A Hardware Security Protection Method for Conditional Branches of Embedded Systems

Qiang Hao , Dongdong Xu , Yusen Qin , Ruyin Li , Zongxuan Zhang , Yunyan You , Xiang Wang *

Posted Date: 16 April 2024

doi: 10.20944/preprints202404.1017.v1

Keywords: embedded system; branch prediction unit; conditional branch; jump address; jump direction

 doi:10.20944/preprints202404.1017.v1

Article

A Hardware Security Protection Method for Conditional Branches of Embedded Systems

Qiang Hao, Dongdong Xu, Yusen Qin, Ruyin Li, Zongxuan Zhang, Yunyan You and Xiang Wang *

School of Electronic and Information Engineering, Beihang University, Beijing 100191, China; haoqiang1994@buaa.edu.cn (Q.H.); xudongdong1994@buaa.edu.cn (D.X.); qinys@buaa.edu.cn; ruyinli@buaa.edu.cn; zzxuan@buaa.edu.cn; yyy614@buaa.edu.cn

* Correspondence: wxiang@buaa.edu.cn; Tel.: +86-10-8231-3686

Abstract: The branch prediction units **(**BPUs) generally have security vulnerabilities, which can be used by attackers to obtain the execution status, jump directions, and jump address of conditional branches, and the existing protection methods cannot defend against these attacks. Therefore, this article proposes a hardware security protection method for conditional branches of embedded systems. This method calculates the number of updates to the branch target buffer (BTB). When it exceeds the threshold, BTB is locked to prevent attackers from analyzing the execution status of branches based on the time difference of whether BTB is updated. Moreover, the hybrid physical unclonable function (PUF) circuit is designed to provide confidentiality protection for the jump directions, jump addresses, and their indexes, preventing attackers from stealing these critical data. If these mechanism fails and attackers successfully tamper with conditional branches, this paper proposes a control flow integrity (CFI) protection mechanism based on branch labels to timely detect tampering with instruction codes, jump addresses, and jump directions. The proposed method is implemented and tested on FPGA. The experimental results show that this method can achieve fine-grained security protection for conditional branches, with about 5.4% resource overhead and less than 5.5% performance overhead.

Keywords: embedded system; branch prediction unit; conditional branch; jump address; jump direction

1. Introduction

Most existing embedded systems use branch prediction units (BPUs) to predict the jump addresses and jump directions of conditional branches, which can improve the task processing speed [1]. However, with the discovery of the Spectre and Meltdown vulnerabilities [2,3], these high-performance processor architectures have the security risk of sensitive data leakage. Attackers can inject malicious code, steal confidential information, tamper with critical data, and compromise the control flow integrity (CPI) of a program through these vulnerabilities that exists in the BPUs [4]. Therefore, embedded system designers should consider security issues at the chip architecture level and adopt secure and efficient protection methods to address these security vulnerabilities [5]. Branch History Table (BHT) and Branch target buffer (BTB) are important modules of the BPU. BHT is used to predict whether the conditional branches will be taken, that is, to predict the jump directions of branches. BTB is used to predict the jump addresses of conditional branches. Attackers can analyze the data of BHT and BTB to obtain the prior knowledge required for constructing malicious code and stealing sensitive information [6,7]. To defend against these attacks, various software-based or hardware-based security protection methods have been proposed [8,9]. Compared with software-based methods, hardware-assisted methods use fewer resources and have faster speed [10]. Therefore, our research is based on hardware-assisted security protection method.

Members of our lab have previously proposed various hardware security protection methods for embedded systems [11–14], but when applied, it was found that attackers could still exploit the vulnerabilities of BPUs to perform attacks. Inspired by existing research [15–18], we started to focus

 doi:10.20944/preprints202404.1017.v1

on the security of the jump directions and jump addresses of branches. We tried to restrict the target address of the jump instruction to be the starting address of a basic block (BB). This method could defend more than 90% attacks against the jump instructions, but the performance overhead was more than 9% [10]. We tried to integrate data integrity protection and control flow integrity protection, which could effectively protect the jump instructions and their jump addresses, but it could not guarantee the security of jump directions [19]. We tried to use the physical unclonable function (PUF) for lightweight cryptographic protection of the jump addresses and jump directions. However, it has been pointed out that existing PUF circuits are insecure under targeted modeling attacks [20].

Considering the attacks on BPUs and the shortcomings of existing protection methods, we believe that the following challenges need to be addressed to fully secure the conditional branches of embedded systems.

- Prevent attackers from obtaining the execution status of branches;
- Prevent attackers from stealing the jump information of branches;
- Detect the tampering attacks on branch information.

To solve the above security challenges, this article proposes a novel hardware security protection method for conditional branches of embedded systems, which can well balance the protection capability, performance overhead and resource overhead.

The main contributions of this article are as follows.

- The branch information protection mechanism based on hardware locking is proposed. When the program is running, this mechanism will calculate the number of updates to BTB. If it exceeds the set threshold, BTB will be locked, preventing attackers from filling up BTB through spy processes, and obtaining the execution status of branches by analyzing the time difference of whether BTB is updated;
- The branch information protection mechanism based on dynamic isolation is proposed. This mechanism encrypts the branch information (such as branch addresses, jump addresses, and their indexes) and achieve dynamic isolation. This article designs the hybrid Arbiter PUF (APUF) circuit that provides the keys. This circuit can confuses the relationship between the input challenge and the output response, and improves the ability to resist machine learning attacks;
- The control flow integrity protection mechanism based on branch labels is proposed. When the program is offline, generate static labels based on the instruction codes and the jump addresses. When the program runs, the designed hardware security monitoring module will generate dynamic labels according to the same process. If the dynamic and static labels are inconsistent, it is considered that the control flow has been disrupted and the CPU is reset.

The rest of this article is shown as follows. Section 2 describes the related work. Section 3 presents the threat model. Section 4 provides a detailed description of the protection method proposed in this article. Section 5 presents the experimental results. Finally, Section 6 concludes the main contents of this article.

2. Related Work

This section describes the current state of research related to conditional branch security protection methods, including preventing obtaining the execution status of branches, encrypting jump information, and monitoring the integrity of jump information.

2.1. Prevent Obtaining the Execution Status of Branches

Attackers can exploit the vulnerabilities of BPU (such as Spectre [2]) to implement branch prediction analysis (BPA) attacks [21] to obtain the execution status of branches.

An architectural support scheme against the BPA attacks was proposed [21], which dynamically identified spy processes injected by attackers and prevented them from filling up the BTB. Although the resource overhead of this method is low, it is difficult to balance the protection capability and performance overhead due to the simple mechanism of identifying spy processes. Conditional Speculation [15,23] dynamically identified dangerous access instructions by implementing different filtering mechanisms, but had a high performance overhead for some test programs. BRB [24] prevented malicious data injected by attackers from being used by assigning separate branch history

 doi:10.20944/preprints202404.1017.v1

tables (BHTs) to different programs, but had a high resource overhead and could not provide the effective protection for the shared parts.

Therefore, in order to prevent attackers from injecting spy processes, analyzing side channel information such as BTB update time differences, and ultimately determining which branches of the program have been executed, it is necessary to design appropriate hardware protection mechanisms to accurately identify spy processes injected by attackers and prevent them from fully obtaining the execution status of branch.

2.2. Encrypt the Branch Information

The information stored in the BPU is usually stored in plaintext, which can be shared between different threads, posing a serious risk of information leakage.

Zhao et al. [16] cryptographically protect the content and index of the BPU by using randomized lightweight processing to prevent attackers from accessing critical jump information. However, they used a simple hardware random number generator, which makes it difficult to adequately secure the ciphertext. Researchers in our lab proposed an efficient cryptographic accelerator that could protect dynamic data security of embedded systems [25]. After that, they proposed two post-quantum cryptographic algorithms that could improve higher levels of cryptographic protection for critical data. However, although the hardware circuit has been optimized, the resource cost and the time of encryption and decryptions are still unacceptable for BPU.

The PUF can generate random numbers by using uncontrollable process deviations introduced during the manufacturing process, and these random numbers can be used to encrypt critical branch information. HCIC [18] used PUF to encrypt the return addresses and jump addresses, which could resist ROP and JOP attacks. However, it did not describe the PUF circuit structure in detail. The PUFs are divided into weak PUFs and strong PUFs according to their ability to generate challenge response pairs (CRAs) [26]. Strong PUFs can generate an exponential number of CRPs, mainly including Arbiter PUF (APUF)[27], Ring Oscillator PUF (RO-PUF)[28], etc. The disadvantage of APUF is its vulnerability to the modeling attacks based on machine learning. To counteract modeling attacks, researchers proposed structures such as Feed Forward APUF [29], and Heterogeneous APUF [30]. However, it was demonstrated that traditional APUFs and their variant structures had been predicted successfully with the rate up to 95% or more under targeted machine learning attacks [20]. Compared with APUF, RO-PUF is more secure, but the resource overhead is too large for lightweight applications [31].

Therefore, according to the actual application requirements of the embedded system, it is necessary to consider the security, performance, resources and other indicators, and design a suitable hardware circuit structure to realize the confidentiality protection of the branch information.

2.3. Monitor the Integrity of Branch Information

The integrity of jump information means that the instruction codes, jump addresses and jump directions of the conditional branches have not been tampered with when the embedded system executes the program.

A fine-grained hardware protection method for instruction code integrity was proposed by wang et al [32]. They used the LHash [33] function to tag the instruction codes and generated unique labels. If the value of the label changes, it means that attackers have successfully tampered with the corresponding code. After that, they proposed a security monitoring and fault recovery architecture for run-time program execution [10], which could safeguard the integrity of partial jump instructions by limiting the target address range. However, its performance overhead was more than 9%. A security method that could monitor both data integrity and control flow integrity was proposed [13], which not only tagged the instruction codes, but also generated control flow integrity (CFI) labels based on jump relationships. To further optimize the performance overhead and resource overhead, the researchers tagged the jump addresses and generated labels [19]. When the embedded system executes the program, if attackers tamper with the instruction codes or the target addresses, the designed security monitoring module will detect the change of the corresponding labels in time.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 April 2024 doi:10.20944/preprints202404.1017.v1

Although the above methods can effectively monitor the integrity of instruction codes and jump addresses, they cannot determine whether the jump directions have been tampered with. Therefore, a more secure and effective hardware monitoring architecture needs to be designed to achieve the fine-grained security protection of conditional branches with lower performance overhead and resource overhead.

3. Threat Model

This section describes the embedded processor used by this article, the structure of the BPU, and the security threats to BPU.

3.1. Xuantie E906 Processor

The conditional branch instructions supported by the E906 are shown in Figure 1 and include BEQ, BNE, BLT, BLTU, BGE, BGEU, C.BEQZ and C.BNEZ. To obtain the internal details of the embedded processor core, we select the Xuantie E906 Processor [34] for research, which is a open source RISC-V processors. The E906 uses a 5-stage pipeline for integer operation, which includes IF (Instruction Fetch), ID (Instruction Decode), EX (Instruction Execution), MEM (Memory access), WB (Write Back). In this article, we focus on the branch prediction in the IF stage and the instruction monitoring in the ID stage.

