Journal of Universal Computer Science, vol. 24, no. 4 (2018), 515-536
submitted: 14/12/17, accepted: 15/4/18, appeared: 28/4/18 © J.UCS

A Fine-Grained Hardware Security Approach for Runtime Code Integrity in Embedded Systems

Xiang Wang, Weike Wang[1], Bin Xu, Pei Du, Lin Li
(School of Electronic and Information Engineering
Beihang University, Beijing 100191, China
wxiang@buaa.edu.cn, wangweike@buaa.edu.cn, xubin1978@buaa.edu.cn
ziyoumudi@buaa.edu.cn, linl.ahch@buaa.edu.cn)

Muyang Liu
(School of Instrument Science and Opto-electronics Engineering
Beihang University, Beijing 100191, China
muyangliu@buaa.edu.cn)

Abstract: Embedded systems are subjected to various adversaries including software attacks, physical attacks, and side channel attacks. Most of these malicious attacks can lead to the invalid execution of programs, and launch of destructive actions or reveal critical information. However, most previous security mechanisms suffer from coarse checking granularity and unacceptable performance overhead, due to strict restriction on system resources. This paper presents a fine-grained hardware-based security approach to ensure runtime code integrity in the embedded systems by offline profiling of the program features and runtime integrity check. We design a hardware implemented instruction stream integrity checker (ISIC) to perform runtime checking of pre-extracted features. Any invalid execution of the program will trigger the corresponding exception signal. We implement the ISIC with OR1200 processor on XC5VLX50T field-programmable gate array (FPGA). The experimental results show that the proposed approach can detect all the attacks destructing integrity of the instruction stream, and the performance overhead induced by the security mechanism is less than 3.45% according to the selected benchmarks.

Keywords: embedded system, basic block, runtime security, code integrity, hardware-based security
Categories: B.6.1, C.5.4, D.4.6

1 Introduction

With the rapid development of wireless communication and the Internet of Things technology, embedded systems are widely employed in all spheres of our daily lives. As embedded systems expose themselves to the Internet and public, they have to deal with some new issues of security. The applications such as cell phones, wearable devices, financial terminals, industrial control systems, and military devices process and store a giant number of users' critical information, and provide users with convenience as well. Apart from some security issues derived from the system architecture and hardware implementation, these embedded devices are increasingly

[1] Corresponding author

subjected to the adversaries that aim to get the private data of the users or control the behavior of the programs to perform malicious actions.

The embedded system can be compromised either through the physical tampering that commits electronic jamming or electronic eavesdropping, or through the perpetration of program exploits that change valid program operation [Serpanos and Voyiatzis, 2013]. Compared with the attacks on physical devices, software security exploits taken advantages of vulnerabilities in operation systems or applications is more widely used due to the less requirements of the victim hardware details. Malicious adversaries exploit to tamper program code and data, inject malicious code, and leak critical information. These attacks are easy to implement because most embedded programs are written in the unsafe program languages such as C and C++, which are not strongly typed and allow direct access to memory without bounds checking [Wang et al., 2008]. Adversaries can easily inject malicious codes and data using these software vulnerabilities, especially as more and more embedded devices are connected to the Internet. Most of these attack patterns eventually lead to the invalid execution of the program. Such attacks, say, stack smashing can be resolved using the techniques such as Write XOR eXecution ($W \oplus X$) [Fiskiran and Lee, 2004], Data Execution Prevention (DEP) [Ahn et al., 2014], Address Space Layout Randomization (ASLR) [Kanuparthi et al., 2012(a)], in-stack canaries [Shehab and Batarfi, 2017], and some software code integrity checkers [Bletsch et al., 2011; Abadi et al., 2005; Davi et al., 2014]. However, most of these techniques need to change the embedded system hardware structures, add redundant system modules or modify the instruction set architectures and the compilers. These technologies cannot be ported to any embedded system platform simply.

In consideration of the potential threats to embedded system security, the embedded systems must have the assurance of code integrity. In other words, the original code and the data processed should be fetched from the original program, and have not been injected, cut, modified or substituted by any adversary. An intractable challenge for embedded system security is the constraint resources, namely limited performance, power and area. Embedded systems with limited resource budgets cannot afford abundant hardware for reliability. Well known desktop security solutions have large system requirement and significant performance overhead, thus cannot be ported to embedded systems for the lack of sufficient system resources. Besides, most of the widely used system security solutions, based on anti-virus software, are program codes themselves and difficult to avoid software vulnerabilities. Thus, some hardware based lightweight security methods are required to ensure runtime security in embedded systems. Several static techniques employ source code scan and review tools to strengthen code security by reducing the program vulnerabilities at the software design phase, without taking the runtime attack threats into account. Some runtime software monitoring and binary instrumentation techniques have been proposed since it was published systematically by N. Oh [Oh et al., 2002]. But these solutions inevitably increase code sizes, introduce performance overhead and are themselves vulnerable to corruption. Recently, various hardware based schemes, such as the novel runtime monitoring architecture proposed by D. Arora [Arora et al., 2006], ROPdefender [Shacham, 2007], CICM [Rogers and Milenkovic, 2009], DIC [Kanuparthi et al., 2012(b)] and BB-CFI [Das et al., 2016], have been proposed for embedded systems, but most of

them either suffer from significant performance overhead or have their own limitations.

In this paper, we present a hardware based technique at basic block granularity to ensure that the embedded program is not deviated from its intended and permissible behavior. It is a novel hardware based security mechanism, which enables the checking of specific properties of the executed program at a fine granularity of the basic block and provides high efficiency in violation detection. To dynamically prevent program code integrity from runtime malicious attacks, we propose a kind of hardware architecture called the instruction stream integrity checker (ISIC). Our mechanism comprises three steps to run. All these steps are performed after the program is compiled and linked. In the first step, the binary code is divided into basic blocks and profiling offline according to the specified basic block division rules. In the second step, the hash value of each basic block is computed, which is assumed as the golden hashes of the program. These golden hashes are stored in the specified hash memory of the processor at the load stage. These hash values are assumed to be trusted and cannot be accessed by any adversary. Here we adopt LHash [Wu et al., 2014], a lightweight hash algorithm suitable for embedded systems, to reduce area footprint and power overhead. Then, the basic block address and the golden hashes are compressed together to form the security monitoring model for each basic block. In the last step, runtime security checking is conducted to enforce the instruction stream integrity with the pre-extracted monitoring model at basic block granularity. We introduce the architecture as a hardware-based monitor that can be appended to any embedded processor to check its dynamic execution trace, as it runs programs and checks whether the trace conforms to the requirements of permissible behaviors, and triggers appropriate response mechanism if any security violation is detected. The hardware architecture is implemented as a programmable module that can be configured to run the program with the ISIC being activated or not.

