World Academy of Science, Engineering and Technology
International Journal of Cognitive and Language Sciences
Vol:18, No:8, 2024

Security Design of Root of Trust Based on RISC-V

Kang Huang, Wanting Zhou, Shiwei Yuan, Lei Li

Abstract—Since information technology develops rapidly, the security issue has become an increasingly critical for computer system. In particular, as cloud computing and the Internet of Things (IoT) continue to gain widespread adoption, computer systems need to new security threats and attacks. The Root of Trust (RoT) is the foundation for providing basic trusted computing, which is used to verify the security and trustworthiness of other components. Designing a reliable RoT and guaranteeing its own security are essential for improving the overall security and credibility of computer systems. In this paper, we discuss the implementation of self-security technology based on the RISC-V RoT at the hardware level. To effectively safeguard the security of the RoT, researches on security safeguard technology on the RoT have been studied. At first, a lightweight and secure boot framework is proposed as a secure mechanism. Secondly, two kinds of memory protection mechanism are built to against memory attacks. Moreover, hardware implementation of proposed method has been also investigated. A series of experiments and tests have been carried on to verify to effectiveness of the proposed method. The experimental results demonstrated that the proposed approach is effective in verifying the integrity of the RoT's own boot rom, user instructions, and data, ensuring authenticity and enabling the secure boot of the RoT's own system. Additionally, our approach provides memory protection against certain types of memory attacks, such as cache leaks and tampering, and ensures the security of root-of-trust sensitive information, including keys.

Keywords—Root of Trust, secure boot, memory protection, hardware security.

I. Introduction

WITH the rapid development and wide application of information technology, computer systems are facing more and more security threats and attack methods. In order to ensure the security and trustworthiness of computer systems, the design and implementation of security infrastructure becomes crucial. The RoT is an important part of the security infrastructure, and it undertakes the important task of protecting the system from external attacks. A RoT is a computer system component that is considered secure and can be used to verify the security and trustworthiness of other components. The RoT usually includes hardware and software. The hardware part refers to the security module in the chip, while the software part refers to the security software used to verify and manage the hardware part [1], [2].

A RoT is a source that can always be trusted in a cryptographic system. Since cryptographic security depends on keys that encrypt and decrypt data, and perform functions such as generating digital signatures and verifying signatures, RoT schemes typically include a hardened hardware module. A prime example is a hardware security module (HSM), which generates and protects keys and performs cryptographic functions within its secure environment. Because the module is for all intents and purposes inaccessible outside the computer ecosystem, the ecosystem can trust the keys and other encrypted information it receives from the root of the trust module, i.e. authentic and authorized. This is especially important as the Internet of Things (IoT) proliferates, because to avoid being hacked, components of the computing ecosystem need a way to be sure that the information they receive is authentic. RoT protects the security of data and applications and helps build trust in the entire ecosystem [3].

Therefore, the self-security based on the RISC-V Root of Trust is very important to the overall security and credibility of the computer system. The RoT is an important part of the system security infrastructure, and it plays the role of verifying the security and trustworthiness of other components. If the RoT itself is not secure, the security and trustworthiness of the entire system will be seriously threatened. If an attacker is able to successfully attack the RoT, they will be able to tamper or forge the output of the RoT, allowing other components of the system to be compromised. Attackers may take advantage of RoT vulnerabilities to implement various attacks, such as side-channel attacks, buffer overflow attacks, stack overflow attacks, and so on. These attacks can lead to system crashes, exposure of sensitive data, or in worst cases, complete control of entire systems.

For the security and trustworthiness of computer systems, the security of the RoT itself is very important. Based on the RISC-V RoT's own security technology, it can protect the RoT from external attacks and threats, thereby improving the security and credibility of the entire system. This technology includes the design of both hardware and software, and proposes a series of security measures against possible attack methods, and conducts feasibility verification. Through this technology, the security of the RoT itself can be ensured, thereby making the entire system more secure and trustworthy.

In the past research on the RoT, Gui [4] designed the RoT for electronic control units. The author proposed a secure framework and a hardware mechanism to mitigate denial of service attacks, mainly using TPM for secure boot, Decryption, and designed a black and white list to filter malicious frames, but in the entire system, there is no memory protection mechanism related to memory leaks and tampering. If the RoT does not protect its own memory security, the key data in the RoT will be vulnerable. Buffer overflow and stack attacks are very deadly threats to the system. Moreover, the author uses TPM in the electronic control unit system to realize its own safe startup. This is not a lightweight safe startup, which will increase the power consumption and cost of the product. At the same time, the hardware RoT of the low-power SoC edge device designed by Ehret [5] also lacks the corresponding

Kang Huang is with the Integrated Circuit Engineering, University of Electronic Science and Technology of China, Chengdu, Sichuan, China, 611731 (e-mail: hk_0777@163.com).
Wanting Zhou, Shiwei Yuan, and Lei Li are with University of Electronic Science and Technology of China.

memory protection mechanism of the RoT itself.

At the level of hardware implementation, we will use PULPino as a research platform to discuss the security technology based on the RISC-V RoT, mainly for the research on the lightweight secure boot and memory security of the RoT itself.

A. Secure Boot

In order to achieve secure boot of system devices, some advanced technologies, such as Trusted Platform Module (TPM) [6], Arm TrustZone [7], Microsoft's fTPM [8], Intel's SGX [9] have been designed and implemented, To protect the operating system kernel and application software running on these computer system devices. However, the above-mentioned technologies are not fully applicable to embedded devices, and the resources they occupy are difficult to support for the RoT of embedded devices.

