Hindawi
Security and Communication Networks
Volume 2020, Article ID 8957641, 14 pages
https://doi.org/10.1155/2020/8957641

WILEY | Hindawi

Research Article

RIPTE: Runtime Integrity Protection Based on Trusted Execution for IoT Device

Yu Qin, Jingbin Liu, Shijun Zhao, Dengguo Feng, and Wei Feng

Institute of Software Chinese Academy of Sciences, Beijing, China

Correspondence should be addressed to Yu Qin; qinyu@iscas.ac.cn

Received 28 March 2019; Revised 26 August 2020; Accepted 4 September 2020; Published 23 September 2020

Academic Editor: Prosanta Gope

Software attacks like worm, botnet, and DDoS are the increasingly serious problems in IoT, which had caused large-scale cyber attack and even breakdown of important information infrastructure. Software measurement and attestation are general methods to detect software integrity and their executing states in IoT. However, they cannot resist TOCTOU attack due to their static features and seldom verify correctness of control flow integrity. In this paper, we propose a novel and practical scheme for software trusted execution based on lightweight trust. Our scheme RIPTE combines dynamic measurement and control flow integrity with PUF device binding key. Through encrypting return address of program function by PUF key, RIPTE can protect software integrity at runtime on IoT device, enabling to prevent the code reuse attacks. The results of our prototype's experiment show that it only increases a small size TCB and has a tiny overhead in IoT devices under the constraint on function calling. In sum, RIPTE is secure and efficient in IoT device protection at runtime.

1. Introduction

1.1. Background. Internet of things and embedded system spread quickly and widely in daily life, which is deployed in intelligent traffic, unmanned aerial vehicle, remote medical, environment monitoring, intelligent home, smart city, factory, etc. Cloud infrastructure extends the capabilities of computing and storage for IoT, and development of LPWAN (Low Power WAN) strengthens IoT's communications, interconnects devices with Internet or mobile Internet conveniently, and also provides rich applications for intelligent hardware in IoT. Statistics from Gartner show that global IoT devices reach 20.8 billion in 2020. IoT security incidents occur frequently. Hackers intruded a large scale of IP cameras, video cameras, and routers in 2016 and conducted DDoS attack to interrupt more than 1200 web server including Twitter and Netflix.

Generally speaking, IoT is vulnerable to malicious attacks, due to lack of security mechanisms on devices. It is a critical problem how to verify remote IoT devices with the expected states. Various schemes are presented to build trust architecture in IoT devices. SMART [1] was an efficient and secure method for establishing a dynamical root of trust in remote IoT device. Intel Lab proposed TrustLite [2] security architecture for trusted execution-based hardware-enforced isolation by lightweight MMU and verified its security capabilities in FPGA prototype for low-cost IoT devices. Tytan [3] improved TrustLite architecture based on MPU (Memory Protection Unit) by TUD, Germany. It started FreeRTOS real time OS on strong isolation provided by security hardware and achieved great performance improvement in real time.

PUFs (Physical Unclonable Functions) [4, 5] are another lightweight method to establish device trust. They have the characteristics of both high robustness and uniqueness, widely applied in device identification and authentication etc. There are many types of PUFs, e.g., SRAM PUF [6, 7], optical PUF [8], Ring Oscillator PUF [9], Flash memory-based PUF [10]. So far, SRAM PUF is the most mature PUF technique in practice. It extracts the unique and reliable device key to build embedded trust by applying fuzzy extractor [11]. Zhao et al. [12] proposed a lightweight root of trust scheme based on SRAM PUF in Trustzone environment, and built the trust architecture from root of trust in an

embedded system. PUF is a convenient and efficient method to construct lightweight trust. It has two obvious advantages: first it can provide a unique device key derived from hardware fingerprint without any storage; second, PUF can bind the system critical data with hardware platform, which reduces the attack surface so that we can use PUF to protect the runtime execution environment for IoT device.

The latest Cortex-M chips such as Cortex-M23/33 support Trustzone technology (called TZ-M [13]) to enhance terminal security. TZ-M can be used to protect firmware, flash image, and peripheral, and support secure boot, trusted execution, and code update for embedded system and software. ARM TrustZone divides platform's computing environment into secure world and normal world based on CPU security extension. Samsung provides Knox [14] for smartphones, which utilizes TZ security features to support business and personal system co-existed in one smartphone. In addition, Samsung released several smartphone products including Galaxy S4, S5, S6, S7, Galaxy Note 3, etc. Knox devices are equipped with root certificates signed by government or enterprise, and what is more, each firmware or software image must be checked by certificates during secure boot. TIMA (Trustzone-based Integrity Measurement Architecture) in Knox measures and monitors kernel integrity at system runtime. Furthermore, TZ is used in fingerprint identification and mobile payment, e.g., Apple's TouchID, Qualcomm Secure Execution Environment, and Huawei smartphone.