We implement the proposed architecture on Xilinx XC5VLX50T field-programmable gate array (FPGA) as a prototype system, which can provide low area and power overhead, and high verification performance. To evaluate the performance of the architecture, we use various scales of benchmarks from MiBench benchmark suite [Guthaus et al., 2001] to generate realistic workloads. The implementation shows that the ISIC can detect all instruction stream integrity attacks due to fine-grained checking granularity. The processor performance overhead induced by the security mechanism of the ISIC is less than 3.45% according to the selected benchmarks. Besides, the prototype system also has the advantages of low power consumption and marginal area footprint. The proposed hardware based monitoring architecture only traces the executing instruction and its corresponding address signals of the pipeline, with less intrusion to existing embedded processor architectures, and can be easily ported to any processor platforms.

The remainder of this paper is organized as follows. In Section 2, we discuss the threat model we focus on and related works. In Section 3, we present the proposed security mechanism in detail. Section 4 gives some security analysis of the proposed mechanism. Section 5 presents our experimental results. Finally, we give a conclusion in Section 6.

2 Background

2.1 Threat Model

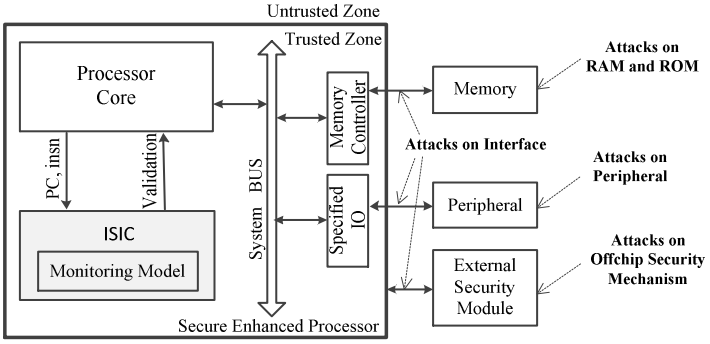

Figure 1: Threat model of the proposed work

Embedded systems can be compromised in many forms, including software attacks, physical attacks, and side-channel attacks. Most of these malicious attacks eventually lead to the invalid execution of the program or the leakage of the critical information. Compared with physical attacks and side-channel attacks, software attacks are much easier to conduct because most of them are irrelevant to hardware details. Adversaries can gain access to the shut-down embedded systems, and modify the original code in the non-volatile memory. As the systems booting up, the malicious actions will be imported into the vulnerable systems. Some attackers take advantage of the interval time between the time when a program is checked for code integrity at boot time, and the time when the program is beginning to execution, to launch runtime attacks. They can access any off-chip memory to inject, modify and delete the program code and data. In this case, any integrity checking mechanism at boot time will out of effectiveness. What's more, runtime software attacks also exploit vulnerabilities such as buffer overflow, format strings, and dangling pointers in operation system, middleware, and applications to launch malicious attacks. Most of these malicious attacks eventually lead to the invalid execution of the program or the leakage of critical information. Most physical and side-channel attacks involve direct or indirect tampering to the interfaces and peripherals to perform spoofing, splicing and replay attacks, as well as change the valid code and data in their featured method.

Figure 1 illustrates the threat model for the embedded system we take into account. We define the regions on the embedded SoC chip as the trusted zone, while all the interfaces and wires connected to the SoC and all system components and peripherals off the chip are assumed to be untrusted. That is to say, all adversaries cannot tamper the pipeline, registers, Cache and any signal inside the embedded processor. The on-chip system bus, memory, peripheral controllers and any security enhanced architectures are also assumed to be immune to all kinds of attacks. Software attacks, physical attacks, and side-channel attacks can be launched at all design stages ranging from the program installation phase to the loading phase and execution phase. The targets of these attacks include the RAMs and ROMs off the

Chip, system components, peripherals, interfaces and some external wires. Some off-chip external security modules are assumed to be untrusted, because they cannot ensure the validation of the security protection function and the interface signals.

2.2 Related Works

Various defense techniques have been proposed to address the security threat in computing systems. Almost all of them put emphasis on static or dynamic analysis of the source code and runtime identity verification of the protected programs. These security methods are implemented in different handling ways, but most of them have their own limitations. Especially in embedded systems, limited system resources make some technical proposals inapplicable any more.

Some static techniques such as source code or binary code scan and review tools verify the program validation at the design phase. Buffer overflow, format strings, dangling pointers and some other software vulnerabilities can be detected by these techniques [Zitser et al., 2004; Dor et al., 2003]. Some desktop computer antivirus software likely techniques adopt learning algorithms to extract malicious attack characteristics in advance, and scan the source code to find pieces of illegal code according to the learned malicious features. The advantage of these static security methods is that they can perform security checks before software is executed to prevent irreparable damage to the hardware and system. But those methods can only detect the illegal code before execution, any load time and runtime attacks may cause the system to crash.

In recent years, some hardware based techniques focusing on runtime security have become prevalent. Threads or process level redundancy based methods [shaye et al., 2007] for reliability need extra parallel hardware resources, resulting in a large waste of system resources. Some additional secure coprocessors and separate Trusted Platform Modules (TPM) are used to enhance system security. R. A. Calix proposed an embedded machine learning processor to detect intrusions by using network-based features to distinguish normal and abnormal actions [Sankaran and Calix, 2016]. But the interfaces and wires between the external security peripherals and the host processor are vulnerable to attacks. The XOM architecture [Lie et al., 2000] uses cryptographic techniques to encrypt the code and data in memory, and decrypt them at the execution phase. The session key is used to isolate the programs running on the same machine from others. Due to the nescience of the session key, any malicious manipulation of the code and data may lead to the collapse of the system. The process granularity of XOM is instruction, which results in serious performance losses. Roger proposed one runtime verification architecture encompass secure installation, secure loading and secure execution with programmable protection mode [Rogers and Milenkovic, 2009]. They append 128-bit signatures to 32-byte I-blocks. The discontinuous execution of the binary code may lead to the failure of the premature signatures. REM [Fiskiran and Lee, 2004] and SPEF [Kirovski et al., 2002] are secure program runtime checkers at the granularity of basic blocks. They use different hash functions to build the keyed Message Authentication Code (MAC) at compile time. But REM requires modification of Instruction Set Architecture (ISA), and it increases the binary size dramatically. D. Arora presented a hardware assisted mechanism to check both code integrity and control flow validation [Arora et al., 2006]. In addition to malicious modification of the program code, changes in the registers in the indirect

jump and branch instructions can be also detected. The multi-level checker proposed by Arora is coarse-grained, and thus emergent invalid procedures cannot be detected at once. In addition, MD4 and MD5 hash functions are used in Arora's work, but they are not applicable to embedded systems. X. Wang [Wang et al., 2013; Wang et al., 2016] presented a code security mechanism with the on-chip secure module. They divided the binary code into basic blocks offline, and verified them at runtime. Recently, some novel intrusion detection techniques have been proposed. Parallax [Andriesse et al., 2015] used the return-oriented programming (ROP) technique as an attack method for the embedded system to verify code integrity by overlapping ROP gadgets with instructions. The method does not rely on code checksum, so it is not vulnerable to cache modification attacks. The results show that Parallax can only protect up to 90% code bytes.

