a currCode of 3 and if the FIFOINFO does not contain new information, the currCode is 2.

The prevBB, prevIC and prevCode are all storage states that are designed to hold the previous BB, IC and type of iBeat instruction that were read from the INS_FIFO. All the three storage states prevBB, prevIC and prevCode are updated from currBB, currIC and currCode respectively, which is shown by three dotted lines with arrows at the end in Figure 8. The updating takes place at the end of the hardware instruction after the error signal has been computed from the logic shown in Figure 8. The possible values of prevCode is 0 or 1 because 2 denotes that no new information was communicated and 3 denotes that the execution has finished in a particular application processor.

The Insn Count Check uses the currBB to index into the hardware table for instruction counts and obtain the number of instructions for currBB. It then compares the table entry with the difference of currIC and prevIC. If the table entry does not match exactly an error signal instruction count error ICE=1 is generated, otherwise ICE=0 is generated.

Similarly, the Control Flow Check uses the prevBB to index into the hardware table for control flow and obtain the possible transitions from the basic block prevBB. If the currBB does not exist in the hardware table for control flow, as one of the valid transitions from prevBB, then an error signal CFE=1 is generated, otherwise CFE=0 is generated.

Since not all the checks are valid at all the time, appropriate signals are selected from the MUX based on the value of firstly the prevCode and then the currCode. For example, if the FIFOINFO received was an iBeatB instruction followed by another iBeatB instruction; which is prevCode=0 and currCode=0, both the ICE and CFE must be checked. However if an iBeatB instruction (prevCode=0) followed by an iBeatE instruction (currCode=1) was received, only the ICE must be checked.

If no information was received through the INS_FIFO for a particular application processor, i.e., currCode=2, the time out error signal TOE generated from the CHK_IC hardware instruction is selected from the MUX as the errorN. The block diagram of the logic used in the hardware of CHK_IC instruction is shown in Figure 9.

The CHK_IC hardware block sends out a signal S to all the application processors and obtains the value of the application processor’s IC. The CHK_IC 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 TOE=1 is generated indicating that the application processor has likely missed out reporting on a check-point due to an attack, otherwise TOE=0 is generated.

We have N identical hardware units shown in Figure 8 for each of the N application processors on the MPSoC to allow fast computation of the errorN and doneN signals. The overall error signal is computed based on a logical OR operation of the individual errorN signals and the done signal is changed to 1 when the final processor finishes
The attack scenario in which the attacker’s BB does not communicate with iGuard using the iBeat instructions is shown in Figure 10(b) (for BB type 1) and Figure 11(b) (for BB type 2). The BB with SID 6079 or 6522 was supposed to follow the BB with SID 6302. But since the attacker’s BB does not use an iBeat instruction to report to the iGuard, the iGuard will infer an error due to a timeout, generating a TOE. The works in [17] and [16] would not be able to detect such an attack as their approaches are reactive, i.e., would only respond if the application processor(s) initiated the communication. Our work easily detects such an attack since the iGuard is proactive and is alerted if it does not receive any communication from the application processor(s).

The attack scenario in which the attacker’s BB does communicate with iGuard using the iBeat instructions is shown in Figure 10(c) (for BB type 1) and Figure 11(c) (for BB type 2). The attacker employs the iBeat instruction as the first instruction in the BB. Since the iBeat instructions have the SID which is encrypted, the iGuard will cause a CFE when the SID is decrypted to an unknown value that will cause the control flow check to fail.

The iBeat instruction in the attack scenario in Figure 10(c) and Figure 11(c) can also be employed by the attacker such that it is not the first instruction in the BB. When the iBeat instruction is not used as the first instruction in the BB, it would fail the instruction count check when it does communicate to the iGuard, causing an ICE. However, it is quite likely that a TOE would be generated before the ICE, since the CHK_QC module of the “active” iGuard processor will detect that a reporting instruction was missed.

When a BB of type 3 faces the attack scenarios mentioned above, it would have an exact same behavior as that shown by a BB of type 2. Both type 2 and type 3 BBs have only one iBeat instruction per BB, which is the first instruction in that BB. Thus we have shown that using the CUFFS architectural framework with the “active” iGuard processor can be used to detect attacks at the basic block level.