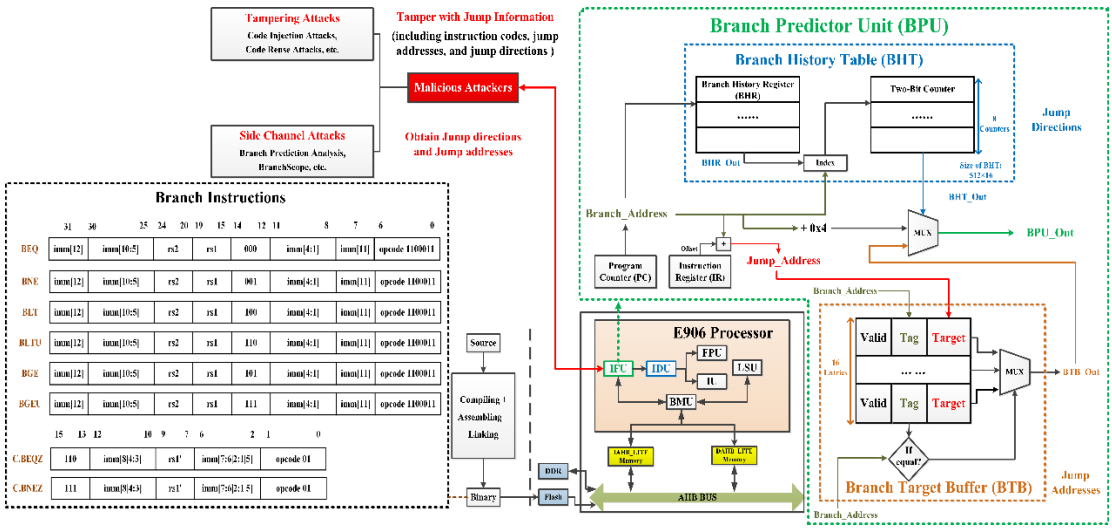


Figure 1. The security threats considered in this article.

3.2. Branch Prediction Unit

The structure of the BPU used in E906 is shown in Figure 1. BPU is located in the Instruction Fetch Unit (IFU) and consists of Branch History Table (BHT) and Branch Target Buffer (BTB). BHT is used to predict the jump directions of conditional branches. Branch History Register (BHR) is used to record the prediction trajectory and execution trajectory of the previous branches. BHT consists of eight 2-bit counters to predict whether a branch will jump or not. BTB is used to predict the jump addresses and contains 16 entries. Each entry contains Valid (indicates whether this entry is valid), Tag (is the lower 16 bits of Branch_Address and is used for indexing) and Target (is the lower 16 bits of Jump_Address). At first, IFU will determine whether the currently instruction is a branch based on its opcode. If it is a conditional branch, Branch_Address and BHR_Out are used as the index to find the corresponding 2-bit counter and predict whether this branch will be taken or not. Meanwhile, Branch_Address will be compared with the Tag of all BTB entries. If the Tag of a certain entry matches successfully, the Target of this entry is extract as BTB_Out. Finally, BPU_Out (which is the address of the next instruction) will be determined based on BHT_Out. If the branch should be taken, BPU_Out is Jump_Address. If the branch should not be taken, BPU_Out is Branch_Address + 0x4.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 April 2024 doi:10.20944/preprints202404.1017.v1

3.3. Security Threats

Figure 1 illustrates the main security threats faced by BPUs. Malicious attackers tamper with or obtain critical jump information (instruction codes, jump addresses, and jump directions) through the vulnerabilities in the BPU.

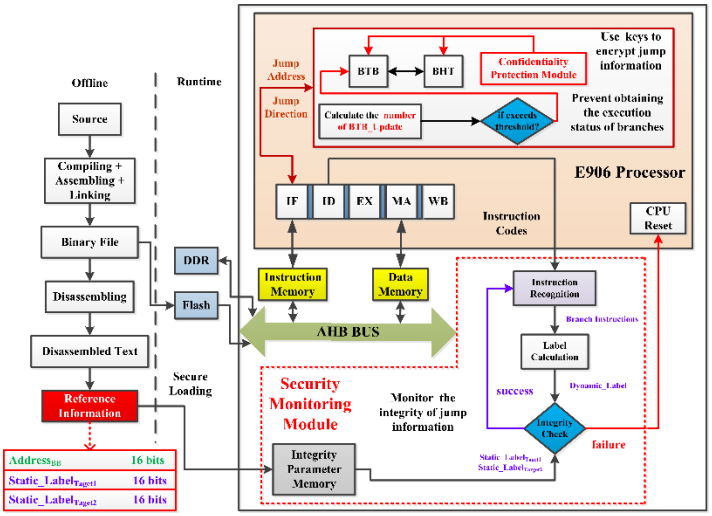
Typical tampering attacks include code injection attacks (CIAs) [35], code reuse attacks (CRAs) [36], etc. Attackers inject malicious codes into the embedded systems or use source program to construct gadgets, and successfully tamper with instruction code and critical registers (such as Program Counter, General Purpose Registers, etc.). These malicious actions will seriously affect the data integrity and control flow integrity of the program and cause security incidents. Side channel attacks is when attackers exploit the side channel vulnerabilities of the BPU to obtain the jump directions and jump addresses of the critical branches. Branch prediction analysis (BPA) [21] and BranchScope [22] are time-based side channel attacks. BPA infers whether a critical branch instruction is executed by the difference in update time between the spy processes and the normal processes in the BTB. BranchScope uses the similar principle to attack BHTs by analyzing time differences to determine the execution of critical branches and obtain jump address.

Researchers have proposed various security protection methods to address the above security threats, but either the protection scope is insufficient or the overhead of performance and resource is too high for lightweight embedded systems. Therefore, it is necessary to investigate a novel protection method that can address the major security threats faced by conditional branches with lower performance overhead and resource overhead.

4. The Hardware Security Protection Method for Conditional Branches

This article proposes a hardware security method that can protect embedded system processors from security threats against conditional branches. Figure 2 shows the overall structure of the proposed method. During program execution, the designed security module will dynamically calculate the number of BTB_Update within the certain clock cycle. If it exceeds the set threshold, BTB will be locked to preventing attackers from filling up BTB through spy processes and obtaining the execution status of branches by analyzing the time difference of whether BTB is updated. Meanwhile, the designed confidentiality protection module will encrypt the branch information inside BHT and BTB.

If the above protection mechanism fails and attackers successfully hijack the control flow of the program, the designed security monitoring module (SMU) will timely detect these attacks by comparing the labels of branch instruction, and initiate the processor reset to re-execute the program. When the program is offline, the reference information of the program will be extracted, including $Address_{INT}$ (the lower 16 bits of Branch_Address), $Static\_Label_{Target1}$ (the label when the branch will be executed), and $Static\_Label_{Target2}$ (the label when the branch will not be executed).



Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 April 2024 doi:10.20944/preprints202404.1017.v1

Figure 2. The overall structure of the proposed method.

When the binary file of the source program is loaded onto the Flash or DDR, all reference information will be stored in the Integrity Parameter Memory by the secure loading manner. When the program is running, SMU extracts the binary code and address from the instruction decode (ID) stage. SMU identifies whether the current extracted instruction is a conditional branch through the opcode shown in Figure 1. As for conditional branches, SMU calculates the Dynamic_Label, and compares it with Static_Label$_{Target1}$ and Static_Label$_{Target2}$. If Dynamic_Label is equal to Static_Label$_{Target1}$ or Static_Label$_{Target2}$, and matches the jump direction and address of this branch, it indicates that the control flow of the program is normal. If it is not equal or does not match the jump direction and address of this branch, it indicates that the control flow integrity (CFI) of the program has been compromised, and the processor need to perform the reset operation. By re-executing the program, the embedded system will restore to normal.

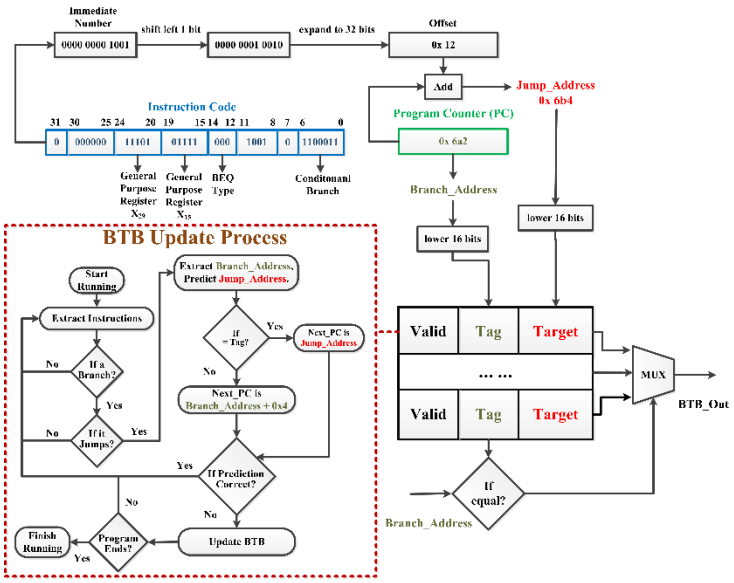
The following text provides the detailed introduction to the branch information protection mechanism based on hardware locking, the branch information protection mechanism based on dynamic isolation, and the control flow integrity protection mechanism based on branch labels.

4.1. The Branch Information Protection Mechanism Based on Hardware Locking

In this section, the proposed branch information protection mechanism based on hardware locking is introduced. This mechanism identifies the characteristics of attack behaviors, and initiates the hardware locking protection mechanism. For attacks such as BPA, the spy processes contain a large number of branch instructions far exceeding those of normal processes, which are used to fill the BTB. Therefore, it is possible to determine whether there is a spy process by the number of BTB_Update. If the number of updates to BTB exceeds the threshold, it is considered that there is a spy process present, and the 1-bit Lock field of the BTB will be set to 1, which can prevent any process from modifying the BTB. In this way, the mechanism can prevent attackers from filling up the BTB and obtaining the execution status of branches.

4.1.1. Identify Characteristics of Attack Behaviors

In this article, we choose the number of BTB_Update within the certain clock cycle as the basis for identifying characteristics of attack behaviors. Figure 3 shows the BTB update process of the E906 processor. When the program starts running, IFU extracts the instruction and determines the instruction type based on the opcode. As shown in Figure 3, the binary code of a instruction is 0000 0001 1101 0111 1000 1001 0110 0011. Its opcode is 1100011 and its 12th to 14th are 000, so it a BEQ type conditional branch instruction.



 doi:10.20944/preprints202404.1017.v1

Figure 3. The update process of BTB.

Equation (1) illustrates how the Jump_Address of a branch instruction is calculated.

$$\text{Jump\_Address} = \begin{cases} \text{Branch\_Address} + (\text{Imm\_Num} << 1), \text{meets conditions} \\ \text{Branch\_Address} + 0\text{x}4, \text{not meets conditions} \end{cases} \quad (1)$$

For the branch instruction in Figure 3, its jump condition is to determine whether the values of the General Purpose Registers (GPR) X_{15} and X_{29} are equal. If the condition is met, the Jump_Address of this instruction is 0x 6b4, which is Branch_Address (0x 6a2) plus 0x 12 (Imm_Num 0000 0000 1001 shift left 1 bit and expand to 32 bits), and it is Next_PC (the address of next instruction). If the condition is not met, this instruction will not be executed, and Next_PC is 0x 6a6, which is Branch_Address plus 0x4. Branch_Address is the address of branch instruction, which can be obtained by the Program Counter (PC). IDU will determine whether the predicted Jump_Address is correct based on the actual execution of the program. If the prediction is incorrect, the BTB will be updated (the number of BTB_Update increases by 1), which means writing the jump information to a new BTB entry or replacing information from existing the BTB entry.