3 Architectures for Runtime Verification

3.1 Model of Permissible Procedures

To detect the attacks involving code integrity tampering, it is not appropriate to use instruction by instruction verification approach. The proposed fine-grained security approach uses the techniques including offline static extraction of program properties and hardware based runtime verification to ensure the security of embedded systems. There are some retrieve rules for the security related program properties. First, the selected properties must be sufficient to indicate all security issues concerned, that is, any security exception of the program will cause the selected properties to be tampered. Besides, they should be easy to be derived from the source code or binary code. In consideration of the limited system resources and performance, the selected monitor model must be the minimum effective set of these properties. After the selection and extraction of the monitor model, the ISIC is adopted to perform runtime validation of these pre-processed features.

In our work, we employ hash value matching by basic blocks cryptographic hash results to perform integrity validation. These pre-computed hash values will be copied to the monitor model memory of the ISIC at load time, and verified during program execution. For the message x_i and its cryptographic hash value $Hash(x_i)$, it is computationally not easy to find another x_j, such that $x_i \neq x_j$, and $Hash(x_i) = Hash(x_j)$. Especially in the instruction integrity verification case, the input messages are the instructions in basic blocks. It is infeasible for adversaries to find a malicious instruction sequence that can be legally executed by the processor and has the same hash value as the original code. In consideration of the performance and area margins in the embedded system, we employ LHash in our scheme. This lightweight hash algorithm can map the digest sizes of 80, 96 and 128 bits, providing pre-image security from 64 bits to 120 bits, second pre-image security and collision security from 40 bits to 60 bits.

In the design phase, the binary code is divided into basic blocks, and the program properties are extracted from each basic block as the indicators of invalid program behaviors. Systems can be compromised in software installation, loading, and execution phases. All malicious manipulation from the previous stages, including injection, modification, deletion and any attacks tampering the program code, will be

transmitted to the execution phase and executed in the processor pipeline. Therefore, we place the security monitoring node in the execution phase. The runtime code integrity checker can detect all previous attacks accumulated in the execution stage.

Procedure 1: instruction stream check flow

```
1:  Inputs: PC, insn, monitor model M(BB_S, LHASH_G)
2:  Output: invalid_status
3:  BB ← set of basic blocks bb_i
4:  BB_S ← set of basic block start addresses bb_s_i, 1≤ i ≤ total of basic blocks
5:  INSN_i ← set of instructions of the ith basic block insn_ij, 1 ≤ j ≤ total
                                        instructions in the ith basic block
6:  LHASH_G ← set of golden hashes lhash_g_i, 1≤ i ≤ total of basic blocks
7:  for all bb_i ∈ BB do
8:     pc= index (bb_i)
9:     if pc = bb_s_i : bb_s_i ∈ BB_S then
10:       for all insn_ij ∈ INSN_i do
11:          LHash_i = f_LHash(insn_i1, insn_i2, ... , insn_ij)
12:       if LHash_i = lhash_g_i then
13:          invalid_status = NULL  /* no error */
14:       else invalid_status = 01    /* LHash value error */
15:    else invalid_status = 10       /* start address error */
```

The inputs of the ISIC connected to the embedded processor are the PC and the INSTRUCTION signals from the instruction decode (ID) stage in the pipeline. At runtime, when the ISIC detects the start of a basic block, the LHash engine of the ISIC is enabled. With the execution of the basic block, current and subsequent instructions are continuously pumped into the LHash engine until an instruction of the end type, which is a branch or a jump instruction, is detected. At the same time, the LHash engine absorbs the instruction sequence and calculates the LHash value within 2 clock cycles. Then the controller reads the corresponding monitor model from the pre-stored memory, and separates out the credible golden LHash value. The dynamically computed LHash values are compared with the pre-stored golden features. If the pre-stored and the computed values are different, an alarm is raised and a malicious code is detected. Procedure 1 listed above describes the instruction stream check flow at the granularity of the basic block.

During the execution of the program, once the ISIC detects a violation of code integrity, it will assert invalid signals and feed the error type back to the processor. A 2-bit special register is used to tag the error types when the hardware ISIC is implemented. This is used to divide the violations into three broad categories.

- ✓ Violation due to the start address loss from the monitor model. The branch and jump instructions are the boundaries between basic blocks, and the ISIC automatically divides the basic blocks according to these instructions. If a branch or jump instruction is generated or replaced by adversaries, then this kind

of violation will occur. In this way, the golden hash of current basic block cannot be indexed successfully.

- ✓ Violation due to the mismatch of basic block cryptographic hash results between the pre-computed and runtime generated. The matching of hash results is validated once at the end of a basic block. This can occur if the instructions in this basic block suffer from attacks of injection, cut, modification and substitution.
- ✓ Violation due to the execution of illegal instructions. Some adversaries can modify a block of instructions within a basic block to fulfil the match check of the cryptographic hash and the relative address. But this may induce the illegal instructions cannot be executed by the processor or invalid access addresses outside the space bounds.

When a violation is detected, the processor will get the error type and make the corresponding response according to some further security requirements. One common form is to terminate the program, flush the processor pipeline and then switch the processor to a secure mode.

3.2 Basic Blocks and Profiling

The purpose of profiling is to extract the basic block features of the target program. According to the extracted information, some hardware modules can be used to validate the execution of the program. Profiling operation plays an important role in our approach. In our work, the input of the profiling process is the binary code instead of any high level programming languages, such as C/C++ and Java. Thus there is no special requirement for the programing language and the compiler used by the programmer. Especially in some bottom designs of embedded systems, cross application of the assembly language and high level languages is required. Some source code based profiling methods mentioned in [Arora et al., 2006], [Das et al., 2016], and [Li et al., 2016] needs more special consideration in this case. Similar to most profiling methods [Kanuparthi et al., 2012; Arora et al., 2006; Das et al., 2016; Mao and Wolf, 2010], we assume that the profiling process and the inputs of the profiling stage are trusted. This means that any software bug and modification in the binary code before input to the profiling stage cannot be detected, which is acceptable because the programmer can guarantee that the binary code is consistent with the expected design scheme when the software development is complete. The status after the program delivery is the key to security.

The basic block is defined as a piece of binary code containing only the instructions which will be executed sequentially. In order to avoid the addition of extra tags to indicate the end of a basic block, we use jump and branch instructions as marks to divide the binary code into basic blocks. Therefore, the ending instruction of each basic block is a jump instruction or a branch instruction. In order to normalize the range of basic blocks, the start address of each basic block is defined as the next instruction of a previous jump/branch instruction, or the target address of a previous jump/branch operation. Such partition method may cause overlap of basic blocks. Compared with some non-overlapping strategies, this method reduces the number of dynamic matches at the same storage cost.

The monitor model defines the reliable monitor features of each basic block. As mentioned above, the monitor model of the proposed approach contains the start address and the pre-computed LHash value of each basic block. Taking a 32 bits RISC-based embedded processor as an example, each instruction and the corresponding address is 32 bits. The data and instructions are all aligned to 4 bytes. Thus the lower 2 bits of the address will be fixed to 2'b00. The actual useful value for the start address is PC[31:2]. Taking the storage overhead into account, only the lower 16 bits in the PC[31:2] are used in the proposed monitor model. This would give the programmer at most 256 KB addressable space. Larger address space can be achieved if more bits are employed in the monitor model.

