Micro Embedded Monitoring for Security in Application Specific Instruction-set Processors

Roshan G. Ragel
The University of New South Wales (UNSW) and National ICT Australia
Sydney, NSW 2052, Australia
0061-2-93857224
roshanr@cse.unsw.edu.au

Sri Parameswaran
The University of New South Wales and National ICT Australia
Sydney, NSW 2052, Australia
0061-2-93857204
sridevan@cse.unsw.edu.au

Sayed Mohammad Kia
Abbaspour University, Tehran, Iran
(visiting UNSW, Sydney)
0061-2-93857229
kia@pwit.ac.ir

ABSTRACT
This paper presents a methodology for monitoring security in Application Specific Instruction-set Processors (ASIPs). This is a generalized methodology for inline monitoring insecure operations in machine instructions at microinstruction level. Microinstructions are embedded into the critical machine instructions forming self checking instructions. We name this method Micro Embedded Monitoring. Since ASIPs are designed exclusively for a particular application domain, the Instruction Set Architecture (ISA) of an ASIP is based on the application executed. Knowledge of the domain gives an insight into the kinds of the security threats which need to be considered. The fact that the ISA design is based on the application makes room to accommodate security monitoring support during the design phase by embedding microinstructions into the critical machine instructions. Since the microinstructions are the lowest possible software level architecture, we could expect to get better performance by implementing security detection using microinstruction routines. Four different embedded security monitoring routines are implemented for evaluation. The average performance penalty with these monitoring routines with ten different benchmarks is 1.93% while the average area and power overheads are 5.26% and 3.07% respectively.

Categories and Subject Descriptors
B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms
Design, Reliability, Experimentation, Security, Verification.

Keywords
Application Specific Instruction-set Processors, Micro Embedded Monitoring, Microinstructions, Security Monitoring, Self-Monitoring Instructions

1. INTRODUCTION
For the last couple of decades, explosive growth of the Internet has brought an increase in Internet systems being compromised by malicious attacks. Successful attacks could have widespread and devastating consequences due to the increased dependence on networked information systems. Many modern electronic devices, such as personal computers, smart cell phones, Personal Digital Assistants (PDAs), network sensors, routers etc., need to store and compute sensitive information. Integrity and availability as well as secrecy, are crucial security policies, not just for military applications but also for the ever-growing numbers of businesses and individuals who use the internet and networked devices.

Viruses and worms can significantly damage the networked world. Traditionally, software approaches, either via cryptography or compiler techniques [11][37][43] have been utilized as the major defenses against these attacks. But these types of defenses result in significant performance penalty. Recently, a number of architectural approaches have been proposed to deal with the buffer overflow attack [27][37][38][48] which is a very common threat.

Design of embedded systems is constrained by strict requirements, such as latency, throughput, power consumption, area, cost effectiveness and time to market pressures. A well-known challenge during an embedded processor design is to obtain the best possible results for a typical target application domain. A custom instruction set to perform special tasks could result in significant improvements for an application domain [52][53]. In recent years, Application Specific Instruction-set Processors (ASIPs) [10] have gained popularity not only in the research community but also in the production of embedded systems. ASIPs are expected to become one of the most important building blocks of future Application Specific Integrated Circuits (ASIC) designs as they combine the flexibility of software with the energy-efficiency, and scalable computational performance of dedicated hardware implementations [16].

1.1 Motivations
Embedded systems are generally considered more tolerant to security and reliability problems. The nature of their design (smaller memories, lower clock speeds) and very minimal input output interfaces are the contributors for this consideration. But the assumptions about the design nature are becoming invalid as

embedded processor manufacturers are increasingly turning to the latest technologies to achieve low power and reduced cost advantages. The fact that the embedded systems are more likely to be used in safety critical systems and consumer products where reliability and security are important makes reliability and security an issue in embedded system design.

Even though there are architectural mechanisms to provide security support, they are not general enough to accommodate solutions to new threats, and usually require compiler support [38]. Researchers have identified security support as a design parameter for embedded processors [23], just like power consumption and area cost. ASIP designers have a greater advantage since ASIPs are designed by considering a particular application domain. The Instruction Set Architecture (ISA) of an ASIP is decided based on the application that executes on it. The knowledge about the domain gives greater insight into the security threats, allowing security support to be incorporated in the design methodology as a design step or as a design parameter by altering the ISA. This method needs no support from the compiler so that any current and future applications could be adapted to this system without any difficulties.

1.2 Related Work

Architectural monitoring support has been used in error-detection mechanisms to implement control flow checking [25] and data-access faults detection [45]. Most of the previous work on hardware-level security concentrates on implementing tamper-resistance and cryptography [14][19][26][32][33][41][42][46]. In [14], Dyer et. al introduce an IBM co-processor that provides physical tamper-resistance and hardware support for cryptography, and in [32][33], Ravi et. al investigated the effect of using an embedded processor for similar support. Cryptography is used as a secondary protection to secure sensitive data even if the system is compromised in an attack.

Stack based buffer overflow attack is one of the main security threats of recent time [9]. Xu et. al [48] followed by Lee et. al [24] and McGregor et. al [27] independently proposed hardware methods to detect stack-based buffer overflow attacks. Other researchers have enhanced support for buffer overflow attacks and considered the effect on embedded systems caused by this attack [13][23][37][38].

The ultimate goal of security attacks is to gain control of the system and destroy the system integrity by altering information which is in the form of software and data. In [15][28][42][47] authors provide hardware mechanisms to check and detect whether the system integrity has been compromised.

Nakka et. al in [29] proposed a framework called Reliability and Security Engine (RSE), which is an additional hardware unit, used to incorporate any of the above mentioned security detection mechanisms. RSE needs compiler support to insert check instructions into the instruction vector. The monitoring hardware in RSE is external to the core processor and therefore needs a well defined interface with the core processor. Since RSE is a separate hardware unit, it needs self-checking mechanism to ensure that it is not compromised. Xu et. al [49] have extended RSE to include Field Programmable Gate Arrays (FPGA) that will be used to reconfigure the security modules in RSE. Vetteth [50] has implemented two RSE modules in hardware using VHDL and synthesized it onto an FGPA.