Intel presented TXT [15] to establish isolation environment based on CPU security extension, which ensures software integrity and data confidentiality. Flicker [16] utilizes TXT to isolate sensitive codes to untrust os and especially provides fine-grained remote attestation of isolated codes. Its TCB is also very small. Later, a series of schemes [17~20] like TrustVisor advance the research from idea of Flicker. TrustVisor [18] is a solution of virtual machine monitor based on VT feature, realizes the security protection of sensitive codes in isolation environment, and only decreases performance by 7% than the normal system. In cloud computing, Intel SGX (Software Guard Extensions) [21] protects memory codes and data and implements root of trust, remote attestation, and data sealing. Dynamic memory management inside enclave [22] is proposed to extend the SGX instruction set, facilitates memory allocation in enclave initialization for software developers. The approach of isolated OS kernel with Intel SGX [23] is also presented on cloud platform. It is obvious that Intel SGX is gradually improved and quickly advanced.

1.2. Contribution. In sum, TEE (Trusted Execution Environment) technology is quite universal in IoT. In order to protect IoT software runtime integrity, we propose a novel and practical scheme of software trusted execution in this paper. It measures and verifies the embedded program code integrity dynamically and enables to prevent common code reuse attacks. Taking advantage of ARM Trustzone and PUF-based lightweight trust, it can efficiently ensure integrity of code execution and security of code distribution.

(1) It enables binding software and IoT device with PUF key. While only software is released and distributed from IoT server, it can be correctly executed on target device, which implies that the binding relationship is authenticated between codes and device. Even though the control flow integrity of software codes is cracked and bypassed by hacker on one device, the same version software deployed on the other devices cannot be attacked in IoT network, so that it can prevent the worm botnet from infecting IoT network.

(2) It measures function codes of software dynamically during its execution and further ensures the whole software runtime integrity on device. It also eliminates security risk of TOCTOU (Time of Check, Time of Use) [24] attack by reducing time interval significantly between measurement and execution and thus prevents the dynamic-link library hijacking through detecting integrity of software's function codes.

(3) It can prevent code reuse attack such as ROP (Return-Oriented Programming) and JOP (Jump-Oriented Programming). The function return address is encrypted under the PUF key. When the control flow is hijacked through modifying function return address on stack, error return address will cause faults for function return.

(4) It presents a comprehensive evaluation of RIPTE under Mibench, a universal benchmark suite of embedded applications for IoT. We also give out performance optimization for return address decryption, and it can be improved 24.5% by precomputing in decryption.

2. Threat Model and Architecture

Attackers attempt to instrument the malicious codes, tamper the execution environment of trusted software codes, and further control IoT devices and network. According to Dolov-Yao security model [?]3, we consider the following adversaries: network intruder, device cracker, and software code attacker. On one hand, adversaries in our scheme aim to tamper the static firmware or software when the device is upgrading software, so that the device installs the vulnerable software with attacker's malicious codes. On the other hand, they seek to dynamically execute the malicious code or behavior reusing code gadget. And the physical attack, DDoS attack, is beyond our discussion in our threat model.

2.1. System BootKit Attack. The firmware/software in IoT device mostly do not have a code signature, so that the attacker can reverse engineer code, modify it, and fuse arbitrary malicious code to IoT device. Due to the authentication with a weak password or no password, the attacker can easily inject the bootkit into the IoT embedded system. The attacker may tamper the integrity of firmware and software in memory at system bootstrap. In order to pass the verification of memory checksum, the attacker start bootkit

to complete memory copy, substitution, or memory compression to hide the malicious codes in unused memory region. The root of trust and secure bootstrap can prevent this kind of attack by the trusted chain establishment.

2.2. Code Reuse Attack. The attackers' objective is to exploit Code Reuse Attack (CRA) vulnerability to execute malicious behavior and obtain the system privileged rights. The attacker can completely control stack memory regions and reuse the existing codes in the system to construct a set of code gadgets running with malicious behaviors. This attack does not inject any extra malicious codes inside the system, so it is difficult to detect and prevent CRA. In recent years, CRA can bypass secure boot and code signature verification in IOS device to achieve system jailbreak and software unlocking. The common CRA includes ROP, JOP, JIT Spraying, etc.