For the LHash segment, we set the 32 bits instruction as the message block XORed to part of previous permutation state and enter into the next permutation engine in the extended sponge function. The optional block size of the LHash internal permutation is 96 bits and 128 bits in this situation. The security properties of the LHash algorithm based on the sponge construction can be concluded as Equation 1 [Bogdanov et al., 2013]:

$$\begin{cases} \text{Collision resistance} : \min\{2^{n/2}, 2^{c/2}\} \\ \text{Second}-\text{preimage resistance} : \min\{2^{n}, 2^{c/2}\} \\ \text{Preimage resistance} : \min\{2^{n}, 2^{c}, \max\{2^{n-r}, 2^{c/2}\}\} \end{cases} \quad (1)$$

Where, n is the digest size, c represents the capacity size of the internal permutation, and r represents the length of the input message blocks. As shown in Table 1, six versions of the LHash algorithm are constructed based on two types of permutations F96 and F128, where b represents the size of the fixed permutation. The absorbing size r is fixed to 32 for the 32 bits RISC processor instructions. The parameters and security bounds can be found in this table. In tag-based applications, 64 or 80 bits security is often appropriate instead of complex constructions providing high security primitives, such as a 512 bits output hash function [Guo et al., 2011]. In our case, we require at least 64 bits preimage resistance security, and 48 bits collision and 2nd preimage resistance security. Therefore, only the parameters in the last two rows can meet the security requirement. To avoid any waste of area or computing power, we set the digest size to 96 bits.

Parameter				**Security Bounds**		
b	**r**	**c**	**n**	**Preimage**	**2nd Preimage**	**Collision**
96	32	64	80	48	32	32
96	32	64	96	64	32	32
96	32	64	128	64	32	32
128	32	96	80	48	40	48
128	**32**	**96**	**96**	**64**	**48**	**48**
128	**32**	**96**	**128**	**96**	**48**	**48**

Table 1: Alternative parameters and security bounds for LHash implementation

To keep the hardware overhead lower, only a few bits of the pre-computed LHash output are selected to constitute the monitor model for each basic block. That means adversaries need to guess fewer bits to bypass the integrity check. If we select the bits in the fixed bit positions, we have $\mathrm{P}(m,n)=\frac{1}{2^n}$, where P(m, n) denote the probability for adversaries to guess the tag value, m represents the digest size of the selected hash function, and n represents the length of the selected LHash bits. As n becomes smaller, the value of P(m, n) will become unacceptable. Thus we use a random number generator to determine the selected positions of the bits in the output hash values. Then we have $\mathrm{P}(m,n)=\frac{1}{\mathrm{C}(m,n)\times 2^n}$, where C(m, n) is the number of ways of choosing n bits out of the m bits output hash value. In our case, we implement the hardware LHash engine with pipelined 32 bits input and 96 bits output, and choose 16 bits from the 96 bits digest. Thus the chance that the adversary guesses the correct hash value for a basic block is $\frac{1}{\mathrm{C}(96,16)\times 2^{16}}$, instead of $\frac{1}{2^{16}}$ in theory.

The monitor model of each basic block is a 32-bit message, which contains the current basic block start address and the golden LHash value. The high 16 bits of the monitor model are used to store the lower bits of the virtual start address of the current basic block, while the low 16 bits are for the randomly selected golden LHash bits. The monitor model structure of a basic block is shown in Table 2, and the definitions of monitor targets in Table 2 are as follows.

Current block start address:
The lower 16 bits virtual start address of current basic block in original binary code

Golden LHash value:
The randomly selected 16 bits trusted LHash value of current basic block extracted from original binary code

Monitor Targets	Width	Range
Current block start address	16 bits	[31:16]
Golden LHash value	16 bits	[15: 0]

Table 2: Monitor model of a basic block

The storage overhead for each basic block monitor model is 4 Bytes. Thus the storage overhead for a program is 4 bytes times the number of basic blocks. Additionally, an extra 96-bit message is added at first in the monitor model to indicate the 16 selected bits in the 96 bits digest values. This constitutes the overall monitor model file for a program. In the process of monitor model extraction, we have developed a Perl script to generate the monitor model from the compiled binary code. Some GNU tools such as objcopy and objdump are invoked by this script. Besides, the random number is generated from the Linux kernel entropy pool by the developed script.

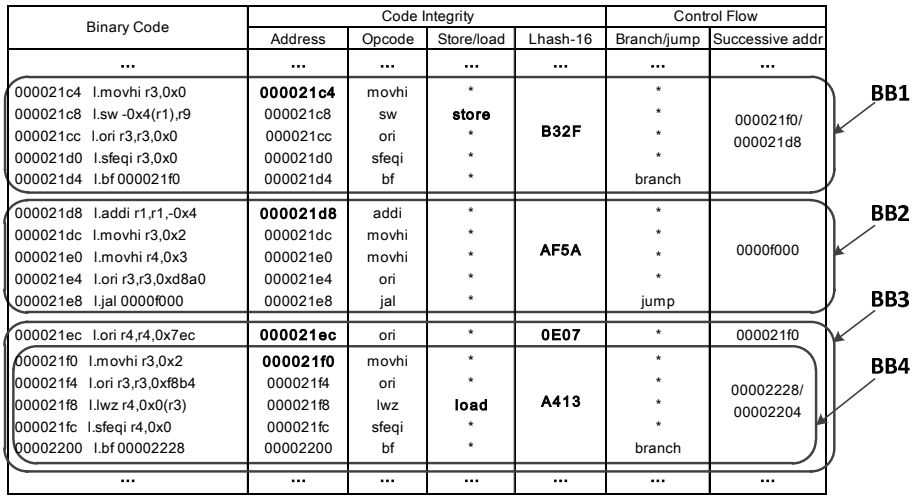

Figure 2: An example of monitor model extraction case for a code segment in the selected OpenECC benchmark

Figure 2 illustrates an example of monitor model extraction for a code segment in the OpenECC benchmark. According to the basic block partitioning strategy, branch and jump instructions divide the binary code into three basic blocks: BB1, BB2 and BB3. At the same time, the start address and hash result of each basic block can be acquired. Then the successive addresses of each basic block are extracted and analyzed. If a successive address is the start address of an existing basic block, no extra processing is required. If not, a new basic block is generated. The successive address is set as the start address of the new basic block. The end address of the new basic block is the address of the nearest branch or jump instruction. In this example, one of the BB1's successive addresses is 0x000021f0, which is not the start of a basic block. Thus BB4 and the corresponding basic block features are generated. After several iterations, the monitor model for this code segment will be generated.

3.3 Microarchitecture Details

The aim of our architecture is to ensure the integrity of the instruction stream. Our architecture monitors the execution of the target program by automatically separating the binary code into basic blocks, and checking related properties to verify whether the program is executed properly and whether the integrity of the codes is corrupted. Before the online running of the software code, a tool written in Perl is utilized to decompile the binary file and separate the codes into basic blocks. Then the software extracts the information on basic blocks, and constructs the monitor model which is going to be downloaded into specified memory.