Most branch prediction attacks start by populating the BPU, then triggering a conditional branch of the normal process, and finally analyzing the side channel information. Take the BPA attack as an example [21]. Attackers first need to construct a spy process, which contains a large number of branch instructions to fill the BTB. Afterwards, the embedded system executes the spy process and the BTB is populated with a large number of attacker-constructed jump addresses. After that, when the embedded system runs the normal process, the branch prediction will fail because the information inside the BTB is mostly filled in by the spy process at this time, and the BTB will update the branch instruction information of the normal process. When the embedded system runs the spy process again, the update of the information in the BTB causes a difference in the execution time of some branch instructions. By analyzing the time-side channel information, attackers can know whether the branches of normal processes have been executed or not.

Usually, normal processes do not execute a large number of branch instructions. Therefore, some researchers identify spy processes based on the occupancy rate of BTB [21]. This is very suitable for BTBs with large capacities (e.g., 4096 entries). This is because the average occupancy of the BTB by normal processes is much lower than the average occupancy of the BTB by spy processes (which is usually above 90%). However, the E906 is a lightweight processor with only 16 BTB entries, which means that the BTB occupancy of a normal process may also be high. In addition, the E906 is a sequential single-threaded processor, and compared with the multi-threaded processor, the BTB of the E906 does not record the process, which means that it is impossible to determine whether the current BTB information belongs to a normal process or a spy process. Therefore, choosing occupancy rate of BTB as the attack behavior characteristic is not feasible for this article.

While complex normal processes can also fill the BTB of E906, they will not be written to the BTB at a high frequency in a short period of time. However, the spy process will concentrate on executing a large number of branch instructions to replace the original information in the BTB. Therefore, in this article, the number of BTB_Update within the certain clock cycle is chosen as the basis for the identification of spy processes. If this number exceeds the set threshold, it indicates that the current processor is densely executing branch instructions, and there may be a spy process present.

The key to calculating the number of BTB_Update is to determine the length of time. Based on the standard five level pipeline, assuming that each instruction is a branch instruction and does not hit within the BTB of E906 processer, it means that it needs to be executed until the last pipeline level to obtain the jump address of the next instruction. BTB has 16 entries, and fully filling it requires 16 branch instructions, which means it requires 80 clock cycles. Therefore, this article counts the number of BTB_Update every 80 clock cycles, rather than calculating the proportion of branch instructions in all program instructions or the occupancy rate of BTB, as other protection mechanisms do.

Table 1 describes the characteristics of the selected benchmarks. The benchmarks of hello_world and coremark are in the folder of the E906 project (opene906-main\smart_run\tests\cases) [34]. In

 doi:10.20944/preprints202404.1017.v1

order to compare with the previous works of our team [10,13,14], we select other eight benchmarks from Mibench as the test programs. These benchmarks are real-life embedded applications of various scales.

Table 1. Characteristics of the selected benchmarks.

Benchmarks	Proportion of Branches in All Instructions	Occupancy Rate of BTB	Max Number of BTB_Update
hello_world	12.54%	37.5%	6
coremark	15.41%	100%	8
basicmath	12.52%	37.5%	2
SHA1	13.07%	37.5%	3
FFT	13.60%	62.5%	5
bitcount	13.15%	50%	3
CRC16	13.23%	37.5%	3
patricia	13.04%	87.5%	7
quicksort	13.22%	62.5%	5
blowfish	13.21%	25%	3

According to Table 1, the proportion of branches in all instructions does not exceed 30%. For example, by analyzing the disassembly texts of benchmarks, it is known that hello_world has 7553 instructions and 947 branches. Therefore, its proportion of branch instructions in all program instructions is 947 divided by 7553 and multiplied by 100%, which is about 12.54%. However, the data obtained through static analysis cannot truly reflect the proportion of branch instructions when the embedded system executes programs. The spy process constructed by the attacker may only contain dozens of branches, which can completely fill the BTB of E906 processer without increasing the proportion of branch instructions a lot. Therefore, choosing the proportion of branches in all instructions as the attack feature recognition is not appropriate for lightweight embedded processors.

Some researchers chose the occupancy rate of BTB as the BPA feature recognition [21]. The processor they selected had 4096 BTB entries. When this processor executed test programs from Mibench, the highest occupancy rate of BTB did not exceed 35%. However, for the lightweight embedded processor with fewer BTB entries (such as E906), even with a normal program, its occupancy rate of BTB will be high. Therefore, for lightweight processors, it is difficult to distinguish between normal processes and spy processes based on the occupancy rate of BTB.

The occupancy rate of BTB shows how many entries of BTB are written to jump information during the entire program execution process. However, attackers usually inject spy processes in a short period of time and try to fill the BTB as much as possible. Therefore, this article chooses the number of BTB_Update within 80 clock cycles to identify the tampering attacks against BTB. If the number of BTB_Update is higher than the expected threshold, it indicates the existence of a spying process and requires protective measures such as locking the BTB. The threshold setting needs to be adjusted according to the normal process of the application. If the threshold is too small, a large number of processes will be identified as spy processes, which will have an impact on BTB access and reduce performance. If the threshold is too high, it will not be able to identify some spy processes, affecting BTB security.

As shown in Table 1, the max number of BTB_Update every 80 clock cycles is 8. Due to the spy processes always attempting to fill all BTB entries, the initial lock threshold selected in this article is 12 to prevent missed detection attacks.

4.1.2. BTB Based on Hardware Locking

Figure 4 shows how the BTB based on hardware locking prevents attackers from tampering with the jump information and obtaining the time side channel information of the corresponding branch.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 April 2024 doi:10.20944/preprints202404.1017.v1

Without any protections, the spy process first fills all entries of BTB. When the embedded system executes the normal process, it replaces the BTB entries with sensitive branches. When the spy process runs again, it will quickly execute a jump when the branch that are not involved in the normal process, as there is no need to update the BTB. However, if a entry needs to be updated, due to the need for pipeline emptying and other operations, the execution time of this branch will be much longer than the former. Therefore, attackers can tamper with BTB through the spy process, analyze the time differences and obtain the branches execution status of the normal process.The modified BTB entry has added a 1-bit Lock field, as shown in Figure 4. When the number of BTB_Update within 80 clock cycles exceeds the locking threshold ($Threshold_{Lock}$), it is considered that there is a spy process, and the Lock field will be set to 1. For a period of time thereafter, no process can write data to BTB anymore. Therefore, attackers will not be able to continue filling or replacing the remaining BTB entries through the spy process.

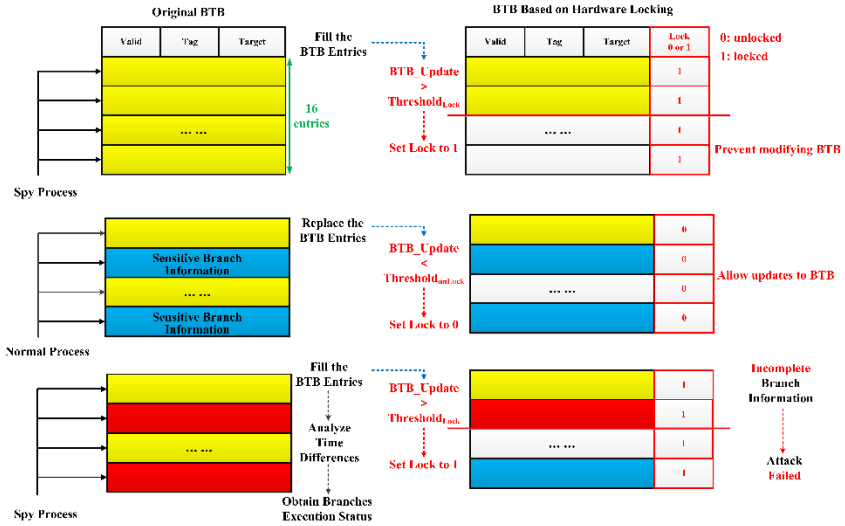


Figure 4. The BTB Based on Hardware Locking.

When the normal process executes, the update count of BTB will be lower than the unlocking threshold ($Threshold_{unLock}$), because the normal process will not execute a large number of branches in a short period of time. At this point, the Lock field is set to 0 to ensure the normal update of BTB.

If the spy process is executed again, BTB will also be locked again to prevent attackers from completely tampering with sensitive branch information within BTB. Due to the incomplete branch information, it is difficult to obtain the branch execution status of normal processes by analyzing time differences, resulting in the failure of attacks.

This method can prevent attackers from tampering with data within BTB through the spy process. However, prolonged locking will affect the operation of normal processes. Therefore, it is necessary to design $Threshold_{unLock}$.

To prevent the filling frequency of spy processes from being too high, $Threshold_{unLock}$ is lower than $Threshold_{Lock}$, which can protect the data in BTB from being tampered with and will not affect the execution of normal processes. The initial $Threshold_{unLock}$ set in this article is the half of $Threshold_{Lock}$, which is 6. In practical applications, $Threshold_{Lock}$ and $Threshold_{unLock}$ should be adjusted according to the characteristics of branches. Compared with software protection methods, this method has a faster response speed. Furthermore, compared with other hardware protection methods, this method only add the 1-bit Lock field, with minimal modification to the BTB and low hardware overhead.

4.2. The Branch Information Protection Mechanism Based on Dynamic Isolation

The protection mechanism based on hardware locking can prevent attackers from tampering with the internal data of BTB through spy processes, thereby obtaining critical branch information of normal processes. However, the jump direction and jump address of branches are often stored in

 doi:10.20944/preprints202404.1017.v1

plaintext in BHT and BTB, which attackers can obtain through other means. Therefore, there is still a certain risk of information leakage in BPU.

Most existing research adopts methods such as logical isolation or physical isolation to protect the key information within BPUs [16]. Logical isolation is to prevent other units from sharing branch predictor data, and a common implementation is to refresh the branch predictor when process switching or permission changes occur. However, refresh based logical isolation schemes often have a significant impact on performance and cannot resist side channel attacks based on competition. Physical isolation is the allocation of separate branch predictor tables for different threads and privilege levels, which can defend against content reuse based attacks and competition based side channel attacks, but the hardware resource overhead is too high. The PUF based dynamic isolation mechanism for branch information proposed in this article can generate proprietary random numbers through the designed hybrid PUF circuit, which can be used as keys to encrypt the key branch information and their index, preventing obtain and analysis by attackers, and achieving dynamic isolation of jump directions and jump addresses.

4.2.1. BTB and BHT Based on Dynamic Isolation

Figure 5 shows the dynamic isolation mechanism proposed in this paper. When the program is running, this mechanism concatenates the higher 16 bits of the Branch_Address with the output of the branch history register (BHR_Out), and generates a digest through the LHash module as the index of the corresponding 2-bit counter. The designed confidentiality protection module will generate two keys, one for encrypting the branch history information stored in BHR, and the other for encrypting the digest used as the index. In this way, attackers will find it difficult to access BHT and obtain the jump directions of critical branches.