The attack scenario in which the attacker’s BB does not communicate with iGuard using the iBeat instructions is shown in Figure 10(b) (for BB type 1) and Figure 11(b) (for BB type 2). The BB with SID 6079 or 6522 was supposed to follow the BB with SID 6302. But since the attacker’s BB does not use an iBeat instruction to report to the iGuard, the iGuard will infer an error due to a timeout, generating a TOE. The works in [17] and [16] would not be able to detect such an attack as their approaches are reactive, i.e., would only respond if the application processor(s) initiated the communication. Our work easily detects such an attack since the iGuard is proactive and is alerted if it does not receive any communication from the application processor(s).

The attack scenario in which the attacker’s BB does communicate with iGuard using the iBeat instructions is shown in Figure 10(c) (for BB type 1) and Figure 11(c) (for BB type 2). The attacker employs the iBeat instruction as the first instruction in the BB. Since the iBeat instructions have the SID which is encrypted, the iGuard will cause a CFE when the SID is decrypted to an unknown value that will cause the control flow check to fail.

The first column of Table I shows the benchmark that was tested. The second column states the type of processor, either application or iGuard. The third column states the number of application processors or iGuard processors in the MPSoC system. The fourth column lists the type of technology used for each of the processor core and the fifth column states the individual core speed. The sixth and seventh columns outline the power and area statistics. In the case of application type processors the power and area figures are a collective statistics for all the application processors on the MPSoC whereas for the iGuard processor type the power and area refer only to the iGuard processor.

We have designed the iGuard processor with only the minimal required features such that its clock speed is higher than the other application processors on the MPSoC. This would allow the INS_FIFO communication from the application processors to be processed at a faster rate, as long as the iGuard processor was clocked separately.

**A. Performance Overheads**

The performance overheads resulting from the tests on the three multimedia benchmarks are shown in Figure 12. 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.

We also compared our result with the approach proposed in [16] which has the least amount of performance overhead among all previously proposed methods for detecting software attacks on MPSoCs. We also refer to the work in [16] as “LOCS” in Figure 12 and Figure 13.

Figure 12 clearly shows that the performance overheads resulting from our architectural framework are only slightly higher than the method proposed in [16]. However, for slightly higher performance overheads, our approach provides a security framework that can detect a greater range of attacks compared to [16] and [17] as described in Section VI.

![Table I](image-url)

**TABLE I**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Processor Type</th>
<th>No. of Proc.</th>
<th>Technology (MHz)</th>
<th>Power (mW)</th>
<th>Area (mm²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JPEG Enc.</td>
<td>Application</td>
<td>6</td>
<td>130</td>
<td>3.55, 38</td>
<td>3.58</td>
</tr>
<tr>
<td>iGuard</td>
<td>1</td>
<td>130</td>
<td>3.55</td>
<td>0.72</td>
<td></td>
</tr>
<tr>
<td>MP3</td>
<td>Application</td>
<td>5</td>
<td>90</td>
<td>5.83, 35</td>
<td>2.08</td>
</tr>
<tr>
<td>iGuard</td>
<td>1</td>
<td>90</td>
<td>5.83</td>
<td>0.53</td>
<td></td>
</tr>
<tr>
<td>JPEG Dec.</td>
<td>Application</td>
<td>5</td>
<td>90</td>
<td>6.37, 35</td>
<td>1.48</td>
</tr>
<tr>
<td>iGuard</td>
<td>1</td>
<td>90</td>
<td>6.37</td>
<td>0.40</td>
<td></td>
</tr>
</tbody>
</table>

The attack scenario in which the attacker’s BB does communicate with iGuard using the iBeat instructions is shown in Figure 10(c) (for BB type 1) and Figure 11(c) (for BB type 2). The attacker employs the iBeat instruction as the first instruction in the BB. Since the iBeat instructions have the SID which is encrypted, the iGuard will cause a CFE when the SID is decrypted to an unknown value that will cause the control flow check to fail.
the application processors in terms of control flow and the number of instructions executed. Our results showed that compared to other hardware-based counter-measures for detecting software attacks on MPSoCs, the runtime penalty for CUFFS was slightly worse although the area overheads were much lower. We believe that our framework is scalable and general enough to be applied to other processors for detection of software attacks in MPSoCs.

REFERENCES