This paper proposes a general framework that could be used to accommodate most of the previous inline detection techniques, and any new security monitors by embedding security monitoring mechanism as microinstructions of critical machine instructions. Therefore the need to have compiler support and self-checking the monitoring hardware is not necessary in processors produced by our framework as the monitoring hardware itself is part of the core processor.

1.3 Our Contribution

For the first time we propose a methodology for embedding security monitoring technique within the microinstructions forming self-monitoring instructions[1]. This method would be general enough to accommodate any inline monitoring technique. In our approach, security monitoring support is considered as one of the design parameters of an ASIP, and integrated into the ASIP design phase.

1.4 Microinstruction

Microinstructions are instructions which control data flow, and instruction-execution sequencing in a processor at a more fundamental level than the level of machine instructions. A series of microinstructions is necessary to perform an individual machine instruction [39]. Thus, a microinstruction could be seen as an instruction that activates the circuits necessary to perform a single machine operation by interconnecting the necessary hardware resources [7]. The microinstructions given in this paper are a pseudo set of instructions that we use to show the hardware operations.

```
source0     = GPR.read0(Rn)
source1     = GPR.read1(Rm)
address     = ADDER32.add(source0, source1)
data        = DMAU.load(address)
null        = GPR.write0(Rd,data)
```

Figure 1. Microinstructions for ldr Rd, [Rn,Rm]

Figure shows the sequence of microinstructions which needs to be executed to perform a **ldr Rd,[Rn,Rm]** instruction in an Arm Thumb™ processor. First both (Rn and Rm) the register values are read from the register-file (GPR) and then they are added together using a 32-bit adder (ADDER32) to evaluate the target address. The target address is used to load the data from data memory via data memory access unit (DMAU) and finally the data is written back to the register-file in Rd's location.

The rest of the paper is organized as follows. Section 2 describes the anticipated major security threats to ASIPs, and section 3 presents the detection mechanisms for those threats. Section 4 in this paper describes the design methodology followed by an

[1] Machine instructions those monitor themselves for improper operations via embedding extra microinstructions.

examination of cost overheads in section 5. Finally section 6 concludes the paper.

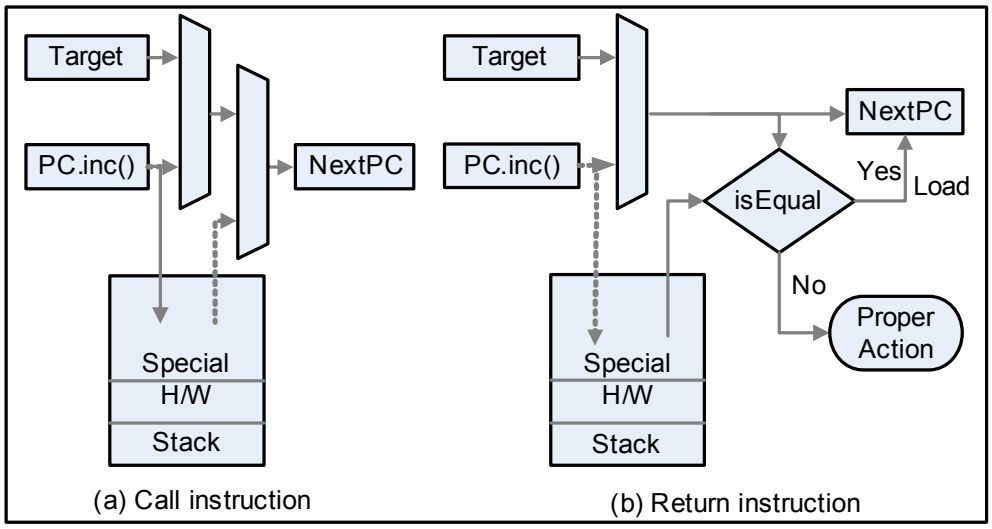


Figure 2. Return Address Stack Monitoring

2. SECURITY THREATS IN ASIPS

In this section we discuss some of the main security threats to embedded processors and therefore to ASIPs. We propose monitoring methods to detect these threats in section 3.

2.1 Buffer Overflow Attack

Even though there are countermeasures in software for buffer overflows, this continues to be one of the main threats for the last 10 years or so [9]. The majority of buffer overflow attacks are related to overwriting the return address in the buffer to an address of a malicious function pointer by which, the control flow is transferred to the attacker. When a call instruction is executed the processor transfers control to the target function and upon completing the procedure, control is returned back to the instruction that follows the call instruction. In a procedure call the last state of the processor before the procedure call is remembered by storing the values in a buffer in a stack fashion. An attacker exploits the weakness of the called procedure by overwriting the return address in the buffer with a value of his choice, to obtain control of the processor.

2.2 Fault Injection Attack

Environmental conditions and external parameters such as temperature, humidity, radiation, and supply voltage are changed to induce faults in an embedded system component. Eventually this attack will make the system non-reliable and therefore the attacker changes a security threat into a reliability problem. Historically, these attacks are performed on very low-end embedded systems such as smart cards [18][21][22][30][44]. But, these attacks are becoming more and more robust such that they could be a security threat to any secure embedded systems.

The following are the main methods in which fault injection attacks compromise the security of an embedded system [32]: (1) availability: disable the availability of a device by occupying it with unwanted data or by damaging it; (2) integrity: damaging the data and application stored in storage devices; and (3) privacy: gaining access to secure data by breaking various cryptographic schemes by taking advantage of random faults [12].

2.3 Data and Software Integrity Attack

Data integrity ensures that data in an embedded system has not been altered or deleted by someone. These attacks could be detected by monitoring unwanted data accesses. Software integrity makes sure that the program in the system is not altered or deleted. We could detect this attack by monitoring whether any unwanted control flow occurs in the system.