This paper proposed a lightweight and secure boot framework for a secure boot mechanism based on the RISC-V processor RoT, using the SM3 national secret digest algorithm to perform hash calculations on Boot Rom, user instructions, and user data for subsequent integrity checks of hash value, so as to ensure that the key data has not been modified; using the AES symmetric encryption algorithm to encrypt the hash value generated by the user command and user data on the HOST side, and realizing the signature operation. During the startup process of the device side, the device side decrypts the encrypted digest (ciphertext), compares the user instructions loaded into memory with the user data calculation digest, compares the calculated digest with the decrypted digest, and verifies the consistency and ensures the authenticity of the data. The key required for AES symmetric encryption is dynamically and randomly implemented through a Key Derivation Function (KDF) circuit.

B. Memory Protection

For the RoT's own security, the key link is to ensure the security of memory and prevent data from being tampered with. Among memory attacks, buffer overflow attacks [10] are very common. The boundary-checked buffer overflows to overwrite the existing data in the buffer. For example, the attack covers the key data of the dynamic stack of the buffer, causing the program to run incorrectly; overwriting its return address, enabling the attacker to hijack the control flow of the processor.

This article will give two memory protection mechanisms: (1) to ensure that the return address stored in the dynamic stack cannot be tampered with. Once the attacker initiates a buffer overflow attack, the mechanism can detect the malicious attack in advance and propose a blocking buffer area overflow attack; (2) to complete the pre-detection and defense of the buffer overflow attack covering the return address.

C. The Work and Contributions of This Article

In summary, we made the following contributions in this paper: (1) A lightweight and secure boot framework is proposed for a secure boot mechanism based on the RISC-V processor RoT to complete data integrity verification and ensure data authenticity; (2) In view of the RoT's own memory attack and the possibility of data tampering, two memory protection mechanisms are proposed to protect RoT from program control flow hijacking attacks, thus ensuring the high security and reliability of RoT itself.

II. Secure Boot Based on RoT Self-Security

Secure Boot is a general-purpose technology that is primarily based on digest algorithms or symmetric/asymmetric encryption algorithms to perform integrity checks and signature verification of processor firmware and cores. This paper proposes a lightweight and secure boot framework based on the secure boot mechanism of the RISC-V processor RoT [11]. The mechanism described in this section is based on the SM3 NATIONAL SECRET DIGEST Algorithm and the AES SYMMETRIC ENCRYPTION Algorithm to implement a secure processor startup mechanism to ensure the integrity and authenticity of key data to ensure that the Boot Code, user instructions, and user data against tamper. This paper uses hardware description language to implement SM3 and AES algorithms based on literature [12], [13], and performs its hardware implementation into the core of PULPino for experiments and verification.

A. Processor Secure Boot Design

Before the PULPino processor executes the Boot Code and starts the pipeline, it is first necessary to design the Boot Rom safety startup circuit to ensure the integrity of the Boot Rom. After the integrity verification of the Boot Rom is completed, the system starts the processor pipeline. Boot Code starts running. The integrity check of the Boot Code is very important, which includes initializing registers and stacks, loading user data and user instructions in RAM to DTCM and ITCM in the processor core domain, and performing program jumps. The Boot Rom secure startup circuit is mainly divided into two parts: 1) The hardware circuit of the national secret SM3 hash algorithm; 2) The control circuit of the Boot Code hash value calculation. The main functions of the control circuit are: a) to control the operation of the Boot Rom safe start circuit; b) to control the Boot Code to enter the SM3 hardware circuit in sequence according to 512Bit groups, that is, the current group information is compressed for 64 times before entering the next group of information; c) to give the indication signal and effective bit of the last 32Bit data, for Boot Code, its last_word_byte_in = 2'b11; d) to compare the Boot Code hash value calculated by the SM3 hardware circuit with the Boot Code reference hash stored in the processor security register to generate a processor start signal. To sum up, the implementation functional block diagram of the Boot Rom secure startup circuit is shown in Fig. 1.

The AES algorithm is a symmetric key algorithm and AES-CTR is an encryption mode within AES. The SM3 and AES-CTR are implemented in a hardware description language and the hardware implementation is embedded in the PULPino kernel. After the execution of the previous Boot

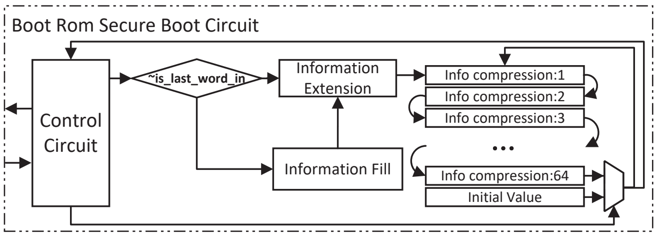


Fig. 1 Functional block diagram of Boot Rom secure boot circuit design and implementation

Rom secure boot circuit is completed, a signal is generated to turn on the processor pipeline and the processor kernel starts to execute the Boot Code data, during which the HOST side first uses SM3 to read the compiled and generated user instruction file I2_stim.slm and the user data file tcdm_bank0.slm by means of registers After that, the hash value of Hash_itcm_reg and Hash_dtcm_reg are obtained, and the two hash values are encrypted by AES-CTR mode to ensure the authenticity of the data; subsequently, the device side decrypts the encrypted digest (cipher text) to obtain the Hash_value. Then, the user instruction ITCM and the user data DTCM are loaded into memory, and the summary Hash_run is computed, and the consistency between the computed summary and the decrypted summary is compared respectively, and if the consistency is passed, the subsequent instructions are executed to complete the integrity check of the user instruction and the user data and ensure the authenticity of the data. If not, the processor kernel will stop running. The implementation block diagram is shown in Fig. 2.