A basic block is a code fragment in which instructions are executed sequentially without a jump instruction except for the last instruction of the block. For a basic block, the first instruction is the entry of the basic block, and the last instruction of the basic block is the end which will lead the stream to jump or branch to a start of a new

basic block. It is possible that basic blocks overlap some other basic blocks, but it will not influence the consumption of memory or the overhead of the system performance. Information of basic blocks is the start address and the offline computed LHash value, both of them are the 16-bit message stored in monitor model memory.

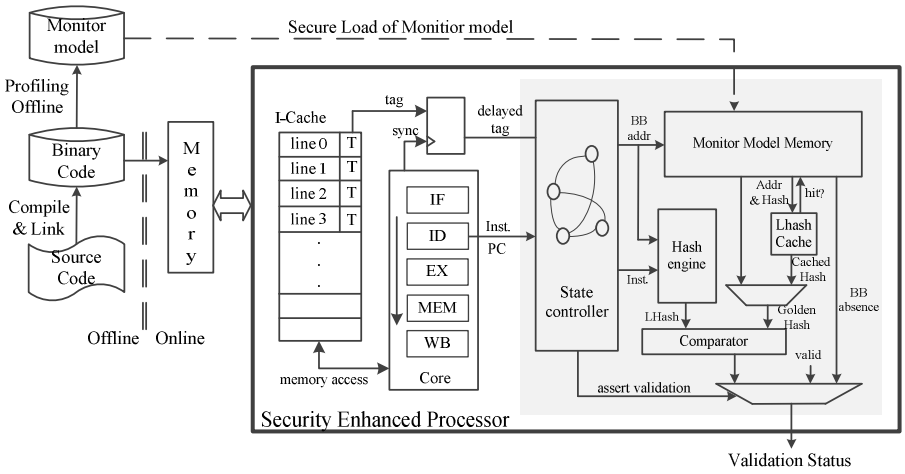

Figure 3: Details of the proposed microarchitecture

Some details of the proposed microarchitecture are shown in Figure 3. In order to be more related to modern industry, an embedded processor with an in-order five-stage pipeline is used here. Our architecture continuously extracts binary instructions and PC addresses from the Instruction Decode stage. Then, the extracted instructions are computed by the LHash module, and the start address of the current basic block is compared with the pre-extracted valid start address in memory of the monitor model. Then the dynamic computed LHash value will be compared to the pre-stored information. The monitor sends a validation status signal after the comparison. An illegal signal will be sent whenever a violation between results computed in the process of execution and those computed in the pre-process is detected. The illegal signal will be used to freeze the processor temporarily until the problem is processed. The integrity of the instruction stream should be checked by verifying instructions one after one theoretically, but because of the restriction on the consumption of memory and instantaneity of the response to the violation, our architecture implements the LHash algorithm to check the integrity of the instruction stream based on basic blocks. In order to further reduce the performance impact of the security enhanced system, ISIC will sample the delayed I-Cache tag signal to bypass the hash match check of the cached and checked basic blocks.

The LHash Algorithm is a lightweight cryptographic algorithm with less overhead on the speed and source consumption. During the absorbing period, the binary instructions extracted from the ID stage will be firstly XORed with part of the internal message one by one. When a jump instruction is detected by the architecture, the current basic block ends, and our architecture will terminate the absorbing period to start a squeezing period. Then the LHash value of the basic block will be calculated

and outputted in the squeezing period. In the pre-process, the binary instructions are calculated in the same way as that in the execution. After the value during the execution is executed, the LHash value will be compared with the one from a basic block with a matching start address. If any violation between two LHash values is found, it is considered that the code is attacked and the architecture will send an illegal signal. If there is no difference between such two values, a legitimate signal will be sent, and the processor will run as usual.

For implementation, a primitive message is initialized first, and is used to XOR all forthcoming binary instructions. The primitive message is a binary message padded by a single bit 1 mixed with necessary 0 bits. In the absorbing period of the algorithm, a newly coming instruction will be XORed, and then the message is permuted by a predetermined internal permutation. Such process is executed once whenever a new instruction is extracted from the ID stage. Here only the bits indicating the type of the instruction will be absorbed by the message because run-time operands are unpredictable in the compilation, and are not pre-stored in the specific memory. After a jump instruction is detected and absorbed, the squeezing period begins, and the LHash value of the basic block is calculated from the message.

The search of the current basic block information in the software monitor model also consumes some clock cycles, which may affect the performance of the system. A hardware implementation of the LHash cache is used to optimize the problem. The LHash cache is a ring buffer and is used to suspend few of the latest basic blocks' information. If the basic blocks that have cached information are executed again, they are no longer need to be searched from the software monitor model. When the calculation of the LHash value of the basic block is over, the LHash value will be compared to the pre-stored LHash value of the basic block. A stall signal will be sent when the difference between two values is found as difference here is regarded as the existence of a corruption in the basic block. In this case, the violation indicates at least one instruction of the basic block is corrupted by attacks or electromagnetic radiation. The timing diagram of the worst case in the runtime integrity validation is shown in Figure 4.

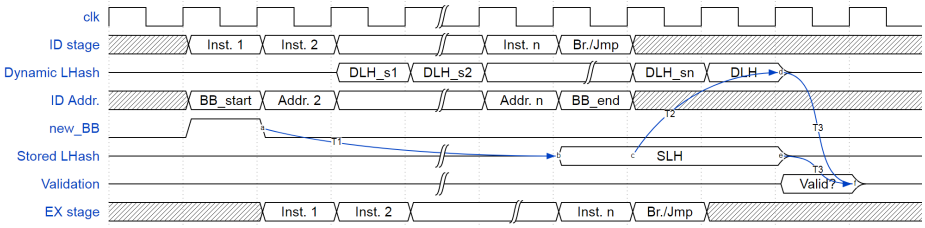

Figure 4: Timing diagram of the worst case in the runtime integrity validation

The worst case occurs only when the I-Cache and the LHash cache are both misses for the current basic block. T1 represents the basic block retrieval time from the software monitor model. Once a new basic block is detected, the starting address will be used for the search task. T2 represents the time to complete the dynamic hash calculation since the golden LHash is ready. T3 indicates the time that the dynamic monitor model is compared with the static one and output the validation status. Once

the monitor detects a possible violation behavior, it asserts the alert signal to generate a highest-priority interrupt to the processor. Moreover, this signal will trigger an interrupt response, such as terminating the program and switching the processor to a secure mode. The frozen signal is asserted when the monitor loses the pace of processor execution. This signal is treated as the normal state, while all pipeline stages are frozen till the monitor catches up with the processor.

Our architecture runs parallel to the processor when it searches for the corresponding information of the executing block and calculates in the absorbing period of the LHash algorithm. The squeezing period and a simple comparison between executing the information of the LHash value, the end address of the basic block and the start address of the successive basic block is executed after the basic block is over.

3.4 Design Flow

In this part, we describe the proposed design flow in steps by which the embedded system is enhanced with the security of code runtime integrity. Some code analysis processes have a close relationship with the processor instruction set, and we describe OR1200, one OpenRISC 1000 architecture processor, as an example in the illustration.