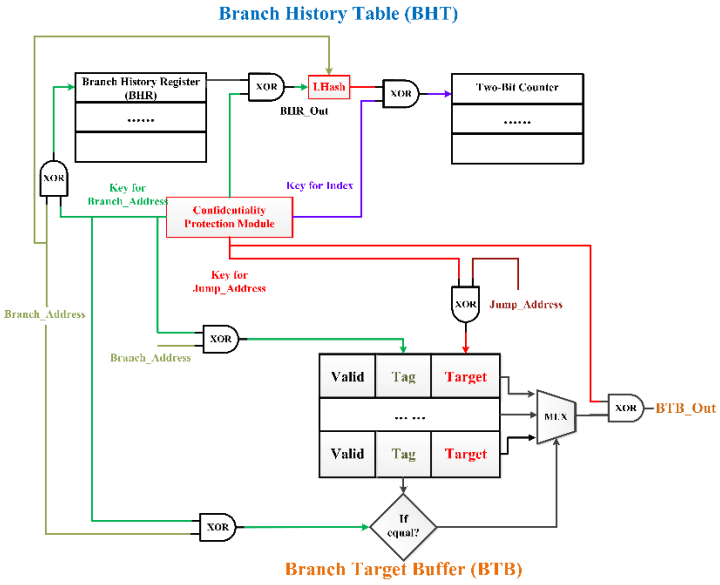


Figure 5. The principle of dynamic isolation.

LHash is a hash function which can be used to generate index values for data [33]. Compared to other hash functions, LHash can provide the required security with minimal resource overhead, making it suitable for lightweight embedded processors [19]. Considering security and area consumption, the LHash module designed in this article uses the 128 bits internal permutation structure and outputs the 96 bits binary number. Due to the BHT size of the E906 processor being 512 (2^8) × 16 and having eight 2-bit counters, the length of the index is 8 bits, which randomly selects 8 bits from the 96-bit output of LHash as the index for the 2-bit counter. Therefore, the length of the Key for index is 8 bits. Due to BHR storing the lower 16 bits of branch addresses, the length of the Key for Branch_Address is 16 bits.

When it is necessary to update the internal data of BTB, the designed confidentiality protection module will generate two keys, one for encrypting the lower 16 bits of Jump_Address and the other

 doi:10.20944/preprints202404.1017.v1

for encrypting the lower 16 bits of Branch_Address. Therefore, the length of these two keys are both 16 bits. The encrypted Jump_Address will be stored in the Target field of BTB, and the encrypted Branch_Address will be stored in the Tag field of BTB, which is used as the index for BTB entry. When it is necessary to predict the jump address of a branch, this module will generate two keys for that branch again. The Key for Branch_Address will XOR with Branch_Address and compare the calculated value with the TAG of each entry in BTB. Only when the value of a certain entry match successfully, BTB will output the Target of this entry. The plaintext of BTB_Out can only be obtained after Target is XOR with the Key for Jump_Address. In this way, attackers will find it difficult to access BTB entries and obtain the branch information.

Our team has proposed various security algorithms and their hardware circuits which can be used for information encryption [11,12]. However, although these algorithms have extremely high security and can defend against quantum attacks, they have high resource overhead, long encryption and decryption times, and are not suitable for branching information. Our team has also designed the hardware accelerator based on traditional cryptographic algorithms such as AES-GCM [25]. However, this module is more suitable for confidentiality protection applications with large amounts of data and has high resource overhead. Therefore, it is not suitable for branching information as well. Taking into account factors such as security, resources, and performance, the confidentiality protection module of this article is based on the designed hybrid PUF circuit and efficiently generates multiple keys required for the dynamic isolation mechanism.

Due to the different keys for different branches, attackers are unable to restore correct branch jump information, and therefore cannot correctly perceive whether there is competition with other processes, nor can they reuse previous historical information, fundamentally resisting competition based and reuse based attacks. The proposed dynamic isolation mechanism does not change the original control logic for updates and queries, only encrypt the jump information and their index with low overhead of resources and performance. This mechanism minimizes changes to the existing branch predictor architecture and only requires a small amount of logic to achieve dynamic isolation.

4.2.2. The Confidentiality Protection Module

Due to constraints such as hardware resources, cost, and computing power, traditional encryption techniques are difficult to widely apply in lightweight embedded systems. However, Physical Unclonable Function (PUF), as a physical security primitive, utilizes the uncontrollable random process deviations introduced during chip manufacturing to generate unique secret keys [31]. As shown in Figure 6, a set of n-bit Challenge (C_1, C_2, ……, C_n) is applied to the PUF circuit, corresponding to the generation of m-bit Response, which is called challenge response pair (CRP). According to the different abilities of PUF to generate CRPs, they are divided into weak PUF and strong PUF [26]. Strong PUF can generate an exponential number of CRPs, suitable for low-cost encryption and decryption, mainly including APUF based on arbitrators, RO-PUF based on ring oscillators, and current mirror PUF [28]. RO-PUF and current mirror PUF are susceptible to environmental noise interference and have high hardware resource costs, making them unsuitable for lightweight encryption and decryption applications. Therefore, this article mainly chooses the PUF circuit based on APUF.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 April 2024 doi:10.20944/preprints202404.1017.v1

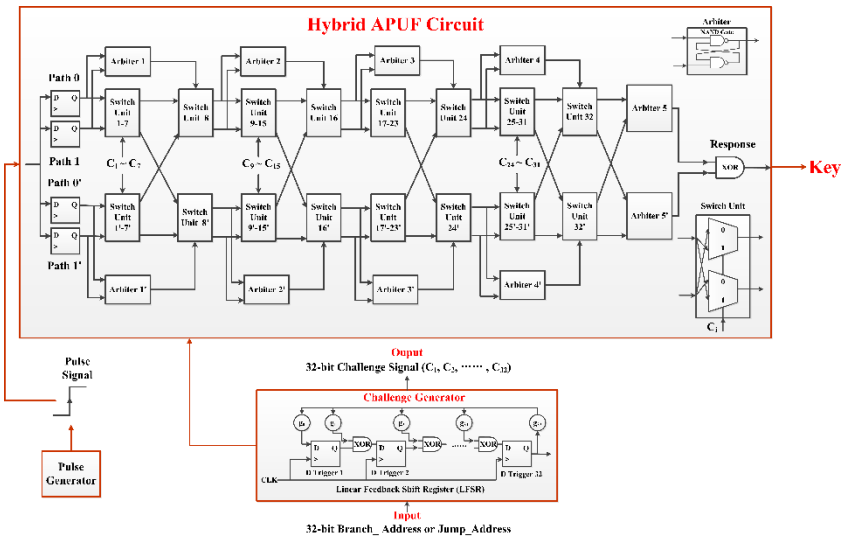


Figure 6. The circuit of hybrid APUF.

The traditional APUF circuit consists of a signal delay path cascaded with n switch units and an arbiter [27]. As shown in Figure 6, a switch unit consists of two parallel multiplexers (MUX). When the challenge signal Ci is 0, the signal path of this switch unit is the parallel path, and when Ci is 1, the signal path is the cross path. Therefore, the transmission path of the signal can be changed by controlling the challenge signal C, which affects the signal delay difference and generates the unpredictable response. The arbiter is a NAND latch. When the pulse signal is applied to the input of the APUF circuit and after being transmitted through two symmetrical delay paths (Path 0 and Path 1), the arbiter arbitrates the output signals of Path 0 and Path 1. If the signal of Path 0 arrives at the arbiter before the signal of Path 1, the output of this PUF circuit is 1, otherwise it is 0.

The total delay deviation of the signal reaching the arbitrator is the accumulation of delay deviations when it passes through each switch unit, as shown in Equation (2).

$$\Delta = (\prod_{i=1}^{n}(1-2c_i)) \bullet (\omega_i)^T \tag{2}$$

c_i is a sub element of the Challenge vector C, and ω_i is a constant vector containing delay parameters for each switch unit, as shown in Equation (3).

$$\begin{aligned} \omega_1 &= \alpha_1 \\ \omega_i &= \alpha_i + \beta_{(i-1)}, 2 \le i \le n \\ \omega_{n+1} &= \beta_n \end{aligned} \tag{3}$$

The calculation formula of α_i and β_i are shown in Equation (4).

$$\begin{aligned} \alpha_i &= \frac{x_i - y_i - u_i + v_i}{2} \\ \beta_i &= \frac{x_i - y_i + u_i - v_i}{2} \end{aligned} \tag{4}$$

x_i represents the delay parameter of the signal passing parallel through the delay path Path 0 when the challenge signal C_i is 0, and y_i represents the delay parameter of the signal passing parallel through the delay path Path 1 when C_i is 0. u_i represents the delay parameter of the signal crossing through the delay path (from Path 0 to Path 1) when C_i is 1, and v_i represents the delay parameter of the signal crossing through the delay path (from Path 1 to Path 0) when C_i is 1.

$$R = \frac{\mathrm{sgn}(\Delta) + 1}{2} \tag{5}$$

 doi:10.20944/preprints202404.1017.v1

The relationship between the response R of the APUF circuit and the total delay deviation Δ is shown in Equation (5). The value range of the sgn function is {−1, 1}. When Δ is greater than 0, R is 1, otherwise R is 0.

Due to the inherent correlation between the output response and input challenge of APUF, it is susceptible to machine learning (ML) algorithm attacks [20]. Attackers can collect a certain number of CRPs and use ML algorithms to build the mathematical model to predict response for arbitrary challenge with a high accuracy. At present, researchers mainly resist machine learning attacks by modifying PUF circuits to enhance the nonlinearity of circuit models [29]. Some researchers have proposed Feed-Forward APUF (FF-APUF), which introduces a feed-forward loop in APUF and uses the decision results of the intermediate stage as the excitation for the subsequent stage switch unit [30]. Some researchers have proposed the double APUF (DAPUF) circuit, which configures the signals sent to the arbiter for judgment according to the principle of cross exchange, enhancing the ability of PUF to resist ML attacks [38]. However, other researchers have pointed out that these structures have a success rate of over 95% predicted under targeted modeling attacks [20]. Therefore, this article combines the advantages of FF-APUF and DAPUF circuits and proposes a hybrid APUF circuit to deepen the nonlinearity of the circuit structure and improve the ability to resist ML attacks.

The proposed hybrid PUF circuit is shown in Figure 6. Add a D trigger at the beginning of each delay path (Path 0, Path 1, Path 0’ and Path 1’) to optimize the routing delay of the PUF circuit. These D triggers are driven by the same clock. Only when the rising edge of the clock arrives, the trigger will output the signal, ensuring that the time for the pulse signal input to each delay path is consistent.

In order to improve randomness, the 32-bit linear feedback shift register (LFSR) is designed as the challenge generator. This circuit expands the input 32-bit Branch_Address or Jump_Address to the maximum of 2^{32}-1 pseudo-random 32-bit challenge signals, which will loop between 2^{32}-1 different states. For the proposed hybrid APUF circuit, each 32-bit challenge signal can generate a 1-bit response. Therefore, to generate the 16-bit Key for Branch_Address, the initial input signal is the 32-bit Branch_Address, and then sixteen challenge signals are randomly selected from the output sequence of the LFSR. After 16 clock cycles, the required key can be generated. Similarly, the initial input signal is the 32-bit Jump_ Address, and after 16 clock cycles, the 16-bit Key for Jump_Address can be obtained. As for the 8-bit Key for Index, the initial input signal of the challenge generator is the 32-bit Branch_Address, and the eight challenge signals are randomly selected from the pseudorandom sequence of the LFSR.