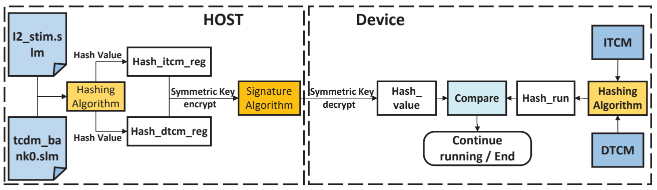


Fig. 2 Secure boot design for user instructions and user data

B. Key Derivation Circuit Design

The key derivation function (Key Derivation Function, KDF) circuit is the core module to realize the dynamic randomization mechanism. According to the KDF circuit, the key required by the processor to encrypt key information can be dynamically and randomly obtained, such as the above AES symmetric encryption. The symmetric key is generated by the KDF circuit. From the literature [14], it can be known that KDF is implemented by TRNG and PUF XOR and then through the digest algorithm. The implementation diagram is shown in Fig. 3), and the core modules are: TRNG, PUF and digest algorithm. The realization of TRNG and PUF both use the ring oscillator as the basic unit module.

In this paper, the RO-TRNG is realized by using the principle that the phase jitter of the ring oscillator obeys the Gaussian distribution. The 72MB random numbers generated by RO-TRNG were collected, and then the correlation

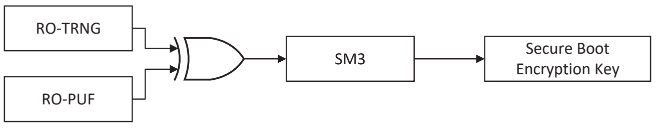


Fig. 3 Sketch of the hardware circuit implementation of the key export function

coefficient between the average entropy per bit of RO-TRNG and the random number sequence was calculated using the ent test. The test results are shown in Table I.

TABLE I
RO-TRNG DIEHARDER TEST RESULTS FOR RANDOM DATA

Test Name	Test Number	Test Result
Diehard Birthdays Test	1	passed
Diehard OPERM5 Test	1	passed
Diehard 32x32 Binary Rank Test	1	passed
Diehard 6x8 Binary Rank Test	1	passed
Diehard Bitstream Test	1	passed
Diehard Count the 1s (stream) Test	1	passed
Diehard Parking Lot Test	1	passed
Diehard Minimum Distance (2d Circle) Test	1	passed
Diehard 3d Sphere (Minimum Distance) Test	1	passed
Diehard Squeeze Test	1	passed
Diehard Runs Test	2	1passed 1weak
Marsaglia and Tsang GCD Test	2	passed
STS Monobit Test	1	passed
STS Runs Test	1	passed
STS Serial Test (Generalized)	30	passed
RGB Bit Distribution Test	12	passed
RGB Generalized Minimum Distance Test	4	passed
RGB Permutations Test	4	passed
RGB Lagged Sum Test	33	passed
RGB Kolmogorov-Smirnov Test Test	1	passed
Byte Distribution	1	passed
DAB DCT	1	passed
DAB Fill Tree Test	2	passed
DAB Fill Tree 2 Test	2	passed
DAB Monobit 2 Test	1	passed

The oscillation frequency of different ROs deployed in an IC or FPGA can vary slightly due to differences in physical characteristics such as capacitance and wiring delay. These variations can lead to small differences in the RO oscillation frequency at different locations, even when the ROs are of the same type. Taking advantage of this characteristic, this paper proposes a design and implementation of an RO-PUF for a KDF circuit.

To demonstrate the robustness and stability of the RO-PUF, we deployed it on the ZYNQ 7010 FPGA and obtained 4096 PUF ID values continuously. We then calculated the on-chip Hamming distance of the PUF using (1), which resulted in a value of only 0.04%. These results clearly demonstrate that the proposed RO-PUF has superior robustness and stability.

$$\mu_{intra}\% = \frac{1}{MN}\sum_{i=1}^{M} h_{ri} \cdot 100\% \qquad (1)$$

The uniqueness of RO-PUF was verified by calculating the inter-slice Hamming distance of RO-PUF, and the average inter-slice Hamming distance was calculated as shown (2);

$$\mu_{inter}\% = \frac{2}{k(k-1)}\sum_{i=1}^{i=k-1}\sum_{j=i+1}^{j=k}\frac{h_{ri}}{n} \cdot 100\% \qquad (2)$$

The RO-PUF designed in this subsection is deployed to 23 FPGAs to collect RO-PUF data and its RO-PUF ID value, and the collected data are averaged to find the inter-piece Hamming distance and standard deviation, and the distribution statistics of the inter-piece Hamming distance are completed, as shown in Fig. 4. It follows a Gaussian distribution with the mean value of 0.4984 and standard deviation of 0.0482. The average inter-piece Hamming distance of the RO-PUF designed and implemented in this subsection is $\mu_{inter}\% = 49.84\%$, which is very close to the ideal value of 50%, which proves that the RO-PUF designed and implemented in this subsection has a high uniqueness.