The first step of the design flow is to process the compiled binary code and get the basic block details required in the monitor model. At first, a script is designed to ensure whether the format of the program file is correct, and is targeted toward the OR1200 platform. Then a GNU tool or32-elf-objdump is used to disassemble the binary code, and the regular expression is employed to search all jump instructions. The basic block starting and ending addresses can be determined by the details as listed in Table 3. As OR1200 has a delay slot after jump instruction, the process is slightly different. The GNU tool can be used to give the starting address of functions according to the program's symbol table, thus the script will get this kind of basic block starting address indirectly. The interrupt entry address can be also inferred by OR1200 standards. At the same time, the script will get the starting and ending addresses pair of the basic block by sorting them. Then, a software implemented LHash function is referred to calculate the golden LHash value of each basic block. Finally, the script will output the monitor model arranged in a specific format. At the end of the preparation stage, we employ a specified Makefile to complete the compile and monitor model extraction work. Besides, some works such as code selection, compiling, target program analysis and program initiation will be completed at this stage.

Instructions	Starting Address	Ending Address
l.j	jump target address	next address
l.bf/ l.bnf/l.jal/ l.jalr	jump target address and the next address	next address
l.jr	None	next address
l.rfe	None	current address

Table 3: The details to determine the starting and ending addresses

Following the first preparation stage, the second stage is the testing. In order to obtain the execution time of the basic block, and to solve the problem of the basic block loss as much as possible, we must first go through a test stage before the formal operation. At this stage, the program will run in the real environment, and the execution time in clock cycles of each basic block will be measured by the security module. The security module will also record the missing basic block. The information obtained in the testing stage will be recorded and used to update the monitor model. Of course, when talking about tests, we will face the test coverage problem. This design does not require full test coverage of basic blocks, because we have both the basic block starting address verification mechanism and code checksum validation mechanism. In any case, higher coverage among basic blocks will mean that the system will be of higher security and accuracy.

At the formal operation stage, the updated monitoring model has been loaded into the specified monitor model memory. In our experiment environment, the monitoring model is placed in the FPGA BRAM, and therefore the proposed ISIC can easily access it. First, the security module will detect the beginning of the program. After beginning, the security module will detect jump instruction. It will treat next instruction after jump as the basic block ending address, and the second instruction after jump as the next basic block starting address. Whenever a new basic block begins and information search is completed, the security module will check the LHash value, and the next basic block starting address. Exception message is recorded if abnormal situation has been met. Finally, when the program ends, the security module will output the exception message.

The design flow of the proposed approach is based on the OR1200 processor. As described before, the proposed hardware based monitoring architecture only traces the executing instruction and its corresponding address signals of the pipeline, and can detect runtime code tampering automatically. This method can work effectively without any requirement of the modification on the compiler or the processor core. The proposed ISIC is a hardware module that is independent of the processor. If the embedded system developers try to transplant the security method into other systems, they only need to adjust off-line static extraction and analysis tools according to the target instruction set. Because different target processor may have different instruction set, and the division strategy of basic block is strictly depend on the jump and branch instruction. Besides, the online hardware hardly needs any modification, except for the automatic recognition of the start and end of the basic blocks related to the instruction set. Thus the proposed hardware security approach for runtime code integrity is of high scalability and can be easily ported to any processor platforms.

4 Attacks and Security Analysis

After construction of the proposed architecture, the efficiency of the architecture is analyzed based on different common threat models. Code injections have always occupied a big proportion of all kinds of attacks. With the development of the technology, buffer overflow attacks based on stack smashing and advanced attacks such as code reused attacks flourish today. In this section, defenses against some common attacks are presented by analysing attack scenarios.

There are only a few sets of signals between the proposed security module and the processor. In our work, the security boundary is the embedded processor chip, thus any direct attacks on the security enhanced processor, such as changing the internal state or tampering monitor models are considered impossible. But this also causes the problem that the monitor model cannot be modified directly by the developer. This is acceptable for the security consideration of the embedded system. If the monitor model needs to be updated, we need to add a set of direct connections from the security module to the outside through a credible process. Security of the credible load of the monitor model is not concerned with this work.

Attacks tampering branch and jump instructions can be categorized as either starting address absence violation or hash mismatch violation. If the operation code of the processor is tampered to become a different type of instruction, it will be identified as an address absence violation. And the corresponding basic block golden hash value cannot be derived from the monitor model successfully. The end position of the current basic block will not be correctly identified, which causes the instructions within the basic block to be considered to be much more, resulting in the hash validation failure.

The code injection attack is one of the most common attacks. The adversaries inject the code into executable codes and change the original intent of the basic block into the malicious one. When any code is injected into executable codes, at least the LHash value of one of the basic blocks must be changed. It is computationally infeasible for LHash value collision attacks when the number of instructions is not the same. Although some adversaries can modify a block of instructions within a basic block to fulfil the match of the cryptographic hash and the relative address, they may induce the illegal instructions cannot be executed by the processor or invalid access addresses outside the space bounds. When the LHash value is calculated and compared in the implementation, the difference between the corrupted basic block's new LHash value and the pre-stored LHash value can be found. Then the attack can be detected, and the CPU will be frozen by receiving a stall signal, and thereby the proposed architecture can detect code injection attacks.

Stack-based, Heap-based buffer overflows, dangling pointer references, format strings vulnerabilities are normal overflow attacks. The purpose of this kind of attacks is to run the injected code not originate from the source program. Generally, the injected code has been placed in stack or head, which is not in the executable range defined by the security module. The security module can detect this kind of attacks successfully. Return-to-libc, heap-spraying, non-control data attacks are advanced overflow attacks. The security module can only defense heap-spraying attacks, because its purpose is also to run the injected code. Return-to-libc and non-control data attacks have never injected code or tampered code. Therefore, the security module will often fail to protect the program.

As mentioned above, most code tampering will fail. However, there are still three situations requiring more consideration. a) The security enhanced processor cannot detect the tampering immediately. This is because the basic blocks are the checking unit of the monitor model, so as long as the program runs to the ending address of the basic block it should detect the attack. Even if the jump or branch instruction has tampered, the regular work of the security module will not be affected until the new end of the basic block is detected. b) In the case of the loss of the basic block, the

proposed ISIC cannot derive the monitor model of the current basic block continuously. Because of lack of protection of strong code checksum value, the attack may be successful. If we want to prevent such attacks completely, we should make sure that all basic blocks have been detected during the preparation and the testing stages. Besides, we also need to set out the security policy to treat the loss of the basic block as an abnormal situation. c) Collision attacks of LHash values suffer from algorithm collision attacks. This attack will succeed if the attacker replaces a code of the basic block, and it is guaranteed that the checksum value did not change. This is very difficult to implement in the process of protecting the processor instructions. The attacker must ensure that the input message sequences are legal instructions that can be executed by the processor, while the attacker has a collision attack. Although we only have 16 bits for the LHash algorithm, it is reasonable to believe that it is not likely to be compromised considering the limited number of instructions and meaningful instruction combinations. Some advanced hash algorithm will have a lower success rate of collision attacks. However, if high security is needed, it can be upgraded to a 32-bit or higher checksum algorithm or advanced algorithms like SHA and MD5, although this will cause performance loss and on-chip memory resource consumption.