2.3. Device Compromising Attack. When one IoT device is compromised, its device key is exposed to the intruder. The intruder using the compromised device may distribute malicious code to other devices or servers with the valid device key. Even worse, he may compromise a large number of devices, form a IoT botnet to carry out a large-scale cyber attack.

2.4. Network Communication Attack. The intruder may eavesdrop, intercept, and modify the communication messages between devices. Due to lack of security protection mechanism, the intruder may tamper and replay the messages to reinstall software of the malicious version or old vulnerable version. But this kind of intrusion must compromise the distribution protocol on software upgrading for IoT device firstly.

Our scheme needs some hardware security extensions to support our security architecture. In order to prevent above adversaries, we assume that our system has the following security features:

(i) A tiny trust anchor is embedded in the device. (1) It establishes the static trust chain from device boot and thus guarantees the integrity and security of rich OS and secure OS. (2) It dynamically measures integrity of software codes at runtime on IoT device. Furthermore, it can attest the trustworthiness of software codes to administrator server with fresh nonce. (3) It can derive encryption key from tiny anchor (e.g., SRAM PUF) [12] to protect the return address and indirect jump address in runtime memory.

(ii) Trusted execution environment provided by ARM TrustZone is assumed to protect secure os and its trusted services such as measurement engine, address checking module, etc. Because device key is derived from trust anchor, it does not need auxiliary secure storage to protect key's sensitive data in TEE.

(iii) DEP (Data Execution Prevention) is open to provide system-level memory protection, and it prevents the code injection attack and other malicious code executed from data pages such as the default heap, stacks, and memory pools.

Against these security threats, we design secure execution architecture (shown in Figure 1) which utilizes the ARM TrustZone to protect the integrity and security of the code execution. The tiny anchor SRAM PUF provides secure device key and random source as lightweight root of trust in the hardware layer. In the system layer, TEE-based Trustzone protects the secure OS, cryptographic service module, and program address validation module, which ensures the target codes trust execution. The target to be protected runs in the normal world where attacker can intrude the virus, trojan horse, and other malicious codes to steal sensitive data, promote privileged rights, even completely control IoT device. The Adminstration Server is responsible for validating, distributing, and installing the trustworthy codes to IoT devices.

TrustZone divides environment into normal world and secure world, and main security service modules are located in a secure world. ME (Measurement Engine) measures the pieces of software codes at runtime. ACM (Address Checking Module) is responsible to decrypt the return address and verify its correctness according to control flow integrity. It can optimize the performance of decryption and verification with the support of target CFI table. AS (Attestation Service) signs the measurement log and attests current softwares' running states. In normal world, the software execution is hooked by client kernel module above TrustZone Driver, and Softwre upgrade module is used to verify and install the instrumented software when Administrator Server distributes it securely.

3. RIPTE Design

3.1. Lightweight Trust Establishment. The static environment after device starts up is guaranteed by lightweight trust chain rooted from SRAM PUF. PUF is used as lightweight trust anchor bound with hardware device, which provides trust device identity of hardware fingerprint, secure environment built on device key. PUF is a standard security component, and now it is gradually becoming a commercial application. In our system, we bind PUF and software together to identify the device and protect software's integrity and confidentiality.

The primitive key is essential to protect software execution, which is derived from primary seed (*pufSeed*) generated by PUF. There are two kinds of keys *IdentityKey* and *EncryptionKey* bound with device in our scheme. *IdentityKey* is used to attest device identity and its software states. *EncryptionKey* is used to protect the sensitive memory data. The function KDF adopts NISTs recommendation [25] and key derivation is as follows:

$$IdentityKey := \mathrm{KDF}_a(\mathrm{ALG\_ECC\_256}, pufSeed, \text{“IDENTITY,”} \mathrm{NULL}, \mathrm{NULL}, 256)$$