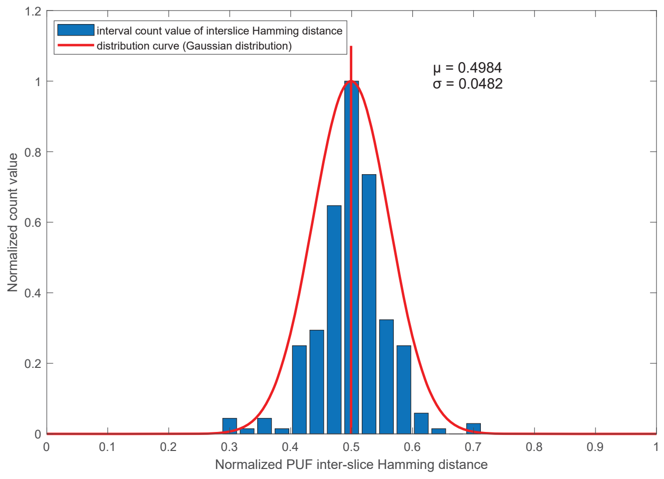


Fig. 4 RO-PUF inter-piece Hamming distance test results

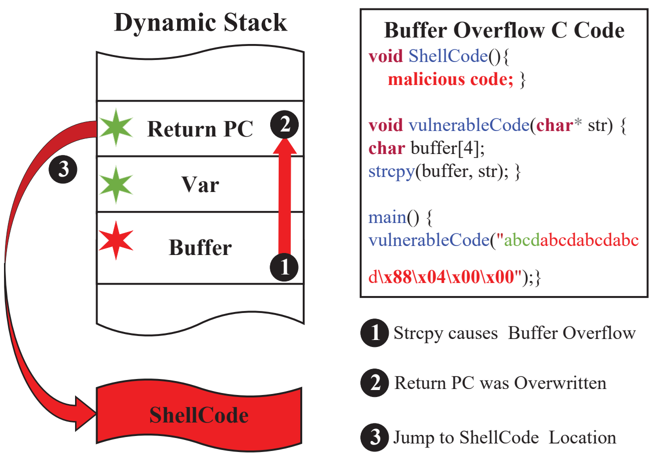
III. Memory Protection Based on Self-Security of RoT

For the RoT's own security, a very critical part is to protect the security of the memory and prevent data tampering. The processor dynamic stack is a buffer for storing function return addresses and local variable running results. Once the dynamic stack data is tampered with, it will cause changes in the processor's running results and running track, and more seriously, the processor's control flow will be hijacked by the attacker. In order to solve the security problem of processor dynamic stack tampering caused by hardware vulnerabilities, two memory protection mechanisms will be proposed below to protect the memory of the RoT itself.

A. Memory Protection Mechanism 1

Buffer overflows (shown as Fig. 5) are the most common vulnerability and can be used to launch a variety of attacks. In an unbounded checker, an attacker can exploit redundant data from user input overloads that may exceed the buffer capacity and potentially malicious data overwriting nearby memory locations and complete hijacking of the processor's control flow, such as in return-oriented programming (ROP), where function pointers and operations violate the integrity of the data flow.

The memory protection mechanism described in this section starts with ensuring the correctness of the return address


Dynamic Stack
Return PC
Var
Buffer
ShellCode
1
2
3
1 Strcpy causes Buffer Overflow
2 Return PC was Overwritten
3 Jump to ShellCode Location

Buffer Overflow C Code

```
void ShellCode(){
   malicious code; }

void vulnerableCode(char* str) {
char buffer[4];
strcpy(buffer, str); }

main() {
vulnerableCode("abcdabcdabcdabcd\x88\x04\x00\x00");}
```

Fig. 5 Buffer overflow exploit

stored in the dynamic stack of the processor, builds the stack of the processor program control flow graph in real time during the program running, and generates a restricted space. According to the program control flow chart stack, The number of unreachable store instructions in the pre-run program. This ensures that the return address stored in the dynamic stack cannot be tampered with. Once the attacker launches a buffer overflow attack, the mechanism in this section can detect the malicious attack in advance and propose to block the buffer overflow attack. The functional structure block diagram of the memory protection mechanism described in this section is shown in Fig. 6.

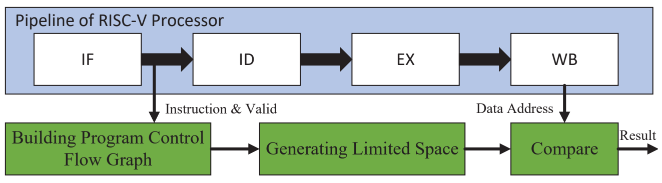


Fig. 6 The implementation diagram of memory protection mechanism 1

1) Building Program Control Flow Graph in Real-Time: The instruction and effective signal of RISC-V processor decoding stage are acquired in real time. FSM (Finite State Machine) and RAM are used to construct program control flow diagrams in real time. The FSM status jump is shown Fig. 7. C1 - C5 in Fig. 7 is the state jump condition, and its corresponding meaning is:

- C1: *JAL* is matched from the instruction flow in the ID stage of RISC-V processor, and the *register rd* is register X1;
- C2: *SW* is matched from the instruction flow in the ID stage of RISC-V processor, and the *register rs2* is register X1;
- C3: *JALR* is matched from the instruction flow in the ID stage of RISC-V processor;
- C4: *LW* is matched from the instruction flow in the ID stage of RISC-V processor, and the *register rd* is register X1;
- C5: *JALR* is matched from the instruction flow in the ID

stage of RISC-V processor.