5 Experiment Results

In terms of platform building, the OR1200 processor which is a 32-bit scalar RISC with a Harvard micro architecture is used in this work. The OR1200 soft core is configured with 4KB Instruction Cache and 4KB Data Cache. The frequency of the core is @100 MHZ. Then the SoC with the proposed architecture is implemented on the Xilinx Virtex 5 FPGA platform. The configuration of the system on chip is listed in Table 4. The FPGA resource used for the proposed instruction stream integrity checker is shown in Table 5.

Content	Details
CPU	OR1200 (svn rev 853) @100 MHz with 4K I/D-$
SoC	based on ORPSOCv2
ISIC	16 BB LHash Cache
FPGA board	Digilent Genesys XC5VLX50T

Table 4: The configuration of the system on chip

Slice Logic Utilization	Used	Available
Slice Registers	421	28,800
Slice LUTs	1,428	28,800
occupied Slices	469	7,200
LUT Flip Flop pairs used	854	-
bonded IOBs	127	480
BlockRAM/FIFO	48	60
BUFG/BUFGCTRLs	2	32
Average Fanout of Non-Clock Nets	4.21	-

Table 5: FPGA resource used for the proposed ISIC

Benchmarks	Total Instructions	Jump & Branch	Total BB	Memory Size(KB)
AES	22170	2926	3535	13.81
openECC	56313	5439	6734	26.30
quicksort	6707	854	1018	3.98
bitcount	19684	2760	3344	13.06
blowfish	19128	2685	3247	12.68
patricia	23130	3288	3853	15.05
SHA1	20455	2822	3400	13.28
FFT	13506	1818	2143	8.37
CRC16	18941	2672	3231	12.62
basicmath	26515	3667	4327	16.90
average			3483	13.61

Table 6: Basic blocks and storage information of the selected benchmarks

To evaluate the performance and area overhead of the proposed architecture, we select various scales of benchmarks from Mibench suite to generate realistic workloads. The basic block information, including the total instructions numbers, jump and branch numbers, total basic block numbers, and average instruction numbers in each basic block, of the selected benchmarks is listed in Table 6. OpenECC has the largest number of basic blocks, and the on chip storage requirement is 26.30KB. The fifth column is the memory space used for the on chip monitor model. This is generally accepted for runtime security monitor.

The proposed architecture monitors the validation of the program at runtime, so that any modification committed from installation to execution can be checked. The dynamic monitoring signal is directly derived from the decode stage of the pipeline, and any invalid execution resulting from the previous attacks can be detected. Our architecture runs parallel with the processor, so that the overhead is affordable most

of the time. The selected benchmarks used in this experiment and the performance overhead are shown in Figure 5.

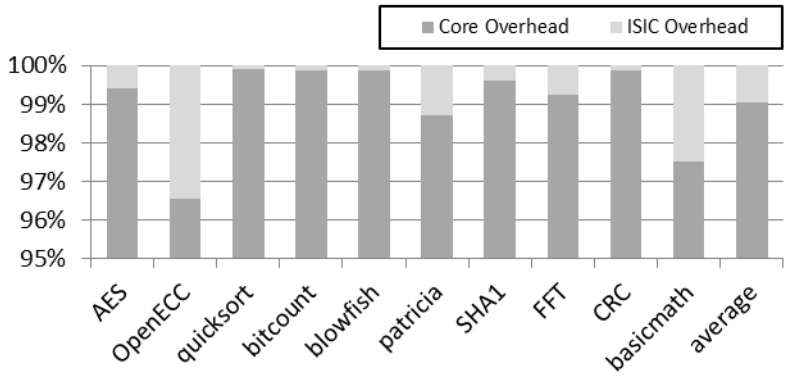

Figure 5: The performance overhead of the core and ISIC

The total performance overhead across the selected benchmarks is 2.50%, ranging from 0.04% (quilksort) to 3.45% (OpenECC). It is obvious that the overhead of OpenECC and basicmath is more than others. That is because the numbers of basic blocks in OpenECC and basicmath are much more than those of others, as can be seen from Table 6. As the number of basic blocks increases, the overall overhead of the proposed ISIC shows an upward trend. On the one hand, the large number of basic blocks increases the time spent in monitoring model retrieval. On the other hand, the program with more basic blocks has more irregular jumps and leads to a decrease in instruction cache and hash cache hit ratio. Besides, some complicated math programs can also lead to a decrease in monitoring performance. For the program like quicksort, bitcount, blowfish and CRC, the algorithm implementation process of these programs needs to conduct a lot of repetitive computing operations, and thus the ISIC performance overhead of these programs is low. In [Rogers and Milenkovic, 2009], the authors implement their work in Sim-Panalyzer ARM Simulator. The performance overhead in CBC-MAC CICM WtV mode, which is one of the most similar patterns of our work, is 43.2%, ranging from 0.17% to 93.8%, with the parameter of 4KB I-$. In [Arora et al., 2006], the authors evaluated the performance impact using the SimpleScalar 3.0/PISA architectural simulation tools, and the processor model is ARM920T with 16KB L1 I-$. In their work, the worst case performance overhead is 4.94%. Compared with these two works, our approach has smaller performance losses or lower hardware requirements.

A limitation of our approach is that only after a basic block is complete, can our architecture check the integrity of the instruction stream. Therefore, when malicious codes are injected or instructions are tampered, the processor has to execute the basic block at least one time because verification of the LHash value is executed at the end of basic blocks. A shadow register mechanism is a solution for this limitation in our architecture. When a basic block is being executed, results of instructions which do not update the memory will be stored in a shadow register file. Instructions for updating the memory will be stored in a specific store buffer until the LHash value is

checked by our architecture. After the validation of the LHash value of the basic block, the values of the shadow register file will be used to update the registers, and the data in the buffer will be sent to the external memory sequentially.

6 Conclusions

This paper proposes a fine-grained hardware based approach for runtime code integrity in embedded systems. The approach can perform validation of program runtime integrity by offline profiling of program features and runtime integrity check. At offline profiling stage, the binary code is divided into basic blocks, and the security features such as starting address and LHash based checksum are extracted. This security sensitive information will be load to the specified monitor model ram at the program load time. Then, an instruction stream integrity checker is designed for rapid dynamic integrity check and any invalid execution of the program will be detected to trigger the corresponding exception signals.

To evaluate the performance overhead of the proposed approach, we implement a SoC with ISIC architecture in Xilinx XC5VLX50T FPGA as a prototype system, which can provide low area and power overhead, and high performance of verification. We use various scales of benchmarks from the MiBench suite to generate realistic workloads for the processor. The implementation shows that the ISIC can detect all attacks on instruction stream integrity. The processor performance overhead induced by the security mechanism of the ISIC is less than 3.45%, and is less than 1% at most of the time, according to the selected benchmarks. Besides, the prototype system also has advantages of low power consumption and marginal area footprint. The proposed hardware based monitoring architecture only traces the executing instruction and its corresponding address signals of the pipeline, with less intrusion to the existing embedded processor architectures, and can be easily ported to any processor platforms.