3. MICRO EMBEDDED MONITORING

As mentioned previously, monitoring routines to check insecure operations are embedded into machine instructions by the addition of extra microinstructions, forming self-monitoring instructions. A secure micro coding is more reliable, as it is performed in the architecture level and the user cannot overwrite them. They have as little overhead as possible, since the monitoring routines are formed with microinstruction.

When a security threat is detected there is more than one choice to follow: terminate the current process and inform the operating system about the fault by returning a trap signal; continue execution of the program in a safe way; or, stop the process. Recovery mechanisms are beyond the scope of this paper and are not covered in this paper.

The following subsections describe some of the possible security monitoring techniques where micro embedded monitoring is used. Note that these techniques do not compile a complete list of security monitoring and not the only type of monitoring methods for the given threats. These are given as an example list only for evaluation purpose.

3.1 Return Address Monitoring

As shown in Figure 2, return address monitoring is used to verify the return address of a return instruction that is associated with

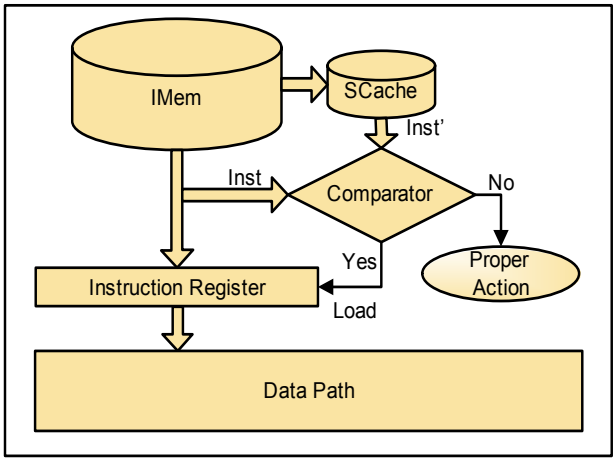
```
current_pc      = PC.read()
return_addr     = PC.inc()
extended_tgt    = EXT.sign(tgt)
tgt_addr        = ADDER32(current_pc, extended_tgt)
null            = PC.write(tgt_addr)
null            = LR.write(return_addr)
null            = RAS.push(return_addr)
```

(a) call instruction (ex: call tgt)

```
next_pc = DMAU.read(sp)
return_addr     = RAS.pop()
checkValid      = COMP.cmp(next_pc, return_addr)
null            = [checkValid] PC.write(next_pc)
```

(b) return instruction (ex: pop pc)

Figure 3. Microinstructions for Return Address Monitoring




```
current_pc      = PC.read()
Inst            = IMem.load(current_pc)
Inst'           = SCache.load(current_pc)
checkValid      = Comparator.cmp(Inst, Inst')
null            = [checkValid] IR.write(Inst)
null            = [checkValid] PC.inc()
```

Figure 4. Fault Injection Detection in Instruction Memory

the last call instruction. The solid lines in Figure 2 are the active lines for a particular operation. During each linked branch (call), a copy of the return address (next to the current PC; indicated by PC.inc()) is stored in a special hardware stack as shown in Figure 2(a). Then the PC is set to the target address (indicated as NextPC in Figure 2(a)). When a return address (indicated as Target in Figure 2(b)) is popped from the memory stack, it is compared against the value stored in the hardware stack using a comparator as shown in Figure 2(b). If they match then the target address is set to the PC. When there is a mismatch, a trap signal is sent to the processor to indicate a possible buffer overflow. More details about this monitoring mechanism could be found in [48].

Figure 3 shows the microinstructions related to return address monitoring. The microinstructions in ***bold-italics*** are the extra microinstructions embedded to perform monitoring. These are inserted into the call and return instructions as illustrated in (a) and (b).

The microinstruction 'RAS.push(return_addr)' will push the return address into a secure stack. When the return instruction is executed, the microinstruction 'RAS.pop()' will pop a copy of the return address from the secure stack and will be compared with the copy from the data memory.

3.2 Fault Injection Detection

We are proposing two monitoring mechanism to detect fault injection. One is to detect faults injected into the instruction memory, and the other to detect faults injected into the data path of an ASIP.

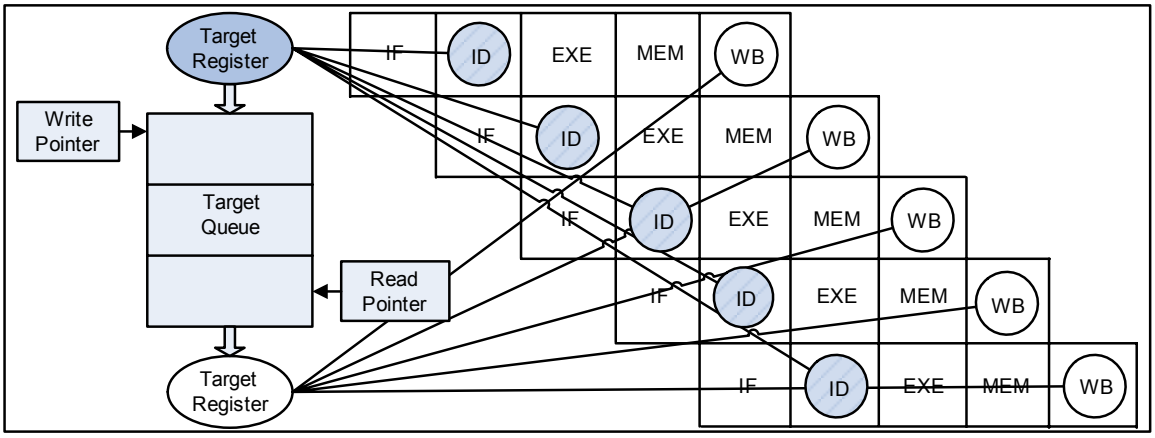
3.2.1 Fault Detection in the Instruction Memory

Figure 4 shows instruction memory monitoring where two copies of a critical instruction are fetched and compared before being written into the instruction register. If a mismatch occurs due to a fault injection in the instruction memory's data bus then the processor is not allowed to proceed further. The fault model used here is based on control flow instructions. Control flow instructions are identified as critical instructions and a copy of all the control flow instructions are stored in a special cache, called *SCache* at load time. When a control flow instruction is fetched a copy of the same instruction is also fetched from the *SCache* and compared to find whether a fault injection has occurred into the control flow instruction stream in the instruction memory.