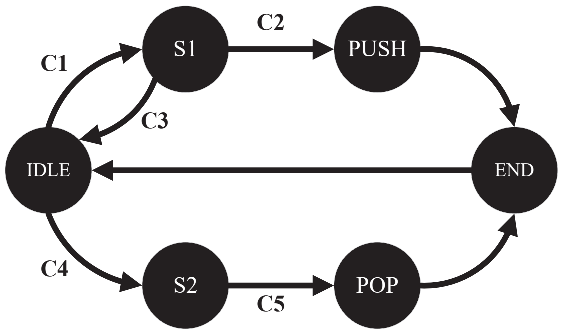


Fig. 7 State transition of FSM for building PCFG in Real-Time

In the PUSH state, the function return address (x1) and its corresponding buffer data address (X2 + IMM) are pushed into the stack. In the POP state, the stack composed of RAM is pushed out. From this, the program control flow graph is builded in real-time.

2) Boundary Space and Detection Result Generation: (1) Restricted space generation: According to the PCFG constructed in real time, the range of restricted space that is not accessible by Store instruction in the current function segment is generated. Take PULPino as an example: (a) when the FSM is in PUSH state, the maximum value of the restricted space range is the data address MAX_A1 corresponding to the current return address into the buffer, and the restricted space range is (MAX_A1-4, MAX_A1]; (b) when the FSM is in POP state, the POP operation is performed, and the maximum value of the restricted The maximum value of the space range is the POP out data address MAX_A2, and the range of the restricted space is (MAX_A2-4, MAX_A2]; (2) Generation of detection results: the data address REAL-TIME_A in the access buffer of the processor write-back stage is collected in real time, and it is judged whether REAL-TIME_A is in the restricted space, and if REAL-TIME_A belongs to the restricted space, there is a malicious attack to overwrite the return address of the processor function; (3) Blocking the malicious attack of overwriting the return address of the processor function: when the protection mechanism gives a warning, first suspend all the flow of the processor to avoid the buffer overflow attack to complete the overwriting of the return address of the function; then block the permission of the Store execution access buffer executed by the current function segment; finally resume all the flow of the processor to complete the blocking of the buffer overflow attack.

B. Memory Protection Mechanism 2

The memory protection mechanisms described in this section can be used to detect and defend against buffer overflow attacks in advance. Memory protection mechanism 2 can realize pre-detection and pre-defense against array out-of-bounds, but at the software level, it depends on the size of the cache space actually requested by the current function segment passed by the custom extension instruction, so there is a risk of being bypassed by the attacker. Memory protection mechanism 1 can only detect and defend against buffer overflow attacks to overwrite the return address in advance. Its advantage is that it is completely implemented by hardware, and attackers cannot bypass it. The two protection mechanisms complement each other and jointly deal with the security threats brought by buffer overflow attacks. Memory protection mechanism 2 is mainly composed of three parts, including custom extended instructions (”Guard”), SIIDM (Store Instruction Information Decoding Module, SIIDM) and HBCB (Hardware Boundary Checking for Buffer, HBCB). Memory protection mechanism 2 is deployed in the PULPino kernel, and its implementation block diagram is shown in Fig. 8.

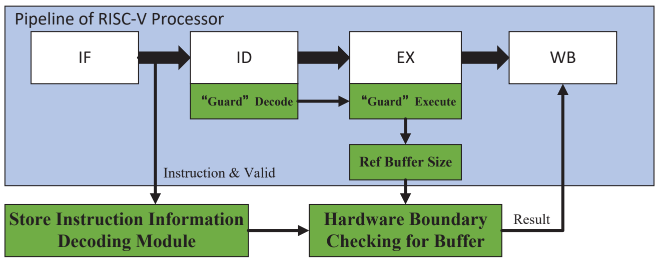

Pipeline of RISC-V Processor
IF
ID
EX
WB
"Guard" Decode
"Guard" Execute
Ref Buffer Size
Instruction & Valid
Store Instruction Information Decoding Module
Hardware Boundary Checking for Buffer
Result

Fig. 8 The implementation diagram of memory protection mechanism 2

1) Custom Extension Instruction – ”Guard”: According to the custom extension of RISC-V processor Instruction Set Architecture (ISA), the Custom Extension Instruction – ”Guard” is realized by modifying the source code of part of the compiler tool chain of the processor and the hardware logic of the ID stage and EX stage of the kernel. The ”Guard” instruction is executed in the RISC-V processor to generate the driver signal of this detection mechanism, and at the same time, it can generate REF_BS(Reference Buffer Size, REF_BS) of the buffer. So that This mechanism can use REF_BS to implement the boundary checking at the hardware level of the buffer. When writing C code, the ”Guard” instruction is added before the function that may cause buffer overflow, such as: gets(), fgets(), strcpy(), strncpy(), strlen(), etc., so as to realize the buffer boundary check on the processor hardware level. The encoding and meaning of ”guard” instruction are shown in Table II.

TABLE II
THE ENCODING AND MEANING OF ”GUARD” INSTRUCTION

insn	imm12	func	const5	opcdoe	operation
Guard	12’h?	8’h01	5’h00	7’h77	$\{20'b0, imm12\} \longrightarrow s_0$

By decoding and executing ”Guard” instruction in RISC-V processor, the immediate data carried by ”Guard” are stored in the security register s_0. For example, *asm volatile (”Guard s0, 12 ”)* means that the ”Guard” passes the buffer size value 12 requested by the function at the C code level to the security register s_0.

2) Store Instruction Information Decoding Module: In RISC-V processor, the instruction that stores data in the buffer is only Store instruction, so the size of the buffer consumed by the current function segment of RISC-V processor can be extracted by parsing Store instruction information in real-time. Based on this, this section designs HBCB to complete the extraction of the size of the buffer consumed by the current function segment.