Acknowledgements

This research is supported by the Key Project of National Science Foundation of China (Grant No. 61232009), the National Science Foundation of China (Grant No. 60973106, and No. 81571142), National High-tech R&D Project of China (863 Grant No. 2011AA010404).

References

[Abadi et al., 2005] Abadi, M., Budiu, M., Erlingsson, U. and Ligatti, J.: “Control-flow integrity”; Proc. CCS (2005), 340–353.

[Ahn et al., 2014] Ahn, Y., Lee, Y., Choi, J., Lee, G., Ahn, D.: “Monitoring translation lookahead buffers to detect code injection attacks”. Computer, 47, 7 (2014), 66-72.

[Andriesse et al., 2015] Andriesse, D., Bos, H. and Slowinsha, A.: “Parallax: Implicit code integrity verification using return-oriented programming”; Proc. 45th IEEE/IFIP Int. Conf. on Dependable System and Networks (2015), 125-135.

[Arora et al., 2006] Arora, D., Ravi, S., Raghunathan, A. and Jha N.K.: "Hardware-assisted run-time monitoring for secure program execution on embedded processors"; IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 14,12 (2006), 1295-1308.

[Bletsch et al., 2011] Bletsch, T., Jiang, X., Freeh, V. W. and Liang Z.: "Jump-oriented programming: A new class of code-reuse attack," Proc. ASIACCS (2011), 30–40.

[Bogdanov et al., 2013] Bogdanov, A., Knezevic, M., Leander, G., Toz, D., Varici, K. andVerbauwhede, I.: "SPONGENT: The design space of lightweight cryptographic hashing", IEEE Transactions on Computers, 62, 10(2013), 2041-2053.

[Das et al., 2016] Das, S., Zhang, W. and Liu, Y.: "A fine-grained control flow integrity apporach against runtime memory attacks for embedded systems". IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 24, 11 (2016), 3193-3207.

[Davi et al., 2014] Davi, L., Koeberl, P. and Sadeghi, A.-R.: "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation". Proc. DAC (2014), 1–6.

[Dor et al., 2003] Dor, N., Rodeh, M. and Sagiv, M.: "CSSV: Towards a realistic tool for statically detecting all buffer overflows in C"; Proc. ACM SIGPLAN Conf. Program Programming Language design and Implementation (2003), 97-106.

[Fiskiran and Lee, 2004] Fiskiran, A. and Lee, R.: "Runtime execution monitoring (REM) to detect and prevent malicious code execution"; Proc. IEEE Int. Conf. Computer Design (2004), 452–457.

[Guo et al., 2011] Guo, J., Peyrin T. and Poschmann, A.: "The PHOTON family of lightweight hash functions", Advances in Cryptology, LNCS 6841 (2011), 222-239.

[Guthaus et al., 2001] Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T. and rt al.: "MiBench: A free, commercially representative embedded benchmark suite"; Proc. IEEE Int. Workshop on Workload Charactertization (2001), 3-14.

[Kanuparthi et al., 2012(a)] Kanuparthi, A., et.al, "A High-Performance, Low-Overhead Microarchitecture for Secure Program Execution,"; Proc. on IEEE Intl. Conf. on Computer Design, , IEEE (2012), 102–107.

[Kanuparthi et al., 2012(b)] Kanuparthi, A. K., Zahran, M. and Karri, R.: "Architecture support for dynamic integrity checking"; IEEE Transactions on information forensics and security, 7 (2012), 321-332.

[Kirovski et al., 2002] Kirovski, D., Drinić, M. and Potkonjak, M.: "Enabling trusted software integrity"; Proc. 10th Int. Conf. Architectural Support for Programming Languages and Operating Systems (2002), 108–120.

[Li et al., 2016] Li, D., Zhan, X., Tong, Q., Zou X. and Liu, Z.: "The design and implementation of embedded security CPU based on multi-strategy"; Chinese Journal of Electronics, 25, 5(2016), 801-806.

[Lie et al., 2000] Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J. and Horowitz, M.: "Architecturual support for copy and tamper resistant software"; Proc. 9th Int. Conf. Architectural Support for Programming Languages and Operating Systems (2000), 168–177.

[Mao and Wolf, 2010] Mao, S. and Wolf, T.: "Hardware support for secure processing in embedded systems", IEEE Transactions on Computers, 59, 6 (2010), 847-854.

[Oh et al., 2002] Oh, N., Shirbvani, P., McCluskey, E.: "Control-flow checking by software signatures"; IEEE Trans. on Reliability. 51. 2 (2002), 111-122.

[Rogers and Milenkovic, 2009] Rogers, A., and Milenkovic, A.: "Security extensions for integrity and confidentiality in embedded processors"; Microprocessors and Microsystems, 33 (2009), 398-414.

[Sankaran and Calix, 2016] Sankaran, R., and Calix, R. A.: "On the fesibility of an embedded machine learning processor for intrusion detection"; Proc. IEEE Int. Conf. on Big Data (2016), 1082-1089.

[Serpanos and Voyiatzis, 2013] Serpanos, D. N., Voyiatzis, A. G.: "Security challenges in embedded systems"; ACM Trans. on Embedded Computing Systems, 12, 1s (2013), 1-10.

[Shacham, 2007] Shacham, H.: "The geometry of innocent flesh on the bone: Return into-libc without function calls (on the x86)", Proc. CCS (2007), 552–561.

[Shaye et al., 2007] Shaye, A., Moseley, T., Reddi, V., Blomstedt, J. and Connors, D.: "Using process-level redundancy to exploit multiple cores for transient fault tolerance"; Proc. Int. Conf. on Dependable Systems and Networks (2007), 297-306.

[Shehab and Batarfi, 2017] Shehab, D.A., Batarfi, O.A.: "RCR for Preventing Stack Smashing Attacks Bypass Stack Canaries"; Proc. on Computing Conference (2017), 795-800.

[Wang et al., 2008] Wang, X., Tehranipoor, M., Plisqieilic, J.: "Detecting malicious inclusions in secure hardware: challenges and solutions"; Proc. on IEEE Int. Workshop on Hardware-Oriented Security and trust, Anaheim (2008), 15-19.

[Wang et al., 2013] Wang, X., Zhao, Z., Lu, Y. and Zhang, Y.: "A Design of Security Module to Protect Program Execution in Embedded System"; Proc. IEEE Int. Conf. on GreenCom-iThings- CPSCom (2013), 1750-1755.

[Wang et al., 2016] Wang, X., Pang, S., Wang, W., Zhao, Z., Zhou, C., He, Z. and et al.: "Hardware-assisted system for program execution security of SoC"; Proc. on ITM Web Conf. (2016), 1-6.

[Wu et al., 2014] Wu, W., Wu, S., Zhang, L., Zhang, J. and Dong, L.: "LHash: a lightweight hash function"; Lecture Notes in Computer Science, 8567 (2014), 291-308.

[Zitser et al., 2004] Zitser, M., Lippmann, R. and Leek, T.: "Testing static analysis tools using exploitable buffer overflows from open source code"; Proc. ACM Int. Symp. Foundations Software Engineering (2004), 97-106.