3.2.2 Data Path

Figure 5 shows, how some of the faults injected into the data path could be detected by keeping track of pipeline registers and monitoring the target register of each instruction. A special hardware target queue is used to store the target registers in the

instruction decode stage of the pipeline and then the same registers are de-queued and checked against the write-back register. Figure 5 (b) and (c) show the microinstructions related to this check. All the machine instructions with a target register are augmented with the set of micro instruction shown in Figure 5(b) and 5(c).



(a) Fault Detection in Datapath

```
target-reg      ← instruction-decode()
pos             = WritePointer.read()
null            = TargetQueue .write(pos, target-reg)
null            = WritePointer.next()
```

(b) Instruction Decode Stage

```
target-reg 0    ← from-pipeline-register
pos             = ReadPointer.read()
target-reg 1    = TargetQueue .read(pos)
validity        = COMP4.cmp(target-reg 0,target-reg 1)
null            = ReadPointer.next()
null            = VALID.write(validity)
```

(c) Register Write Back Stage

Figure 5. Fault Detection in the Data Path

IF	Fetch PUSH	Fetch POP
ID	data = GPR.read()	addr = SP.read()*
EXE	addr = SP.dec-and-read()*	null = SP.inc()
MEM	null = DMAU.write(addr,data)	data = DMAU.read(addr)
WB		null = GPR.write(data)

(a) Micro Instructions of PUSH and POP instructions

```
max     =   Stack-Upper-Limit-Register.read()
min     =   Stack-Lower-Limit-Register.read()
valid0  =   COMP32LE.cmp(max,addr)
valid1  =   COMP32GE.cmp(min,addr)
valid   =   valid0 & valid1
null    =   VALID.write(valid)
```

(b) Micro Instruction for Monitoring

Figure 6. Monitoring Stack Boundaries

3.3 Boundary Checks on Memory Access

Memory access fault detection and faulty indirect jumps are detected by storing the boundary values to special registers at load time. When an indirect[2] memory access or an indirect jump instruction is encountered, the target address is compared with the boundary values to verify that they are either going to access memory, or jump to a memory location within the boundary values.

3.3.1 Data Memory Access

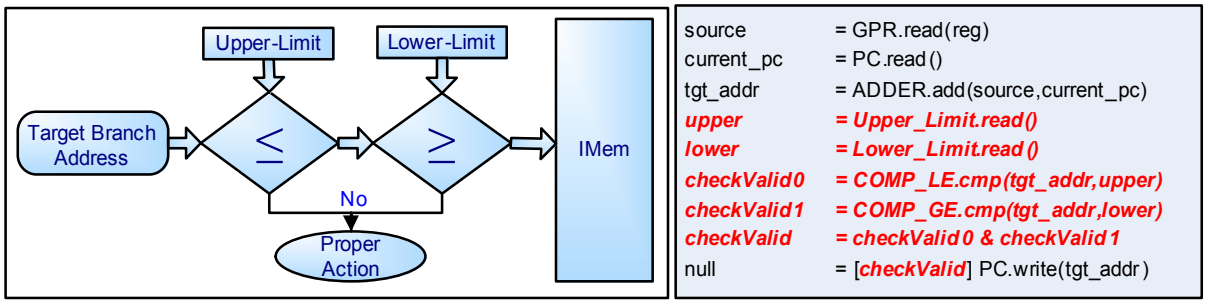
Figure 6 shows the microinstructions for PUSH and POP instructions in (a) and the microinstructions that are added to do boundary check in (b). Figure 6(a) is divided into five rows each of which belongs to a pipeline stage (IF – Instruction Fetch, ID – Instruction Decode, EXE – Instruction Execution, MEM – Memory Read and Write and WB – Register Write Back).

Column two in Figure 6(a) belongs to PUSH instruction and column three to POP. The microinstructions in (b) should be added to locations in (a) where there are asterisks (*), to check stack boundaries. When stack addresses are read from the stack pointer (SP), they are compared against stack boundary values stored in special registers called Stack-Upper-Limit-Register and Stack-Lower-Limit-Register using two comparators COMP32LE and COMP32GE as shown in Figure 6(b). A boundary violation is written to the VALID register and then a proper signal could be sent to the processor. Similar routines could be inserted into all the load and store instructions to detect data memory boundary access violation.

[2] Only indirect values are checked because they will refer to registers which could be dynamically changed.

3.3.2 *Instruction Memory Access*

Figure 7 shows how a jump/branch instruction is monitored. Such an instruction transfers control to an instruction in the instruction memory location which is within the boundary of the current process/program. Since this monitoring is related to security, sufficient and proper condition is covered.



(a) boundary violation detection (b) branch instruction (ex: br reg)