According to SIIDM, analyzing whether the currently collected valid instructions is the type of Store instruction, whether the immediate data carried by the Store instruction is greater than or equal to 0, and Whether the increment Δimm of the immediate data carried by the Store instruction compared with that carried by the previous Store instruction is 0 or 1 or 2 or 4. If the above conditions are met at the same time, the size value Y_i of buffer space to be consumed by the current Store instruction is generated according to the funct3 encoding of the instruction. The calculation formula of Y_i is as (3):

$$Y_i = \begin{cases} 4 & StoreType == Word \\ 2 & StoreType == HalfWord \\ 1 & StoreType == Byte \\ 0 & others. \end{cases} \tag{3}$$

The value of i ranges from 1 to N, N refers to the maximum count value of continuously detected valid instructions that meet the conditions. Based on the obtained Y_i, calculating the size of the buffer consumed in the current function segment in real time, The calculation formula is as (4):

$$RT\_BS = \sum_{i=1}^{N} Y_i. \tag{4}$$

3) Hardware Boundary Checking for Buffer: This module is mainly used to detect and block buffer overflow in real-time. After the execution of ”Guard” instruction, we read REF_BS stored in the security register s_0 in real-time. At the same time, we obtain the size value RT_BS of buffer consumed by the current function segment extracted and calculated by SIIDM. HBCB compares REF_BS with RT_BS in real time. If $REF\_BS - RT\_BS \geq 0$ indicates that there is no buffer overflow attack in the current function segment, otherwise, there is a buffer overflow attack in the current function segment.

According to the result of HBCB, the pipeline of processor is stop immediately to avoid the buffer overflow attack to complete the overwriting of the return address of the function. Then, disable the permissions of the Store instructions executing during the current function segment to store the buffer. Finally, all the pipeline of processor is restored and the buffer overflow attack is blocked.

IV. Experiments and Analysis of Results

A. Secure Boot Experiment and Result Analysis Based on RoT Self-Security

1) Secure Boot Experiment and Result Analysis: The Boot Rom secure boot circuit plays a critical role in ensuring the security of the RoT. As depicted in Fig. 9, this circuit calculates the hash value of the Boot Code and generates a simulation waveform of the secure boot signal. The hash value is obtained and can be observed in the sm3_finished_out signal. In order to initiate the processor pipeline, the hash value of the Boot Code must be compared with the hash value of the security register stored in the processor. If these values match, the pipeline start signal is activated, allowing the processor to commence operation. This process provides an added layer of security to the RoT by verifying the integrity of the Boot Code and ensuring that it has not been tampered with.

After the pipeline is turned on, the Boot Code command performs an integrity check on the user command and user data and verifies the authenticity of the data, and compares whether the calculated digest is consistent with the decrypted digest. If they are consistent, the verification passes and the subsequent commands are continued, the simulation waveform is shown in Fig. 10.

The corresponding programs were run on PULPino's experimental platform, and the results of their code and platform runs are shown in Fig. 11.

2) Key Derivation Circuit Experiment and Result Analysis: The implemented RO-TRNG, RO-PUF and SM3 are created as per Fig. 3 and deployed into the ZYNQ7010 FPGA, and then the function of this KDF circuit is tested. Using ila to trigger the final output valid signal sm3_finished_out of the KDF circuit, the result of the KDF circuit function test is shown in Fig. 12.

The PUF ID is 96'h366074446d7514e960ec200d, and the extracted 96Bit true random number is 96 h6dfaee3931c31ec8700a08d0, the PUF ID value is 96'h5b9a9a7d5cb60a2110e628dd, and the output of the key derivation function circuit this time is 256' h3b730147_0d720089_2f1b4412c698fd22_94e209b6d2f7fa24 _b8acde3c354a9a8c, which is consistent with the value calculated by the SM3 online calculation tool.

B. Experiments and Results Analysis of Memory Protection Based on Trust Root Self-Security

Since PULPino's compilation toolchain does not perform a boundary check on the size of the dynamic stack space requested by the function, it is assumed that the attacker knows the physical entry address of the malicious code function and plants a simple buffer overflow attack in the C program. To simply run the example attack in PULPino, this thesis writes the shellcode directly into the C program. The example program with the buffer overflow attack is run on the baseline processor and on the processor with secure memory protection (the example program on the processor with secure memory protection has an additional ”guard” instruction before strcpy), and the code and its results on both platforms are shown in Fig. 13. After the baseline processor suffers a buffer overflow attack, the return address of func1 (PC=0x4d0) is changed to the entry address of func2 (PC=0x488), thus executing the shellcode injected by the attacker, and the program control flow is hijacked by the attacker; the processor with the secure memory protection mechanism is hijacked by the attacker due to the ”guard” instruction and its hardware. The processor with secure memory protection mechanism can ensure the normal operation of the processor due to the ”guard” instruction and its real-time boundary check logic at the hardware level, which

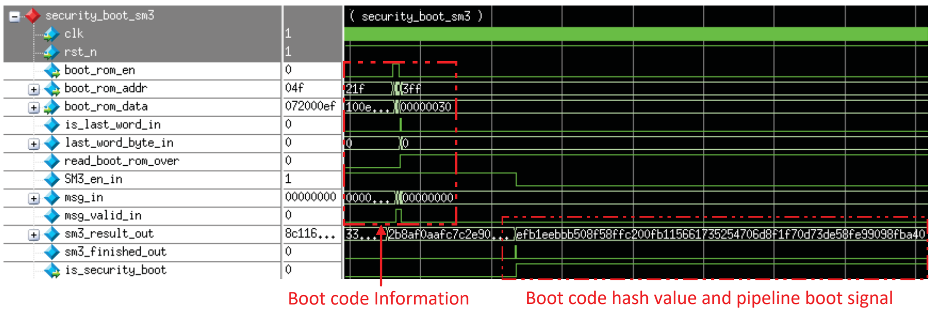


Fig. 9 The simulation waveform of the Boot Code hash value calculated by the processor secure boot circuit