Due to the 32-bit challenge signals, the designed hybrid PUF circuit has 32 switch units. In order to disrupt the mapping relationship between challenges and responses and reduce the accuracy of key inference based on ML attacks, a mirrored APUF with the same structure as the original APUF is designed, ensuring that the original APUF is parallel and symmetrical with the mirrored APUF. The output of the original APUF and the mirror APUF will be obfuscated by XOR to obtain the 1-bit output response. Due to the comparison signals of the Arbiter 5 and Arbiter 5’ both come from the same type of delay paths, the asymmetry of the APUF signal delay path will be effectively compensated in the hardware implementation, which can improve the ability of the PUF circuit to resist ML attacks.

To expand the selection range of signal delay paths, enhance the randomness of the PUF circuit, and achieve maximum utilization of switch units, the lower delay path and upper delay path of the symmetrically set switch units output are crossed with each other, achieving mutual inversion of input excitation. For example, Path 1 is no longer the parallel path from Switch Unit 7 to Switch Unit 8, but crosses to Switch Unit 8'. Similarly, Path 0' will move from Switch Unit 7' to Switch Unit 8. Unlike traditional APUF circuits, the pulse signals of hybrid PUF circuits can be transmitted through the cross path on the mirror symmetric APUF delay path, which can effectively improve the randomness of the circuit output response and resist ML attacks.

The feedforward loop of the PUF circuit is shown in Figure 6, and its quantity has a significant impact on the stability and ML attack resistance of the PUF circuit. The relationship between the number of feedforward circuits Num_{FF} (only consider the original APUF) and the number of interval switch units Num_{SU} is shown in Equation (6).

 doi:10.20944/preprints202404.1017.v1

$$\begin{aligned} Num_{FF} &= \frac{n}{Num_{SU}+1} \\ Num_{SU} &= 2^j - 1, j = 1,2,3,\ldots\ldots,\log_2(n)-1 \end{aligned} \tag{6}$$

n is the total number of switch units. The hybrid PUF circuit designed in this article has a total of 32 switch units, so the selectable range of the Num_{SU} is {1, 3, 7, 15} and the selectable range of the Num_{FF} is {2, 4, 8, 16}.

Table 2 illustrates the variation of the indicators of hybrid PUF circuits with the number of feedforward loops. Stability is measured by the average intra-Hamming distance (HD), and the calculation formula is shown in Equation (7).

Table 2. The variation of PUF circuit indicators with the number of feedforward loops.

Num_{FF}	Stability	Accuracy [1]	Resource [2]
2	95.1%	58.8%	209
4	98.8%	53.3%	217
8	92.5%	60.1%	233
16	82.3%	64.9%	265

[1] Based on the CMA-ES [37], which is one of ML algorithms, predict whether the output response of the proposed hybrid PUF circuit is correct. The number of CRPs used for training is 100K. [2] Based on the Xilinx Kintex 7 FPGA, implement the proposed hybrid PUF circuit. The basic hardware resources are look-up tables (LUTs).

$$\begin{aligned} Stability &= 1 - \frac{1}{T}\sum_{t=1}^{T}\frac{HD(R,R_t)}{L}\times 100\% \\ HD(R,R_t) &= \sum_{k=0}^{L-1} R[k] \oplus R_t[k] \end{aligned} \tag{7}$$

R represents the output response of the hybrid PUF circuit at 30 degrees centigrade and standard voltage, and Rt represents the output response of the t-th time under different environmental conditions. T represents the total number of environmental conditions, which is the total number of times the circuit response is measured. L is the length of the output response. HD(R, R_t) is the Hamming distance between the response R and R_t. This article will implement the designed hybrid PUF circuit on FPGA, and test its stability in the temperature range of 25 to 75 degrees centigrade with a step size of 10 degrees centigrade. Each experiment will be repeated 10 times, and the average stability under multiple environmental conditions will be calculated. Based on the test results in Table 2, taking into account the stability, resistance to ML attacks, and hardware resource overhead, Num_{FF} and Num_{SU} selected in this article are 4 and 7, and the structure of the hybrid PUF circuit is shown in Figure 6.

4.3. The Control Flow Integrity Protection Mechanism Based on Branch Labels

If the protection mechanism proposed above fails, attackers can manipulate the jump directions or addresses of branches and hijack the control flow of the program through CIAs [35], CRAs [36], etc. Although researchers have proposed various fine-grained control flow integrity (CFI) monitoring methods [10,13,14,19,32] that can detect tampering attacks on instruction addresses, instruction code, and jump addresses in a timely manner, these methods cannot determine whether the jump direction of branches during program execution has been tampered with. This article proposes the control flow integrity monitoring mechanism based on branch labels, which integrates the jump conditions of branches into the label calculation process. By comparing the dynamically calculated label values during program execution to ensure they meet design expectations, it accurately determines whether the jump directions are correct.

4.3.1. Extract the Reference Information

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 April 2024 doi:10.20944/preprints202404.1017.v1

When the program is offline, extract the reference information required for controller integrity monitoring, as shown in Figure 7. Firstly, compile and link the source code to generate the binary code of the program. Secondly, disassemble the binary code to obtain the disassembly text. Afterwards, perform static analysis on the text and divide instructions into several basic blocks (BBs). BB is a collection of sequentially executed instructions, and only contain one jump instruction. For the RISC-V instruction set, jump instructions can be divided into unconditional jumps and conditional branches. The unconditional jump only have one jump address, while the conditional branch have two. When the program is running, if the jump condition of the branch is met, the jump address is its branch address plus the offset. If it is not met, the jump address is its branch address plus 0x 4, which means the next instruction is executed in sequence.

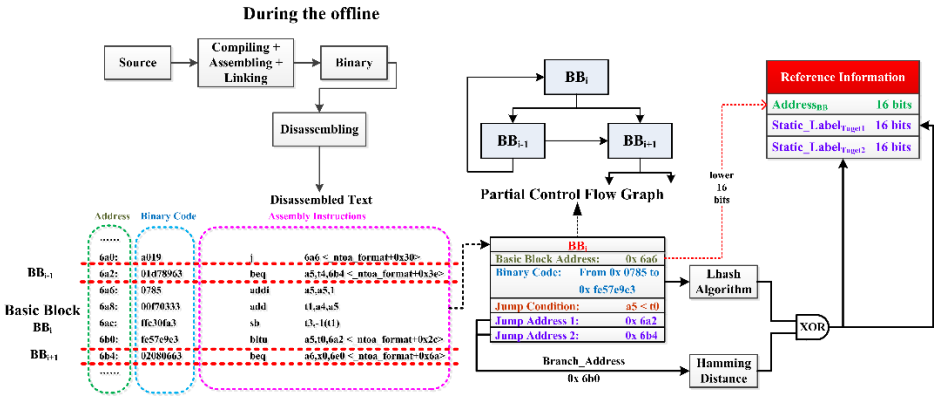


Figure 7. The extraction process of the reference information.

As shown in Figure 7, according to the basic block partitioning rules, part of the program code can be divided into three basic blocks, which are BB_{i-1}, BB_i and BB_{i+1}. BB_{i-1} and BB_{i+1} only have one instruction, which is a BEQ type instruction (its Branch_Address is 0x 6b0). BB_i has four instructions and the last is a BLTU type instruction. By conducting static analysis on the BBs, the control flow graph (CFG) can be obtained to understand the possible execution paths of these BBs. Taking BB_i as the example, its basic block address is 0x 6a6, which is the address of the first instruction in BB_i. The jump condition of BB_i is to determine whether the unsigned number of register a5 is less than the unsigned number of register t0. If the jump condition is met, the jump address of BB_i is 0x 6a2, which is the address of BB_{i-1}. Otherwise, the jump address is 0x 6b4, which is the address of BB_{i+1}. Similarly, static analysis can be performed on BB_{i-1} and BB_{i+1}, and based on the jump relationship, these BBs can be linked together to form the partial CFG.

Further process the basic block information obtained from static analysis to obtain the reference information required for CFI monitoring. Take the lower 16 bits of the basic block address as $Address_{BB}$, which can be used for indexing the reference information of this BB in the security monitoring module (SMU). Using the LHash algorithm [33], map the binary codes of all instructions within the BB to a 16-bit digest. If attackers tamper with any instruction code of this BB, the corresponding digest will also change. Therefore, the digest represent the integrity of instruction code within BBs. Calculate the Hamming distance (HD) between the jump address and the branch instruction address (Branch_Address) of the BB, and XOR the result with the digest to obtain 16-bit $Static\_Lable_{Target1}$ and $Static\_Lable_{Target2}$. If attackers hijack the control flow and alter the execution order of instructions, the calculated HD will also change. Therefore, HD can represent the CFI of BBs. If the only jump instruction within the BB is not a conditional branch, but an unconditional jump, then the reference information for that BB only has valid $Address_{BB}$ and $Static\_Lable_{Target1}$, while $Static\_Lable_{Target2}$ is null.

This article considers both instruction code integrity and control flow integrity when designing the static labels. When the program is running, the designed SMU analyzes the decoded instructions and divide these instructions into several BBs. Referring to the static analysis process, SMU calculates

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 16 April 2024 doi:10.20944/preprints202404.1017.v1

the digest and the Hamming distance, and XOR the two to obtain the Dynamic_Label of BBs. Based on the $Address_{BB}$, SMU can find the corresponding static labels. For the BBs containing unconditional jumps, check whether Dynamic_Labels are equal to $Static\_Lable_{Target1}$. For the BBs containing conditional branches, check whether Dynamic_Label is equal to $Static\_Lable_{Target1}$ or $Static\_Lable_{Target2}$. If the dynamic and static labels are not equal, it is considered that instruction codes or program control flow have be tampered with.

4.3.2. Monitor the Control Flow Integrity

Although SMU can detect tampering with instruction code or control flow in a timely manner by comparing dynamic and static labels, it is sometimes difficult to determine whether the jump directions of conditional branches have been tampered with. This is because static analysis is difficult to accurately determine whether the jump condition of the conditional branch at a certain moment is met. As shown in Figure 7, the execution path from BB_{i-1} to BB_i is a loop. When the value of the general purpose registers (GPRs) meets the jump condition of BB_{i-1} or does not meet the jump condition of BB_i, the embedded system will execute BB_{i+1}. It is difficult to determine when the jump conditions of these BBs are met or not through static analysis alone. Although it is possible to record the execution path of BBs and the execution status of conditional branches by software simulators, the required amount of data is too large and not suitable for CFI monitoring in lightweight embedded processors. Therefore, the proposed CFI protection mechanism will incorporate the jump conditions into Dynamic_Label calculation when monitoring BBs containing branches. Figure 8 shows the CFI monitoring process for branches.