Figure 7. Branch Instruction Monitoring

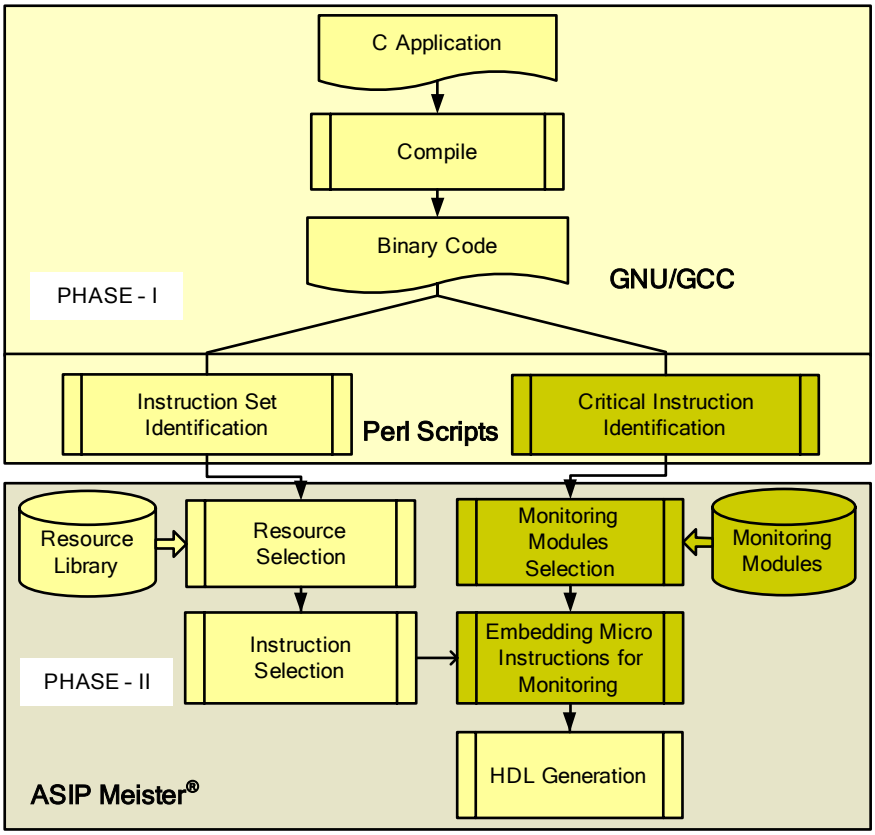


Figure 8. Design Methodology

The microinstructions shown in bold italics in Figure 7(b) are related to the monitoring scheme. As shown in Figure 7(b) the upper and lower addresses of the instruction memory vector are read from special registers called Upper_Limit and Lower_Limit. They are compared with the current target address with the help of two comparators (COMP_LE and COMP_GE). Target address is set to the current PC location where there is no boundary violation.

4. METHODOLOGY

Design methodology describes how the design of our proposed system is performed, and the evaluation methodology outlines the simulation and synthesis for finding overheads and verifying the correctness of the system. Once again, please note that we are not claiming that our system is complete. The solution described here serve as an example system for evaluation purposes.

4.1 Design Methodology

The design flow for a particular application is given in Figure 8. The approach consists of two phases: 1) Compilation and analysis; 2) ASIP generation with self-monitoring instructions.

In Phase-I, application program is compiled to assembly code and then to binary using GNU/GCC [5] cross-compiler for a base RISC architecture (Arm Thumb™ Instruction Set [1][6]). Then the assembly code is analyzed to identify target instructions and critical instructions. Target instructions are the subset of the ARM Thumb™ instructions necessary to generate the ASIP for

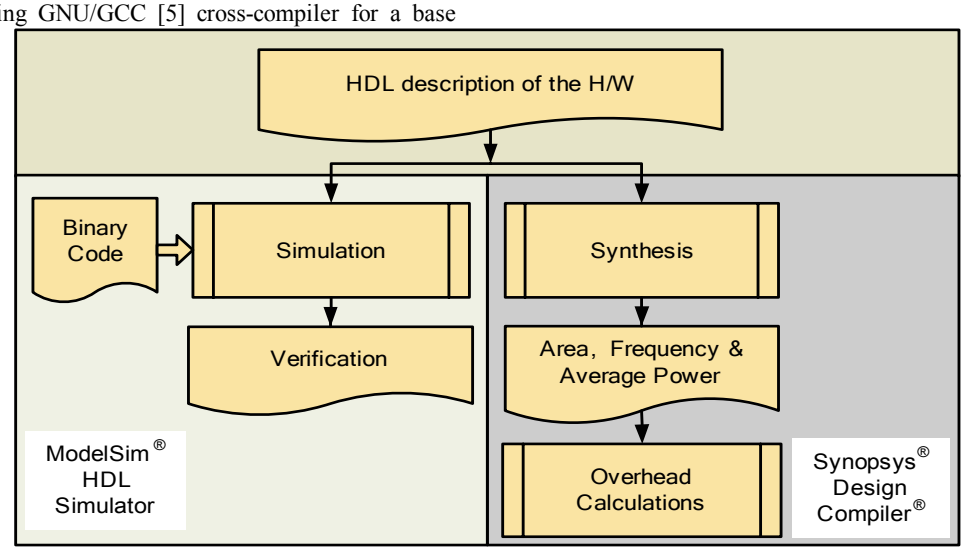


Figure 9. Evaluation Methodology

that particular application. For example, if that application does not require multiply instruction, that instruction is not included in the target instruction set. Critical instructions are the type of instructions in which we are going to embed security monitoring. Critical instruction identification needs lot more research and is beyond the scope of this paper. Finding target instructions is automated using Perl [8] scripts and critical instructions are handled manually at this stage. Thus, the output of Phase-I is the instruction set for the target processor and the monitors associated with each instruction.

Phase-II uses ASIP Meister [2][3][4], an automatic tool for designing ASIPs. This tool captures target processors' specifications using a GUI, "Architecture Design Entry System" and generates micro-operation level simulation model and RT level processor description for logic synthesis in VHDL. Based on the target instruction set of the processor from Phase-I, resources are selected from a resource library and monitoring modules are selected from the check module library. Then the target machine instructions are selected from a base RISC architecture, and microinstructions of the critical machine

Table 1. Frequency, Area and Power Comparison of security monitors for ASIPs based on Arm Thumb

Bench Marks		Adpcm decode	Adpcm encode	Bit count	Basic math	Crc32	qsort	Rijndael decode	Rijndael encode	Sha encrypt	String search
Frequency (MHz)	N/C	231	233	221	227	227	225	219	219	222	226
	BIM	225	226	214	220	220	218	214	214	215	218
	DP	223	227	220	215	216	217	218	218	219	216
	IC	230	232	219	226	226	224	215	215	221	225
	RAS	229	228	220	226	221	219	216	216	220	225
Area (1000 cells)	N/C	84.4	80.3	82.2	82.6	83.5	96.3	99.7	99.7	88.1	81.0
	BIM	88.4	83.9	86.0	85.9	87.3	98.1	105	105	92.7	84.0
	DP	92.0	83.8	85.2	83.8	86.7	97.6	104	104	92.7	83.8