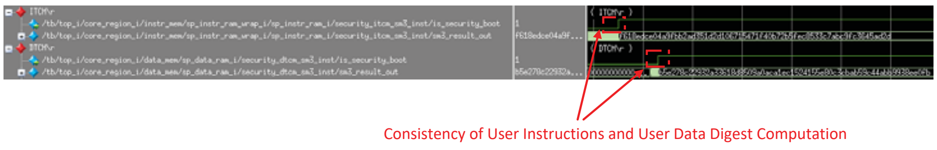
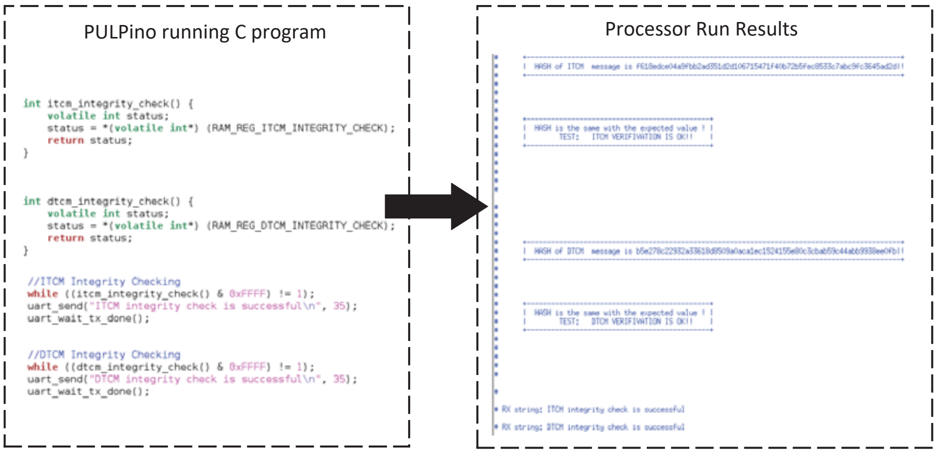


Fig. 10 Consistency check of user instruction and user data



```
int itcm_integrity_check() {
    volatile int status;
    status = *(volatile int*) (RAM_REG_ITCM_INTEGRITY_CHECK);
    return status;
}


int dtcm_integrity_check() {
    volatile int status;
    status = *(volatile int*) (RAM_REG_DTCM_INTEGRITY_CHECK);
    return status;
}

 //ITCM Integrity Checking
 while ((itcm_integrity_check() & 0xFFFF) != 1);
 uart_send("ITCM integrity check is successful\n", 35);
 uart_wait_tx_done();

 //DTCM Integrity Checking
 while ((dtcm_integrity_check() & 0xFFFF) != 1);
 uart_send("DTCM integrity check is successful\n", 35);
 uart_wait_tx_done();
```

```
+------------------------------------------------------------------------------------------+
|  HASH of ITCM  message is f618edce04a9fbb2ad351d2d106715471f40b72b5fec8533c7abc9fc3645ad2d!!
+------------------------------------------------------------------------------------------+

+--------------------------------------------+
|  HASH is the same with the expected value ! |
|       TEST:   ITCM VERIFIWATION IS OK!!     |
+--------------------------------------------+

+------------------------------------------------------------------------------------------+
|  HASH of DTCM  message is b5e278c22932a33618d8509a0aca1ec1524155e80c3cbab59c44abb9938ee0fb!!
+------------------------------------------------------------------------------------------+

+--------------------------------------------+
|  HASH is the same with the expected value ! |
|       TEST:   DTCM VERIFIWATION IS OK!!     |
+--------------------------------------------+

* RX string: ITCM integrity check is successful
* RX string: DTCM integrity check is successful
```

Fig. 11 User instructions and user data verification results running on the processor

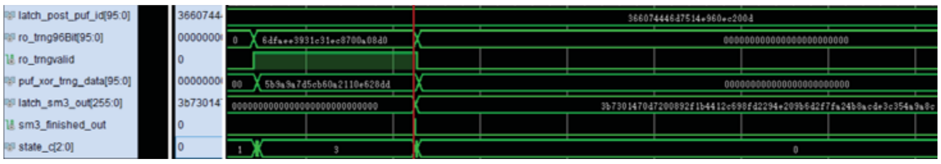


Fig. 12 Functional test results of the key derivation function circuit

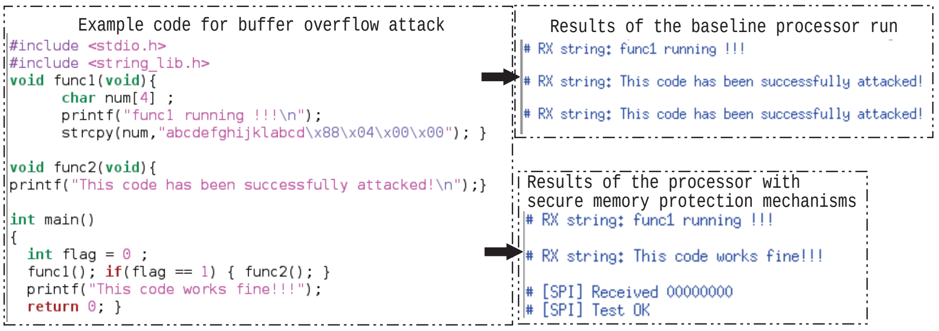
completes the pre-detection and defense of the processor against buffer overflow, thus invalidating the buffer overflow attack on the processor.