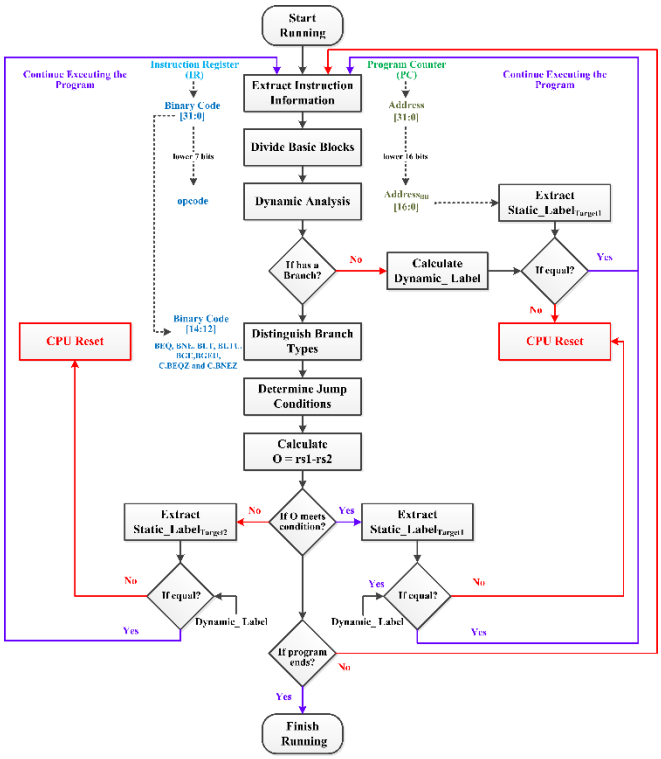


Figure 8. The CFI monitoring process for branches.

When the embedded system starts running the program, SMU extracts instruction information from the e906 processor pipeline, including 32-bit binary code from the instruction register (IR) and 32-bit instruction address from the program counter (PC). Based on the opcode, the instruction type can be identified and these instructions can be divided into several BBs. Through dynamic analysis, the information of each BB is extracted, including the address of the BB ($Address_{BB}$), the types of jump instructions included, the jump address, and the digest of all internal instruction codes.

 doi:10.20944/preprints202404.1017.v1

If the jump instruction within the BB is not a conditional branch, but an unconditional jump, the Dynamic_Label of this BB is first calculated. Meanwhile, based on the address, retrieve the $Static\_Label_{Target1}$ of the corresponding BB in the integrity parameter memory (IPM). As shown in Figure 2, when the program code is loaded into the Flash or DDR of the embedded processor, the previously extracted reference information will also be securely loaded and stored in IPM. Due to the unconditional jump having only one jump address, if the dynamic and static labels are different, it is considered that the integrity of the program instruction code or control flow has been tampered with. The processor reset operation is initiated and the program is re-executed. If the labels are consistent, continue executing the instructions until the program ends.

If the jump instruction within the BB is a conditional branch, determine the branch type based on the binary code [14:12]. Different types of branches have different jump conditions. The condition for the BEQ type branch is whether rs1 and rs2 are equal. The condition for the BNE type branch is whether rs1 and rs2 are not equal. The condition for the BLT and BLTU type branch are whether the value of register rs1 is less than the value of register rs2. The BLT type branch compares signed numbers, while the BLTU type branch compares unsigned numbers. The condition for the BGE and BGEU type branch are whether the value of rs1 is is greater than or equal to the value of rs2. The C.BEQZ type branch is a compression instruction with the length of only 16 bits, and its jump condition is whether rs1 is equal to zero. The condition for the C.BNEZ type branch is whether rs1 is not equal to zero. To dynamically determine whether the jump condition at a certain time is met, SMU calculates the difference (O) between the internal data of rs1 and rs2.

As for the the BNE type branch, If O is 0 at this time, it indicates that the jump condition is met, and the jump address is the current PC value plus the offset. At this time, it is necessary to compare whether the Dynamic_Label is consistent with $Static\_Label_{Target1}$. If O is not 0, it indicates that the jump condition is not met at this time. The Dynamic_Label and $Static\_Label_{Target2}$ should be compared.

5. Experiments and Results

This article implements the proposed hardware protection method for conditional branches on the FPGA platform. The selected FPGA chip is Xilinx Kinex-5, XC7K325T-2FFG900I, and the development board is ALINX AX7325. We have implemented an embedded system on-chip (SoC) on the FPGA based on an open-source RISC-V processor (Xuantie E906 [34]). The clock frequency of SoC is 100 MHz. As shown in Table 1, we have selected ten sets of commonly used programs or algorithms in the industrial control field as benchmarks in the open-source project of E906 [34] and Mibench [10].

5.1. Security Analysis

There are currently many attacks targeting branch information, such as side channel attacks [21], critical data theft [16], and control flow tampering [19]. Therefore, this article proposes three hardware protection mechanisms for branch information to prevent attackers from completely tampering with BTB, dynamically isolating critical data of BHT and BTB, and monitoring the control flow integrity during program execution. The following text will analyze the security of the proposed mechanism separately.

5.1.1. Security of the Hardware Locking Protection Mechanism

Branch Prediction Analysis (BPA) [21] is a side channel attack against BTB. The attack process is shown in Figure 4. If attackers can accurately predict whether a branch of benchmarks will jump, then this BPA attack is considered successful. Assuming that the test program executes p branches during runtime, and attackers successfully predict whether q branches will jump, the attack success rate is $q \div p \times 100\%$.

Table 3 shows the impact of with or without the hardware locking protection mechanism on the defense against BPA. Facing the unprotected embedded system, the success rate of BPA attacks are close to 100%. This is because attackers can theoretically inject spy processes before the normal process executes branches, and completely tampering with BTB. When a branch of the normal process

 doi:10.20944/preprints202404.1017.v1

needs to be executed, due to its information being different from the spy process, BTB needs to be updated. Therefore, attackers can theoretically fully understand the branch execution status of the normal process through the difference in BTB update time. However, if there is a conditional branch in the first few instructions when the test program starts running, this attack may fail because the spy process has not yet fully filled the BTB. Therefore, even without protection, the success rate of attacks are not 100%.

Table 3. Security testing for defending against Branch Prediction Analysis attacks.

Benchmarks	Attack Success Rate	
	Without Protection	With Hardware Locking BTB
hello_world	98.1%	58.3%
coremark	97.4%	59.6%
basicmath	97.9%	56.4%
SHA1	97.6%	57.5%
FFT	98.3%	58.1%
bitcount	97.3%	59.4%
CRC16	97.5%	58.7%
patricia	98.8%	57.9%
quicksort	98.6%	58.3%
blowfish	98.5%	55.9%

With the proposed hardware locking protection mechanism, if the number of BTB_ Update every 80 cycles caused by spy processes exceeds the $Threshold_{Lock}$, the lock field of BTB entries will be set to 1 to prevent BTB from being completely tampered with. Therefore, attackers will not be able to obtain the execution status of most branches and can only make random guesses, resulting in the decrease in attack success rate to nearly 50%. This indicates that the proposed branch information protection mechanism based on hardware locking can effectively resist side channel attacks such as BPA.

5.1.2. Security of the Proposed Hybrid APUF

The proposed branch information protection mechanism based on dynamic isolation uses the designed confidentiality protection module to encrypt and decrypt branch addresses, jump addresses, and their indexes. Therefore, the security of this mechanism mainly depends on the cryptographic circuit used, which is the hybrid APUF.

Table 4 shows the security comparison of the multiple APUF circuits. Since the different researchers proposed APUF circuits used different size of challenge, even if the type is the same, the structures of APUF circuits are different and there are differences in safety. Therefore, this article refers to previous research results [20,27,29,37] and designs the traditional APUF, 2XOR-APUF, DAPUF, and FF-APUF. The challenges of these circuits is consistent with the proposed hybrid APUF, which are 32 bits.

$$\text{Randomness} = \frac{1}{n}\sum_{e=1}^{n} R_e \times 100\% \tag{8}$$

$$\text{Uniqueness} = \frac{2}{k(k-1)}\sum_{f=1}^{k-1}\sum_{g=i+1}^{k}\frac{HD(R_f, R_g)}{n} \times 100\% \tag{9}$$

Table 4. The security comparison of the multiple APUF circuits.

APUF Types	Stability	Randomness	Uniqueness	Accuracy		
				CMA-ES	LR	ANN

 doi:10.20944/preprints202404.1017.v1

Traditional APUF	98.85%	51.31%	49.57%	98.28%	99.31%	99.27%
2XOR-APUF	98.83%	48.89%	49.65%	95.18%	97.32%	96.98%
DAPUF	98.81%	49.63%	50.11%	64.84%	65.65%	75.16%
FF-APUF	97.84	50.58%	48.91%	86.13%	87.65%	88.01%
Hybrid APUF	98.79%	49.85%	50.06%	53.32%	56.78%	57.93%

Stability is a measure of whether the APUF circuit can ensure the same response from the same challenge in different environments. The calculation of stability is shown in Equation (7). Ideally, the stability should be 100%. Randomness reflects the distribution of 0 and 1 in the response, and ideally, it should be 50%. The calculation of randomness is shown in Equation (8). Assuming the response has n bits, R_e is the response of the e-th bit. Uniqueness reflects the ability of different PUF entities to respond differently to the same incentive, ideally at 50%. The calculation of randomness is shown in Equation (9). k represents the number of APUF entities, HD represents the Hamming distance between the response R_f generated by the f-th entity and the response R_g generated by the g-th entity. This article implements APUF circuits with the same structure at five different positions inside the FPGA, which can be seen as implementing five different APUF entities.

Referring to previous researches [37], this article also selected three typical machine learning (ML) algorithms (CMA-ES, LR, and ANN) to predict the output response of APUF circuits. Covariance matrix adaptation-evolution strategy (CMA-ES) is one of the mainstream attack algorithms for PUF, which can customize fitness functions based on the structure and has high response prediction accuracy. Logistic regression (LR) is a linear regression model with simple structure and low computational complexity, which can also be used to predict the response. This article uses gradient descent algorithm as the optimizer of LR, with a learning rate of 0.01, and softmax is used to activate the LR layer. Artificial Neural Network (ANN) is a complex network composed of a large number of interconnected neurons, which can be used to fit the input and output correlation functions of APUF. The ANN designed in this article has three hidden layers (each layer has 128 nodes and is connected to a dropout layer), uses ReLU for nonlinear transformation, and uses softmax for activating the output layer. The learning rate of this ANN is 0.01, and the optimizer is momentum algorithm. The training set for the three ML algorithms consists of 100K CPRs, while the testing set consists of 10K CPRs.

As shown in Table 4, the hybrid APUF proposed in this paper has stronger resistance to machine learning attacks and better security compared to traditional APUF, 2XOR-APUF, DAPUF, and FF-APUF. The stability, randomness, and uniqueness of the hybrid APUF circuit are also close to ideal values.

5.1.3. Security of the Control Flow Integrity Protection Mechanism

In order to evaluate the security of the proposed CFI protection mechanism, this paper designs an attack circuit inside the processor. This circuit can tamper with signals and registers, and simulate attacks on conditional branches and unconditional jumps. Due to the conditional branches having two potential jump addresses, compared to unconditional jumps, attackers can not only tamper with the instruction codes or jump addresses, but also tamper with the jump directions to disrupt program control flow. For each benchmark, the attack circuit will randomly select 1000 attack points to tamper with instruction code, jump addresses, or jump directions.

Table 5 shows the detection rates of the proposed mechanism against different types of control flow tampering attacks. From the data in the Table 5, it can be seen that the proposed protection mechanism can effectively detect tampering attacks on conditional branches. This is because if an attacker tampers with the instruction codes of branches, the digest calculated by SMU will undergo significant changes compared to the digest calculated during static analysis, resulting in inconsistent dynamic and static labels. Similarly, if the jump addresses are tampered with, the HD calculated by the SMU will be different from the HD calculated during static analysis, also resulting in inconsistent dynamic and static labels. Moreover, compared to our previous research [13,19], since this article

 doi:10.20944/preprints202404.1017.v1

calculates whether the jump condition is met, SMU will choose whether to compare the Dynamic_Label with Static_Label$_{Target1}$ or Static_Label$_{Target2}$ based on the actual jump situation. Therefore, this mechanism can detect tampering with the jump directions.