	IC	85.4	80.8	82.3	82.7	83.7	96.5	99.9	99.9	88.9	81.2
	RAS	94.6	92.2	93.5	93.2	95.8	107	112	112	100.0	85.6
Average Power (mW)	N/C	6.02	5.61	5.92	5.62	5.84	6.02	6.93	6.93	6.31	5.71
	BIM	6.05	5.65	5.94	5.66	5.88	6.08	6.98	6.98	6.34	5.72
	DP	6.39	5.98	6.27	6.01	6.2	6.37	7.21	7.21	6.7	6.05
	IC	6.06	5.64	5.94	5.64	5.87	6.06	6.96	6.96	6.34	5.73
	RAS	6.37	5.96	6.26	5.95	6.18	6.35	7.26	7.26	6.66	5.95

microinstructions of the critical machine instructions are embedded / augmented with monitoring routines, generating self-monitoring instructions. Finally, VHDL code representing the designed ASIP is generated.

4.2 Evaluation Methodology

The evaluation platform for a particular application is given in Figure 9. ModelSim is used to simulate the VHDL code from Phase-II. The binary of the application from Phase-I is used for simulation. Simulation is used to verify the correctness of the methodology.

Synopsys Design Compiler® is used for the synthesis of ASIPs. ASIP with embedded monitoring routines is compared with an ASIP without monitoring routines to calculate overheads. TSMC's 90nm Core Library with Typical Conditions (tcbn90gtc) is used for synthesis.