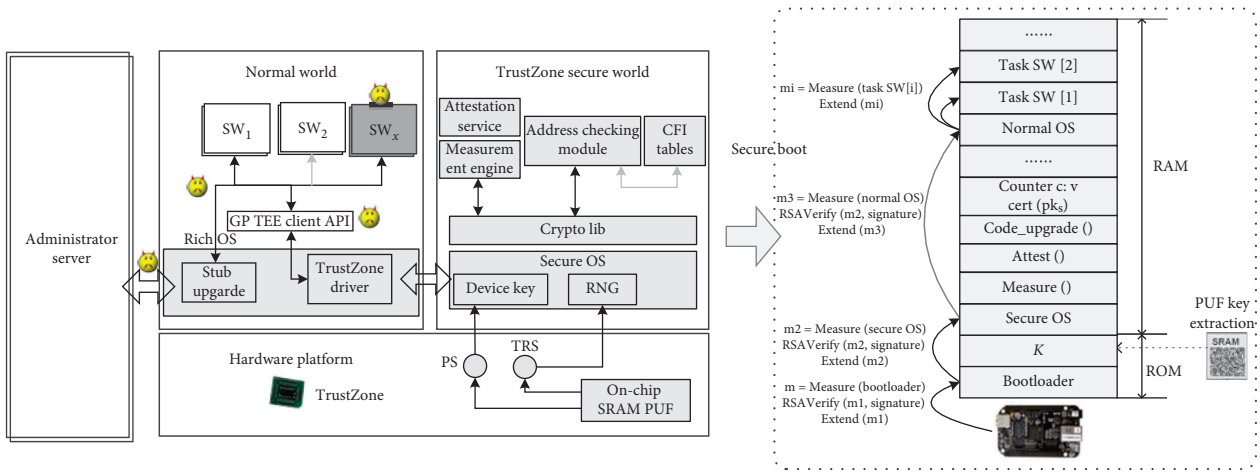

Figure 1: RIPTE architecture and its secure boot based on TEE.

$EncryptionKey$: = KDF_a (ALG_AES_256, *pufSeed*, "ENCRYPTION," "NULL, NULL, 256)

The IoT device starts up from bootloader and then measures each component's integrity, verifying its code signature one by one. This process will build the initial trust environment. As Figure 1 shows, the establishment of static trust chain can be divided into 4 stages.

(1) The device identity key and encryption key are derived from SRAM PUF fingerprint by fuzzy extraction algorithm after the device starts up. Then device measures integrity of bootloader and verifies measurement values according to its code certificate.

(2) The bootloader measures the Secure OS, verifies its image signature by certificate and then extends the integrity measurement event to system log. If the signature verification fails, Secure OS will halt until the image is repaired. Then Secure OS starts trust services such as algorithms of Measure, Attest, and Code_upgrade, which are executed in TEE protected by ARM TrustZone.

(3) Secure OS measures the Normal OS and verifies its image signature by OS certificate, so that the trusted chain will be established from hardware PUF, bootloader, Secure OS to Normal OS.

(4) Normal OS measures software integrity and extends measurement event to system log. Due to the variety of subs, integrity verification is not mandatory. Multiple software will simultaneously run in unprotected normal world, and virus, worm, etc. can intrude the system by exploiting system vulnerabilities. Consequently, it is extremely important to protect runtime integrity of software.

The core algorithms to protect softwares runtime integrity are provided by trusted service modules located in an isolated execution environment based on TrustZone. The main algorithms include the following.

3.1.1. Measure. It measures the code fragments of function level at runtime and returns the measurement results to attestation service and other trusted services. The SHA1 hash algorithm is used to calculate the digest of code fragments. The measurement algorithm is activated before the code fragments execute at runtime.

3.1.2. Attest. It reports the proofs on information flow integrity and codes integrity, signs the proofs by device identity key, and sends the attestation data to the remote verifier to prove its trustworthiness. The device identity key is derived from hardware PUF fingerprint by fuzzy extraction algorithm.

3.1.3. Code Upgrade. It is the trusted upgrade algorithm of firmware and software in IoT device. The certificate cert (pk_S) is used to verify the code signature on distributed software from administrator server. The monotonic counter c is used to prevent replay attack and version rollback attack on software distribution. When the software version is upgraded, the counter will increase one. The device upgrades software according to codes binding with device key derived from PUF. The invalid codes will be rejected during software upgrading.

3.2. Runtime Integrity Protection. The integrity check ensures security of the initial environment before software runs. However, attackers can breach software's runtime integrity by exploiting vulnerabilities to inject malicious code or reuse the system codes. For example, ROP reuses the short instruction sequences (called gadgets) that end with ret instruction, overwrites the stack return address with gadgets address, and changes the execution flow of programs.