Table 5. The detection rate of control flow tampering attacks.

Benchmarks	Attacks on Conditional Branches			Attacks on Unconditional Jumps	
	Instruction Codes	Jump Addresses	Jump Directions	Instruction Codes	Jump Addresses
hello_world	100%	100%	100%	100%	93.4%
coremark	100%	100%	100%	100%	92.2%
basicmath	100%	100%	100%	100%	95.7%
SHA1	100%	100%	100%	100%	96.3%
FFT	100%	100%	100%	100%	96.4%
bitcount	100%	100%	100%	100%	94.3%
CRC16	100%	100%	100%	100%	95.4%
patricia	100%	100%	100%	100%	96.8%
quicksort	100%	100%	100%	100%	96.1%
blowfish	100%	100%	100%	100%	93.6%

However, although this mechanism can effectively detect tampering with the instruction codes of unconditional jumps, the attack detection rate for the jump addresses can only reach over 90%, not 100%. This is because unconditional jumps contain indirect jump type instructions (JALR). If attackers only tamper with the values of registers involved in the JALR instructions, and the tampered jump addresses are valid, SMU will not detect these attacks.

5.2. Resource Overhead Assessment

The hardware locking BTB proposed in this article only requires the addition of the 1-bit Lock field. The BTB of the E906 processor has 16 entries, so the hardware resource overhead is 1×16 = 16bits. Adding the circuit that calculates the number of BTB_Update within 80 cycles, as well as the circuit that controls whether BTB is locked based on the number of BTB_Update, Threshold$_{Lock}$, and Threshold$_{Unlock}$, the total resource overhead of the proposed mechanism is only 110 bits, which can be almost negligible compared to existing protection methods. For example, the anti BPA protection method proposed in [21] has a resource overhead of 8KB.

Table 6 shows the hardware resource overhead of different confidentiality protection circuits. Refer to Ref. [20], Ref. [27], Ref. [29], and Ref. [37], we design the different circuits based on traditional APUF, 2XOR-APUF, DAPUF, and FF-APUF. Moreover, except for the APUF structure, these circuits has the same other circuit components as the confidentiality protection circuit based on hybrid APUF proposed in this article. Such as the pulse generation module, the challenge generation module, and so on. As shown in the data in the table, the proposed confidentiality protection circuit consumes more LUTs and Slices than traditional APUF and FF-APUF. This is because in order to enhance security, this article adds a mirrored APUF circuit, which can confuse the output response and reduce the prediction accuracy based on machine learning algorithms. Compared with the circuits based 2XOR-APUF and FF-APUF, the resources consumed by the circuit proposed in this article are basically equivalent.

Table 6. Comparison of the hardware resource overhead for confidentiality protection circuits.

Circuits	LUTs	Flip Flops	Slices
Traditional APUF	409	156	133
2XOR-APUF	672	165	223
DAPUF	687	165	228
FF-APUF	429	150	142

 doi:10.20944/preprints202404.1017.v1

Hybrid APUF	712	168	231

The proposed CFI mechanism requires additional area to store reference information. Compared to our previous work [10,13], this article combines instruction code integrity and control flow integrity when calculating the static labels of BBs, reducing the reference information of each BB to only 48 bits in size. The reference information size in Ref. [10] is the same as the program code size, while each BB in Ref. [13] has 64-bit reference information. Therefore, compared to the aforementioned studies, the method proposed in this article incurs less resource overhead caused by reference information.

The SoC based on E906 consumed 34173 LUTs, 14426 Slice Registers, and 10529 Slices without incorporating the method proposed in this article. The three proposed hardware protection mechanisms are implemented on FPGA. Including the hardware locking BTB, confidentiality protection circuit based on hybrid APUF, and security monitoring module based on branch labels designed in this article, the total resource overhead of the proposed methods are 1851 LUTs, 816 Slice Registers, and 613 Slices. Therefore, the proposed method only increased the resource cost of LUTs by approximately 5.4%.

5.3. Performance Overhead Assessment

This article chooses the cycles per instruction (CPI) to evaluate the embedded system performance overhead caused by the proposed method. Table 7 shows the CPI changes of embedded systems executing different benchmarks before and after the addition of the proposed method. It can be seen from Table 7 that for each benchmark, the performance overhead is less than 5.5%, and the average value is 4.59%. The performance overhead mainly comes from the time spent on calculating dynamic labels and retrieving static labels, when SMU monitors the CFI of programs. For the method proposed in this article, the more jump instructions the program needs to execute, the higher the performance overhead. However, for the lightweight embedded processors, in their typical application scenarios, the executed program is not very complex and does not contain too many jump instructions. Therefore, the performance overhead caused by the proposed method is acceptable for lightweight processors. Compared with existing research on control flow protection, such as references [10] (its performance overhead is about 9.33%) and [13] (its performance overhead is about 6.18%), the performance overhead of the proposed method is acceptable for lightweight embedded systems.

Table 7. The change in CPI caused by the proposed method.

Benchmarks	With Proposed Method	Without Proposed Method	Performance Overhead
hello_world	1.28	1.24	4.03%
coremark	3.38	3.21	5.30%
basicmath	2.37	2.28	3.95%
SHA1	2.07	1.98	4.55%
FFT	2.51	2.39	5.02%
bitcount	1.42	1.36	4.41%
CRC16	1.65	1.58	4.43%
patricia	1.55	1.47	5.44%
quicksort	1.83	1.76	3.98%
blowfish	3.51	3.35	4.78%

5.4. Comparison with Other Security Protection Methods

Table 8 shows the comparison of this article with other security protection methods in terms of security, resource overhead, and performance overhead. When evaluating security, this article focuses on the ability of different methods to protect branch information, such as preventing attackers

 doi:10.20944/preprints202404.1017.v1

from tampering with the BPU, protecting the confidentiality of branch information, and monitoring the CFI of the programs.

Ref. [15] proposed a software protection method (Conditional Speculation) that limited the execution of memory instructions, which could resist attacks based on Spectre vulnerabilities. However, this method cannot prevent attackers from stealing branch information or monitor the CFI. Moreover, although the average performance overhead of this method is low, for special cases, it may limit the execution of normal memory instructions, resulting in a sharp increase in performance overhead.

Table 8. The comparison of security, resource overhead, and performance overhead.

Methods	Prevent Obtaining Execution Status of Branches	Encrypt Branch Information	Monitoring the CFI	Resource Overhead	Performance Overhead
Conditional Speculation [15]	Yes	No	No	About 1.52 KB	The average is 2.8%, but in extreme cases it may approach 30%.
Secure Branch Predictor [16]	Yes	Yes	No	BTB_Area increased by 0.24%, PHT_Area increased by 0.11%.	The average is a few percent, but sometimes it may be more than 20%.
Countermeasure against BPA [21]	Yes	No	No	Area increased by 8KB	0.12%
HCIC [18]	No	Yes	Yes	Binary_Size increasd by 0.78%.	0.95%
M-Cache based Security Monitor [10]	No	No	Yes	Area increased by 20.99%.	9.33%
Hardware Monitoring Module [13]	No	No	Yes	Consumed 486 Slices and 1374 LUTs.	Less than 9.52%
The Propose Method	Yes	Yes	Yes	Consumed 613 Slices and 1851 LUTs.	Less than 5.5%

Ref. [16] proposed a secure branch predictor that encrypted the content and indexes of BTB and BHT to resist malicious perception attacks on branches. However, the security of this method mainly depends on the random number required for encryption. Attackers can successfully predict random numbers based on ML algorithms and obtain branch information predicted from this preditor.

Ref. [21] proposed a hardware method to resist BPA, which calculated whether the BTB occupancy rate exceeds the threshold and determined whether BTB updates were blocked. However, this method did not consider the possibility of normal processes being filled with BTB if the BTB entries are small. Therefore, this method is not suitable for lightweight processors like E906.

Ref. [18] proposed a hardware-assisted control flow checking method (HCIC) that used PUF to encrypt return addresses and jump addresses, which could resist code reuse attacks (CRAs). However, Ref. [18] did not discuss the countermeasures of PUF against ML attacks, which posed security risks.

 doi:10.20944/preprints202404.1017.v1

Ref. [10] and Ref. [13] both monitored the CFI based on label comparison, and promptly detected tampering with instruction codes and jump addresses. However, these methods did not consider whether the jump conditions of branches were met, and could not detect attackers tampering with the jump directions.

Compared to the above methods, the protection method proposed in this article can prevent attackers from obtaining the execution status of branches, encrypt the branch information, and detect tampering with instruction codes, jump addresses, and jump directions. For the lightweight embedded processor (E906) with only 16 BTB entries, this article calculates the number of BTB_Update per 80 clock cycles, rather than the BTB occupancy rate during the entire program execution period, to determine if there is any spy processes, and initiates hardware locking mechanisms to prevent attackers from completely tampering with the BTB. In order to protect the confidentiality of branch information, this article designs a hybrid APUF with ML attack resistance to perform lightweight encryption and decryption on content and its indexes, achieving dynamic isolation of branch information during program execution. This article integrates the calculation of branch jump conditions into the process of CFI monitoring, achieving comprehensive protection of instruction codes, jump addresses, and jump directions.

6. Conclusions

This article proposes a hardware security protection method for conditional branches of embedded systems. This method calculates the number of BTB_Update every 80 clock cycles during program execution. If the value exceeds the lock threshold, the Lock field of each BTB entry is set to 1. At this point, any process will be unable to modify the branch information within BTB, making it difficult for attackers to obtain the execution status of program branches through attacks such as BPA. Moreover, to protect the confidentiality of branch information within BTB and BHT, this article designs a dynamic isolation mechanism based on hybrid PUF, which performs lightweight encryption and decryption on the indexes, branch addresses, and jump addresses of branches to prevent attackers from stealing these critical data. In order to resist machine learning attacks against APUF (such as CMA-ES, LR, and ANN), this article integrates the advantages of multiple APUF circuits, ensuring the stability, randomness, and uniqueness of the circuit, while significantly reducing the accuracy of attackers in predicting the output responses. Once the above protection mechanisms are breached or bypassed, attackers can tamper with the instruction codes, jump addresses, or jump directions. The hardware SMU designed in this article will detect such attacks in a timely manner by comparing dynamic and static labels, achieving real-time monitoring of the control flow integrity. After FPGA implementation, this article evaluates the security, resource overhead, and performance overhead of the proposed method. The experimental results show that the hardware locking mechanism proposed in this paper can effectively resist BPA attacks, the designed hybrid APUF circuit has high resistance to ML attacks, and the proposed CFI monitoring mechanism can timely detect tampering attacks on instruction codes, jump addresses, and jump directions. Moreover, the resource overhead of the proposed method is about 5.4% of the SoC based on E906, and the performance overhead is less than 5.5%, making it suitable for lightweight embedded processors.