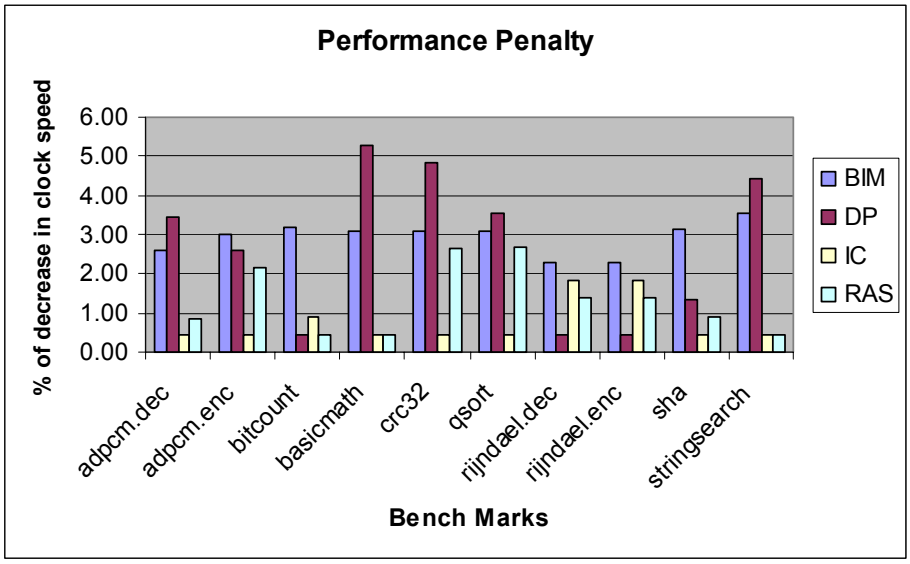


Figure 10. Performance Overhead

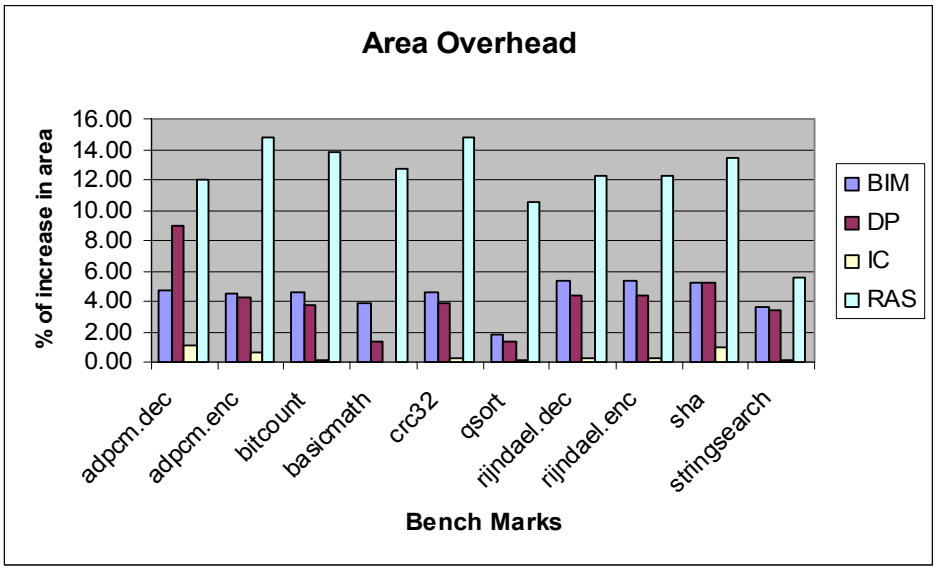


Figure 11. Area Overhead

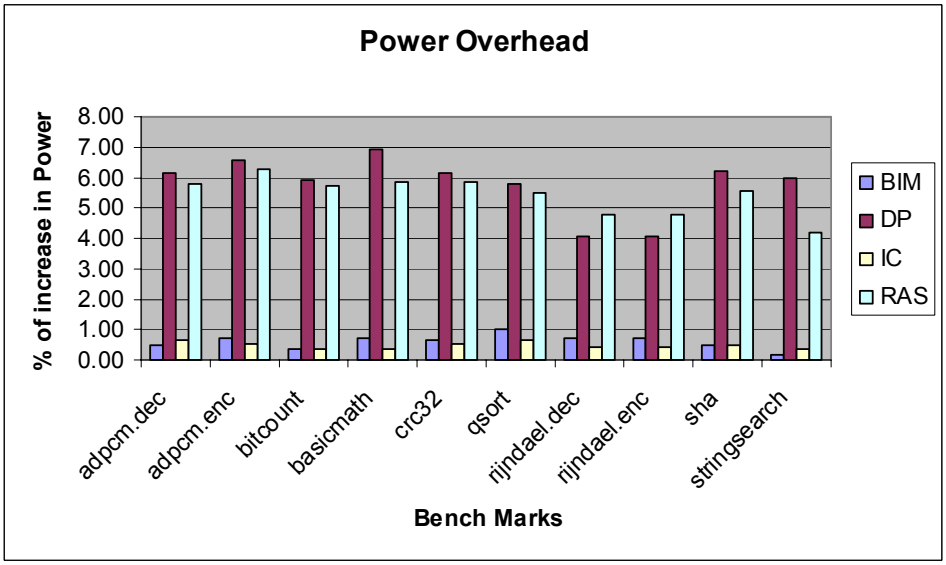


Figure 12. Power Overhead

5. COST AND OVERHEAD

Table 1 shows the synthesis results of ASIPs generated for ten different benchmarks of MiBench benchmark suite [17]. ASIPs are synthesized for each Benchmark with no security monitoring (N/C) and for different security monitoring methods (Instruction Memory Data Bus Monitoring (IC), Data Path Monitoring (DP), Return Address Stack Monitoring (RAS) and Branch Instruction Monitoring (BIM)). The processor model is used to measure the frequency, area and power. The measured values are compared against that of the processor model with no monitoring- systems.

From Table 1 it is evident that the clock frequency is lower for any processor with a monitoring system, than that of the processor with no monitoring system for the same benchmark. The reason for the reduction in the frequency is the additional logic added to monitor the system. The reduction in the clock frequency is not the same for a particular monitoring technique in all benchmarks. The reason for this is the security monitoring techniques are implemented in different sets of instruction types. For example, Branch Instruction Monitoring (BIM) will be imposed only on branch instructions. Therefore the impact of the BIM system on an ASIP will depend on the number of types of branch instructions in an application. This number depends on the nature and amount of instructions in an application, and the frequency reduction will be varying for different benchmarks with differing monitoring systems.

Figure 10 shows the percentage of performance penalty due to the inserted security monitors. The average performance penalty in ten benchmarks is 1.93%.

It is obvious that increasing logic will cost more area. The increment in area also depends upon the total number of instructions to which the monitoring is required and the amount of logic used for each monitoring system. The area cost for RAS system is more compared to those of the others. RAS uses a register-file as a hardware stack (with 32 entries). The use of register-file will increase the area considerably. Maximum area penalty is around 15% and it is reasonable and is more reliable than using a separate hardware to monitor the processor.

Figure 11 shows the area overhead due to security monitors. The average area overhead for ten different bench marks with four security monitors is 5.26%.

Power consumption has increased only by a maximum of 7% compared to the processor without monitoring. Power is consumed more in the case of DP which requires more signal flow during each decode stage and register write back stages for all the instructions with at least one destination register. This causes the increment in power to be high in DP compared to other monitoring systems discussed in this paper except for two applications, rijndael.dec and rijndael.enc. These two applications have more function calls compared to the others and their RAS power consumptions are more compared to DP. Note that RAS monitoring is related to function calls.

Figure 12 gives the additional power consumption of ASIPs due to embedded security monitors. The average of ten ASIPs with four different monitors is 3.07%.

Vetteth has implemented two RSE modules in hardware description language and evaluated the performance and area overhead [50]. The RSE framework with two modules occupy 6% extra area with 5.5% performance penalty in a double issue DLX processor [51]. Our overheads are similar to the overheads shown in [50] If our checks were to be implemented in RSE, then compiler support would have been essential. As it stands now, we do not require compiler support.

The results above show that we can implement security monitoring at the microinstruction level with low area and power penalty with a minor reduction in performance. The security monitors are evaluated for overheads on a one-by-one basis. When all the checks are implemented together, the performance overhead would be that of the maximum performance penalty and the area and power overheads would accumulate.

6. CONCLUSION AND FUTURE WORK

In this paper we have proposed a methodology for detecting security attacks in embedded processors, and implemented this system using ASIPs. For the first time we have proposed the idea of embedding microinstruction routines within vulnerable machine instructions that could invite security attacks to detect whether they are compromised. We have also presented the overheads for such security detection routines with ten different benchmark applications from MiBench.

The techniques we have presented here for a secure system to reduce the possibility of successful attacks. It may not be good enough to cover few of the main threats and leave the others claiming that they are not frequent [35]. Future work will include finding the security requirement according to the specific applications. It is also essential to develop a measurement tool for the real secure behavior in the design. Meanwhile, the recovery mechanism needs to be worked out for each and every case.

7. REFERENCES

[1] Arm Reference Manual, Advanced RISC Machines Ltd. 2000.

[2] ASIP Meister Tutorial, PEAS PROJECT. 2003.

[3] *ASIP Meister User Manual, PEAS PROJECT.* 2003.

[4] ASIP Meister, Available at http://www.eda-meister.org/asip-meister.

[5] The GCC Team, GNU/GCC Compiler, Free Software Foundation.

[6] An Introduction to Thumb Advanced RISC Machines Ltd. 1995.

[7] Merriam-Webster's Online Dictionary, 10th Edition, Available at http://www.m-w.com.

[8] Perl Programming Language, Available at http://www.perl.org.

[9] The SANS Institute, The SANS/FBI Twenty Most Critical Internet Security Vulnerabilities. 2004.

[10] Alomary, A., T. Nakata, and Y. Honma, *PEAS- I: A Hardware/Software Co-design System for ASIPs.* IEEE International Test Conference, 1993: p. 2-7.

[11] Baratloo, A., N. Singh, and T. Tsai, Transparent Run-Time Defense Against Stack Smashing Attacks. 2000.

[12] Boneh, D., R.A. DeMillo, and R.J. Lipton, *On the Importance of Checking Cryptographic Protocols for Faults.* Lecture Notes in Computer Science, 1997 p. 37-51.

[13] Deckard, J., *Defeating Overflow Attacks* The SANS Institute 2004.