We utilize the memory security mechanism DEP to prevent illegal execution by injecting malicious codes in stack and data segments. Hackers can inject malicious codes in memory pages such as stack, heap, and memory pool marked as nonexecutable, but these injected codes are prevented from executing by DEP. The method of runtime integrity protection is associated with DEP in TEE architecture in our scheme. It mainly includes three aspects: (1) runtime integrity of code block; (2) correctness of code return address; and (3) validity checking of indirect jump address of code. The integrity protection on code block and return address is implemented implicitly by decrypting return memory address (cf. Figure 2).

3.2.1. Software Code Runtime Measurement. The general method of measuring codes is static measurement in IMA [26] when the executable codes are loading into the memory, which effectively guarantees codes integrity at loadtime. We are expected to protect codes runtime integrity further. The software is divided into function blocks denoted by $fn\_\text{block}[i]$ according to the granularity of function slices. Before the function is called during program execution, code block of this function must be measured at runtime, and its integrity is checked indirectly when function return address is decrypted.

The runtime measurement checks integrity of each code block, so it can prevent dynamic-link library hijacking to change the function invocation. The measurement occurs at the time of function calling. Obviously, it reduces the interval between check and execution of code integrity and lowers the security risk of TOCTOU attack. The frequent triggers of runtime measurement may affect performance in the general terminal platform like PCs and smartphones, but it is suitable for IoT application of fixed and fewer codes. We consider the measurement engine must be located in a secure execution environment based on TrustZone, so hackers cannot attack the measurement engine and disable or bypass the runtime measurement.

The measurement engine must trace the function calling with software code execution (Algorithm 1). The detailed measurement algorithm is shown as follows.

3.2.2. Function Return Address Protection. The code reuse attack like ROP cannot tamper the software code itself, but it modifies the function return address in stack and jumps to the existing gadgets in system libraries ended with ret instruction. In order to ensure the runtime integrity, we design the encryption of return address to protect codes correct call and return and prevent illegal jump to ROP gadgets. The encryption does not protect the integrity of code instructions in memory code segment, but ensures the correctness of function return address. Any exceptional modification on encrypted return address in stack must cause invalid function return at final.

The encryption on return address follows $\text{retaddr}'_j = \text{Enc}_K(\text{retaddr}_j, m_j)$, where K is the device encryption key derived from PUF, and m_j is the measurement value of code block on function f_j. We recommend the AES encryption algorithm to protect return address, which has enough security strength to resist differential attack. In a word, the return address is bound with function measurement and encrypted with device PUF key.

In our scheme, measurement engine measures the calling target function, and the address checking module encrypts or decrypts the return address. Figure 2 describes the encryption process on return address, and decryption is a similar process in reverse. Its concrete method is given below:

(1) When function calling occurs, measurement engine hooks call and pushes operation on function calling, e.g., $bl\,0x11524$, push $\{fp, lr\}$ in ARM devices. It passes binary instructions currently executed on code segment to address checking module.

(2) Address checking module parses the return address of function calling, uses the device key to encrypt the address, and then pushes the encrypted address into stack.

(3) The function executes normally until the function returns. The data except the return address have no change on stack, and only the function return address is encrypted. The attacker can modify all the data on stack for code reuse attack.

(4) Measure engine hooks pop operation before function return, e.g., pop$\{fp, pc\}$, and then passes binary instruction currently executed to address checking module.

(5) Address checking module fetches the encrypted address, and decrypts it with device key. The original return address is popped to the instruction pointer register. Finally the function returns as usual.

We should note that the first function block 0 is not necessary to return. Therefore, the first function is only measured, but not encrypted. Because the return address in encrypted with measurement of function codes, the decryption implies the check of function measurement. The measurement must be correct; otherwise, the decryption may cause memory crash because of an invalid return address.

3.2.3. Indirect Jump Address Checking. The address jump of function calling is protected by return address encryption in the above subsection, but the indirect jump may violate control flow integrity inside function, which can cause JOP attack. So we use the address checking policy to prevent illegal indirect jump with the support of information flow integrity. Measurement engine hooks the indirect jump instruction, and check address module verifies the target address based on function calling table in TEE. If it conforms to address checking policy, normal jump is allowed; otherwise, exceptional jump is rejected.