Author Contributions: Conceptualization, Q.H.; methodology, Q.H.; software, Q.H. and D.X.; validation, Q.H., Y.Q. and R.L.; formal analysis, Z.Z. and Y.Y.; investigation, Q.H. and D.X.; resources, X.W.; data curation, Y.Q. and Z.Z.; writing—original draft preparation, Q.H.; writing—review and editing, Q.H.; supervision, X.W.; project administration, X.W.; funding acquisition, X.W. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China (Grants No. 60973106 and No. 81571142), the Key Project of the National Natural Science Foundation of Chi-na (Grant No. 61232009), and the Open Foundation of Space-Trusted Computing and Electronic Information Technology Laboratory (Grant No. OBCandETL-2019-02).

Data Availability Statement: Data are contained within this article.

 doi:10.20944/preprints202404.1017.v1

Acknowledgments: We sincerely appreciate the Shaoxing Yangyu AI Chip CO., LTD, Zhejiang 312035, China, for providing financial support and technical resources. The method proposed in this article may be applied to the smart chip and its equipment of this company in the future.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Arıkan, K.; Palumbo, A.; Cassano, L.; Reviriego, P.; Pontarelli, S.; Bianchi, G.; Ergin, O.; Ottavi, M. Processor Security: Detecting Microarchitectural Attacks via Count-Min Sketches. *IEEE Trans. Large Scale Integr. Syst.* **2022**, *30*, 938-951.
2. Wang, G.; Chattopadhyay, S.; Gotovchits, I.; Mitra, T.; Roychoudhury, A. oo7: Low-Overhead Defense Against Spectre Attacks via Program Analysis. *IEEE Trans. Software Engineering* **2021**, *47*, 2504-2519.
3. Zheng, B.; Gu, J.; Wang, J.; Weng, C. CBA-Detector: A Self-Feedback Detector Against Cache-Based Attacks. *IEEE Trans. Dependable and Secure Computing* **2022**, *19*, 3231-3243.
4. Nasahl, P.; Schilling, R.; Mangard, S. Protecting Indirect Branches Against Fault Attacks Using ARM Pointer Authentication. In Proceedings of the 2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), ysons Corner, VA, USA, 12-15 December 2021; pp. 68-79.
5. Wang, Y.; Wang, Q.; Chen, X.; Chen, D.; Fang, X.; Yin, M.; Zhang, N. ContainerGuard: A Real-Time Attack Detection System in Container-Based Big Data Platform. *IEEE Trans. Industrial Informatics* **2022**, *18*, 3327-3336.
6. Bhattacharya, S.; Maurice, C.; Bhasin, S.; Mukhopadhyay, D. Branch Prediction Attack on Blinded Scalar Multiplication. *IEEE Trans. on Computers* **2020**, *69*, 633-648.
7. Islam Chowdhuryy, M. H.; Liu, H.; Yao, F. BranchSpec: Information Leakage Attacks Exploiting Speculative Branch Instruction Executions. In Proceedings of the 2020 IEEE 38th International Conference on Computer Design (ICCD), Hartford, CT, USA, 18-21 October 2020; pp. 529-536.
8. Le Bon, C.; Rohou, E.; Tronel, F.; Hiet, G. DAMAS: Control-Data Isolation at Runtime through Dynamic Binary Modification. In Proceedings of the 2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), Vienna, Austria, 06-10 September 2021; pp. 86-95.
9. Kim, S.; Mahmud, F.; Huang, J.; Majumder, P.; Tsai, C.; Muzahid, A.; Kim, E. J. WHISTLE: CPU Abstractions for Hardware and Software Memory Safety Invariants. *IEEE Trans. Computers* **2023**, *72*, 811-825.
10. Wang, X.; Zhao, Z.; Xu, D.; Zhang, Z.; Hao, Q.; Liu, M. An M-Cache based security monitoring and fault recovery architecture for embedded processor. *IEEE Trans. Large Scale Integr. Syst.* **2020**, *28*, 2314-2327.
11. Xu, D.; Wang, X.; Hao, Y.; Zhang, Z.; Hao, Q.; Zhou, Z. A More Accurate and Robust Binary Ring-LWE Decryption Scheme and Its Hardware Implementation for IoT Devices. *IEEE Trans. Large Scale Integr. Syst.* **2022**, *30*, 1007-1019.
12. Xu, D.; Wang, X.; Hao, Y.; Zhang, Z.; Hao, Q.; Jia, H.; Dong, H.; Zhang, L. Ring-ExpLWE: A High-Performance and Lightweight Post-Quantum Encryption Scheme for Resource-Constrained IoT Devices. *IEEE Internet of Things J.* **2022**, *9*, 24122 - 24134.
13. Hao, Q.; Zhang, Z.; Xu, D.; Wang, J.; Liu, J.; Zhang, J.; Ma, J.; Wang, X. A Hardware Security-Monitoring Architecture Based on Data Integrity and Control Flow Integrity for Embedded Systems. *Appl. Sci.* **2022**, *12*, 7750.
14. Wang, X.; Zhang, Z.; Hao, Q.; Xu, D.; Wang, J.; Jia, H.; Zhou, Z. Hardware-Assisted Security Monitoring Unit for Real-Time Ensuring Secure Instruction Execution and Data Processing in Embedded Systems. *Micromachines* **2021**, *12*, 1450.
15. Zhao, L.; Li, P.; Hou, R.; Huang, M. C.; Liu, P.; Zha, L.; Meng, D. Exploiting Security Dependence for Conditional Speculation Against Spectre Attacks. *IEEE Trans. Computers* **2021**, *70*, 963-978.
16. Zhao, L.; Li, P.; Hou, R.; Huang, M. C.; Li, J.; Zhang, L.; Qian, X.; Meng, D. A Lightweight Isolation Mechanism for Secure Branch Predictors. In Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 05-09 December 2021; pp. 1267-1272.
17. Koruyeh, E. M.; Haji Amin Shirazi, S.; Khasawneh, K. N.; Song, C.; Abu-Ghazaleh, N. SpecCFI: Mitigating Spectre Attacks using CFI Informed Speculation. In Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 18-21 May 2020; pp. 39-53.
18. Zhang, J.; Qi, B.; Qin, Z.; Qu, G. HCIC: Hardware-assisted Control-flow Integrity Checking. *IEEE Internet Things J.* **2018**, *6*, 458-471.
19. Hao, Q.; Xu, D.; Zhang, Z.; Wang, J.; Le, T.; Wang, J.; Zhang, J.; Liu, J.; Ma, J.; Wang, X. A Hardware-Assisted Security Monitoring Method for Jump Instruction and Jump Address in Embedded Systems. In Proceedings of the 2022 8th Annual International Conference on Network and Information Systems for Computers (ICNISC), Hangzhou, China, 16-19 September 2022; pp. 197-202.
20. Shi, J.; Lu, Y.; Zhang, J. Approximation Attacks on Strong PUFs. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems* **2019**, *39*, 2138-2151.

21. Tan, Y.; Wei, J.; Guo, W. The Micro-architectural Support Countermeasures against the Branch Prediction Analysis Attack. In Proceedings of the 2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications, Beijing, China, 24-26 September 2014; pp. 276-283.
22. Evtyushkin, D.; Riley, R.; Abu-Ghazaleh, N.; Ponomarev, D. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. *ACM SIGPLAN Notices* **2018**, *53*, 693-707.
23. Li, P.; Zhao, L.; Hou, R.; Zhang, L.; Meng, D. Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks. In Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Washington, DC, USA, 16-20 February 2019; pp. 264-276
24. Vougioukas, I.; Nikoleris, N.; Sandberg, A.; Diestelhorst, S.; Al-Hashimi, B. M.; Merrett, G. V. BRB: Mitigating Branch Predictor Side-Channels. In Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA), Washington, DC, USA, 16-20 February 2019; pp. 466-477.
25. Zhang, Z.; Wang, X.; Hao, Q.; Xu, D.; Zhang, J.; Liu, J.; Ma, J. High-Efficiency Parallel Cryptographic Accelerator for Real-Time Guaranteeing Dynamic Data Security in Embedded Systems. *Micromachines* **2021**, *12*, 560.
26. Bahar Talukder, B. M. S.; Ferdaus, F.; Rahman, M. T. Memory-Based PUFs are Vulnerable as Well: A Non-Invasive Attack Against SRAM PUFs. *IEEE Trans. on Information Forensics and Security* **2021**, *16*, 4035-4049.
27. Ge, L.; Parhi, K. K. Molecular MUX-Based Physical Unclonable Functions. In Proceedings of the 2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Limassol, Cyprus, 06-08 July 2020; pp. 482-487.
28. Hamza, M. A.; Issa, H. H.; Eisa, S. FPGA-Based Modified Ring Oscillator Physical Unclonable Function for Internet of Vehicles. In Proceedings of the 2023 40th National Radio Science Conference (NRSC), Giza, Egypt, 30 May 2023 - 01 June 2023; pp. 208-216.
29. Alkatheiri, M. S.; Zhuang, Y. Towards fast and accurate machine learning attacks of feed-forward arbiter PUFs. In Proceedings of the 2017 IEEE Conference on Dependable and Secure Computing, Taipei, Taiwan, 07-10 August 2017; pp. 181-187.
30. Avvaru, S. V. S.; Zeng, Z.; Parhi, K. K. Homogeneous and Heterogeneous Feed-Forward XOR Physical Unclonable Functions. *IEEE Trans. Information Forensics and Security* **2020**, *15*, pp. 2485-2498.
31. Maes, R.; Herrewege, A. V.; Verbauwhede, I. PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator. In Proceedings of the Cryptographic Hardware and Embedded Systems -- CHES 2012, Leuven, Belgium, 9-12 September 2012; pp. 302-319.
32. Wang, X.; Wang, W.; Xu, B.; Du, P. A fine-grained hardware security approach for runtime code integrity in embedded systems. *J. Univers. Comput. Sci.* **2018**, *24*, 515-536.
33. Wu, W.; Wu, S.; Zhang, L.; Zou, J.; Dong, L. LHash: A lightweight hash function. In Proceedings of the Springer International Conference on Information Security and Cryptology, Guangzhou, China, 27–30 November 2013; pp. 291–308.
34. T-head-Semi/opene906. Available online: https://github.com/T-head-Semi/opene906 (18 October 2021).
35. M, I.; Kaur, M.; Raj, M.; R, S.; Lee, H.-N. Cross Channel Scripting and Code Injection Attacks on Web and Cloud-Based Applications: A Comprehensive Review. *Sensors* **2022**, *22*, 1959.
36. Salehi, M.; Hughes, D.; Crispo, B. MicroGuard: Securing Bare-Metal Microcontrollers against Code-Reuse Attacks. In Proceedings of the 2019 IEEE Conference on Dependable and Secure Computing (DSC), Hangzhou, China, 18-20 November 2019; pp. 1-8.
37. Xu, C.; Zhang, L.; Law, M. -K.; Zhao, X.; Mak, P. -I.; Martins, R. P. Modeling Attack Resistant Strong PUF Exploiting Stagewise Obfuscated Interconnections With Improved Reliability. *IEEE Internet of Things J.* **2023**, *10*, 16300 - 16315.
38. Alamro, M. A.; Zhuang, Y.; Aseeri, A. O.; Alkatheiri, M. S. Examination of Double Arbiter PUFs on Security against Machine Learning Attacks. In Proceedings of the 2019 IEEE International Conference on Big Data (Big Data), Los Angeles, CA, USA, 09-12 December 2019; pp. 3165-3171.