[14] Dyer, J.G., et al., *Building the IBM 4758 Secure Coprocessor.* Computer, 2001. **34**: p. 57-66.

[15] Gebotys, C.H., Low energy security optimization in embedded cryptographic systems. 2004, ACM Press. p. 224-229.

[16] Glokler, T. and H. Meyr, *Design of Energy-Efficient Application-Specific Instruction Set Processors (ASIPs)*. First Edition ed. 2002: Kluwer Academic Publishers.

[17] Guthaus, M.R., et al., *Mibench: A free, commercially representative embedded benchmark suite.* In IEEE 4th Annual Workshop on Workload Characterization, Austin, TX, 2001: p. 83-94.

[18] Hess, E., et al., Information Leakage Attacks Agaist Smart Card Implementations of Cryptographic Algorithms and Countermeasures. 2000. p. 55-64.

[19] Joglekar, S.P. and S.R. Tate, ProtoMon: Embedded Monitors for Cryptographic Protocol Intrusion Detection and Prevention. 2004, IEEE Computer Society.

[20] Kc, G.S., A.D. Keromytis, and V. Prevelakis, *Countering code-injection attacks with instruction-set randomization*. 2003, ACM Press. p. 272-280.

[21] Kelsey, J., et al., Side Channel Cryptanalysis of Product Ciphers. 1998. p. 97-110.

[22] Kmmerling, O. and M.G. Kuhn, Design Principles for Tamper-Resistant Smartcard Processors. 1999. p. 9-20.

[23] Kocher, P., et al., Security as a New Dimension in Embedded System Design. 2004.

[24] Lee, R., et al., Enlisting Hardware Architecture to Thwart Malicious Code Injection. 2003, Springer Verlag LNCS.

[25] Mahmood, A. and E.J. McCluskey, *Concurrent Error Detection Using Watchdog Processors-A Survey*. IEEE Trans. Computers, 1988. **37**: p. 160-174.

[26] Marwedel, P. and C. Gebotys, Secure and safety-critical vs. insecure, non safety-critical embedded systems: do they require completely different design approaches? 2004, ACM Press. p. 72-73.

[27] McGregor, J., et al., A Processor Architecture Defense against Buffer Overflow Attacks. 2003, Springer Verlag. p. 237-252.

[28] Muresan, R. and C.H. Gebotys, Current flattening in software and hardware for security applications. 2004, ACM Press. p. 218-223.

[29] Nakka, N., et al., An Architectural Framework for Providing Reliability and Security Support. 2004, IEEE Computer Society.

[30] Quisquater, J.J. and D. Samyde, *Side Channel Cryptanalysis*. 2002. p. 179-184.

[31] Ragel, R.G. and S. Parameswaran, Soft Error Detection and Recovery in Application Specific Instruction-set Processors. 2005.

[32] Ravi, S., A. Raghunathan, and S. Chakradhar, Tamper Resistance Mechanisms for Secure, Embedded Systems. 2004.

[33] Ravi, S., et al., *Security in embedded systems: Design challenges*. Trans. on Embedded Computing Sys., 2004. **3**: p. 461-491.

[34] Reinhardt, S.K. and S.S. Mukherjee, *Transient fault detection via simultaneous multithreading*. 2000, ACM Press. p. 25-36.

[35] Richarte, G., Four different tricks to bypass StackShield and StackGuard protection. 2002.

[36] Schneider, F.B., G. Morrisett, and R. Harper, *A Language-Based Approach to Security*. Lecture Notes in Computer Science, 2001. **2000**: p. 86-101.

[37] Shao, Z., et al., Security Protection and Checking in Embedded System Integration Against Buffer Overflow Attacks. 2004, IEEE Computer Society.

[38] Shao, Z., et al., Defending Embedded Systems Against Buffer Overflow via Hardware/Software. 2003, IEEE Computer Society. p. 352.

[39] Sint, M., MIDL - a microinstruction description language. 1981, IEEE Press. p. 95-106.

[40] Smith, S.W. and S. Weingart, *Building a high-performance, programmable secure coprocessor*. Comput. Networks, 1999. **31**: p. 831-860.

[41] Suh, G., et al., AEGIS: Architecture for tamper-evident and tamper-resistant processing. 2003.

[42] Suh, G., et al., Hardware mechanisms for memory integrity checking. 2002.

[43] Wagner, D., et al., A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. 2000: San Diego, CA. p. 3-17.

[44] Wiley, R. and E. Wiley, *Smart Card Handbook*. 2000.

[45] Wilken, K.D. and T. Kong, *Concurrent Detection of Software and Hardware Data-Access Faults*. IEEE Trans. Comput., 1997. **46**: p. 412-424.

[46] Xu, J., Intrusion Prevention Using Control Data Randomization, in Suppl. of IEEE International Conf. on Dependable Systems and Networks (DSN), San Francesco, CA 2003.

[47] Xu, J., Z. Kalbarczyk, and R.K. Iyer, *Transparent Runtime Randomization for Security*. 2003, IEEE Computer Society.

[48] Xu, J., et al., Architecture support for defending against buffer overflow attacks. 2002.

[49] Xu, J. et al., An Architectural Framework for Providing Security and Dependability Support, 2004.

[50] Vetteth, A., Hardware Implementation of Reconfigurable Modules for Reliability and Security Engine, Master's Thesis, University of Illinois at Urbana Champaign, May 2005.

[51] H.Eveking, Superscalar DLX Documentation, http://www.rs.e-technik.tu-darmstadt.de/TUD/res/dlxdocu/DlxPdf.zip.

[52] Fisher, J.A., Customized Instruction-sets for Embedded Processors, in DAC'99: Proceedings of the 36th ACM/IEEE conference on Design Automation. 1999, ACM Press: Orleans, Louisiana, United States. p. 253-257.

[53] Fisher, J. A., Faraboschi, P., and Desoli, G. Custom-Fit Processors: Letting Applications Define Architectures. International Symposium on Microarchitecture, Micro-29, Paris, France, 1996, 324-335.