V. Conclusion

In our own security technology based on the RISC-V RoT, we use secure boot and memory protection mechanisms to protect the RoT from external attacks and threats, thus improving the security and trustworthiness of the entire system. For secure boot, we implemented a series of verification and authentication steps to ensure the security of the system during the boot process. First, we use a secure boot circuit module to verify the integrity of the Boot Code and check for modifications. Subsequently, we use digital signature techniques to verify the authenticity and integrity of user commands and user data to ensure that they have not been tampered with. For memory protection mechanism, we use hardware and software-based memory protection mechanism to prevent memory leakage and tampering. To verify the effectiveness of our proposed technique, we conducted a series of experimental tests. The results show that our proposed technique can ensure the security of the RoT itself to a certain extent, thus improving the security and trustworthiness of the whole system.

Acknowledgment

This work was supported by Stability Program of Science and Technology on Communication Security


Example code for buffer overflow attack
```
#include <stdio.h>
#include <string_lib.h>
void func1(void){
      char num[4] ;
      printf("func1 running !!!\n");
      strcpy(num,"abcdefghijklabcd\x88\x04\x00\x00"); }

void func2(void){
printf("This code has been successfully attacked!\n");}

int main()
{
  int flag = 0 ;
  func1(); if(flag == 1) { func2(); }
  printf("This code works fine!!!");
  return 0; }
```
Results of the baseline processor run
```
# RX string: func1 running !!!

# RX string: This code has been successfully attacked!

# RX string: This code has been successfully attacked!
```
Results of the processor with secure memory protection mechanisms
```
# RX string: func1 running !!!

# RX string: This code works fine!!!

# [SPI] Received 00000000
# [SPI] Test OK
```

Fig. 13 Results of the buffer overflow attack example program running on a baseline versus a processor with a secure memory protection mechanism

Laboratory (2022) and Sichuan Science and Technology Program(No.2021YJ0082).

REFERENCES

[1] R. Perez, "Silicon systems security and building a root of trust," in *2015 IEEE Asian Solid-State Circuits Conference (A-SSCC)*. IEEE, 2015, pp. 1–4.
[2] B. Møller, M. Pedersen, and T. Bøgedal, "Formally verifying security properties for opentitan boot code with uppaal," *To Appear. MA thesis. AAU*, 2021.
[3] T. Lu, "A survey on risc-v security: Hardware and architecture," *arXiv preprint arXiv:2107.04175*, 2021.
[4] Y. Gui, A. S. Siddiqui, and F. Saqib, "Hardware based root of trust for electronic control units," in *SoutheastCon 2018*. IEEE, 2018, pp. 1–7.
[5] A. Ehret, E. Del Rosario, K. Gettings, and M. A. Kinsy, "A hardware root-of-trust design for low-power soc edge devices," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2020, pp. 1–6.
[6] A. Tomlinson, "Introduction to the tpm," *Smart Cards, Tokens, Security and Applications*, pp. 173–191, 2017.
[7] E. Benhani, L. Bossuet, and A. Aubert, "The security of arm trustzone in a fpga-based soc," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1238–1248, 2019.
[8] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon *et al.*, "ftpm: A firmware-based tpm 2.0 implementation," *Microsoft Research*, pp. 0–23, 2015.
[9] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the intel sgx kingdom with transient out-of-order execution," in *Proceedings fo the 27th USENIX Security Symposium*. USENIX Association, 2018.
[10] N. R. Weidler, D. Brown, S. A. Mitchel, J. Anderson, J. R. Williams, A. Costley, C. Kunz, C. Wilkinson, R. Wehbe, and R. Gerdes, "Return-oriented programming on a cortex-m processor," in *2017 IEEE Trustcom/BigDataSE/ICESS*. IEEE, 2017, pp. 823–832.
[11] J. Haj-Yahya, M. M. Wong, V. Pudi, S. Bhasin, and A. Chattopadhyay, "Lightweight secure-boot architecture for risc-v system-on-chip," in *20th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2019, pp. 216–223.
[12] X. Zheng, X. Hu, J. Zhang, J. Yang, S. Cai, and X. Xiong, "An efficient and low-power design of the sm3 hash algorithm for iot," *Electronics*, vol. 8, no. 9, p. 1033, 2019.
[13] J. Daemen and V. Rijmen, "Aes proposal: Rijndael," 1999.
[14] F. Armknecht and J. Guajardo, "Fourth international workshop on trustworthy embedded devices (trusted 2014)," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1548–1549.

Kang Huang received the bachelor of electronic information engineering from University of South China in 2020. He is currently a master student with the School of University of Electronic Science and Technology of China. His research interests lie in areas of integrated circuit design and hardware security.

Wanting Zhou received the Ph.D. degree in communication and information systems from University of Electronic Science and Technology of China, Chengdu, China, in 2014.She is currently a research associate with University of Electronic Science and Technology of China. Her research interests include integrated circuits, machine Learning and hardware security.

Shiwei Yuan received the master of Electronic science and Technology from University of Electronic Science and Technology of China in 2022.

Lei Li received Ph.D. degree in communication engineering from University of Electronic Science and Technology of China, Chengdu, China, in 2010. He is currently an associate researcher with University of Electronic Science and Technology of China. His current research interests include integrated circuits, machine learning, and hardware security.