Figure 3 depicts the address checking process according to security policy. Take IoT devices of ARM instruction set for example, measurement engine hooks jump instruction such as b, bl, bx, blx, and bxj and passes the binary

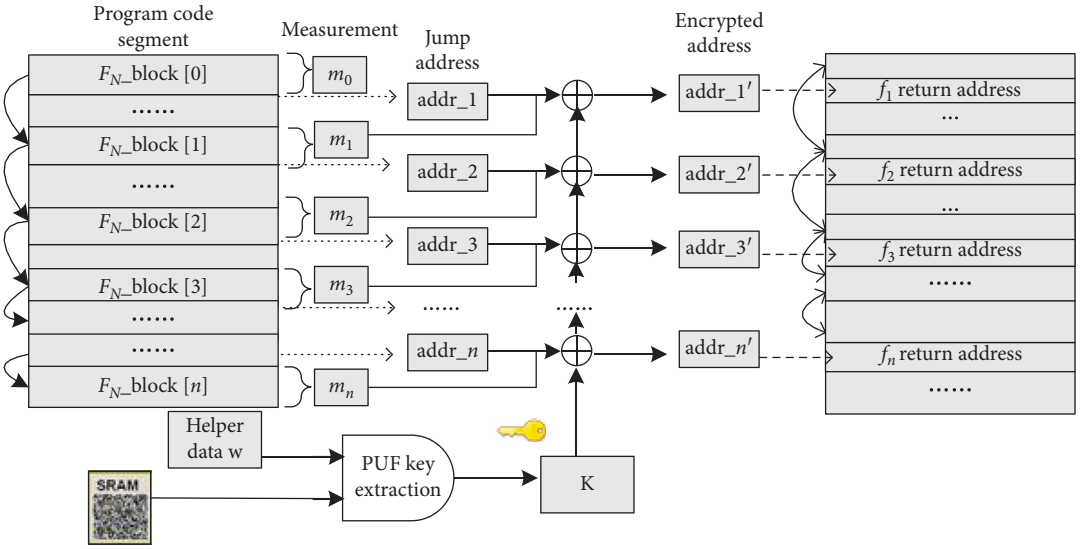

Figure 2: The encryption and decryption method on program return address.

Input: code blocks of software: $fn_b\text{lock}[j]$, $j = 0, \ldots, n$
Output: measurement results and log: χ, log
(1) The code blocks are loaded into memory with each block's virtual address ranged from $fn_b\text{lock}[j]$.start to $fn_b\text{lock}[j]$.end;
(2) Device X measures SW at runtime.
(3) (1) Initialize SW measurement log as log: ={}, aggregated measurement fingerprint as χ: = 0;
(4) 2) The first function: $f_0 = fn\_\text{block}[0]$. Device reads and measures it, s_0: = Read(SW, f_0.start, f_0.end), m_0: = HASH(s_0);
(5) (3) The measurement is recursively triggered with function calling. And its algorithm measure(f_0)
(6) **for** f in function_call_table(f_0) **do**
(7) (a) Reads code block s_j of function f, s_j: = Read(SW, f.start, f.end), measures it and obtain fingerprint m_j: = HASH(s_j);
(8) b) Record measurement log for s_j, log: = log $\cup \{(\text{desc}_j, m_j)\}$, where desc_j is the description information of code block, e.g. function name, type;
(9) (c) Aggregate fingerprints for SW, χ: = HASH($m_j \| \chi$);
(10) (d) measure(f);
(11) **end**
(12) (4) Device X stores χ, log of every code block execution for future attestation.
(13) **return** χ, log

Algorithm 1: Measurement algorithm of software codes.

instruction to check address module (please refer to return address protection above on checking process of return instruction). Then check address module parses the target address from the jump instruction and checks whether the jump target address is limited in function scope according to function calling table. If target address is inside the function, jump instruction is allowed; otherwise, check address module determines whether the target address is not the entry address of other function. If yes, jump instruction is allowed, and if no, it is refused because the target address may be the codes of JOP gadget.

3.3. Software Package Distribution. The instrumented software codes cannot be executed directly on device in our scheme, so a centralized server must rewrite software binary code bound with assigned device and distribute software to the device. In stub distribution, the server must authenticate the device identity and ensure the trustworthiness of distributed software code to IoT device. The security mechanism such as integrity checksum and code signature will be applied to software code distribution. As for software's OTA (over the air) update or incremental update, they are not considered in our scheme beyond the security. Identity keys of server, sample device, and target device are predeployed before software distribution. In our system, only authorized device or device group can install distributed software from server, and the attacker cannot forge software codes from compromised device or rollback software version to cheat device through the distribution protocol. After software installation, secure boot ensures the loading and execution of a correct and trustworthy software.

The distribution protocol participants include adminstration server, sample device (called initiator), and target device.

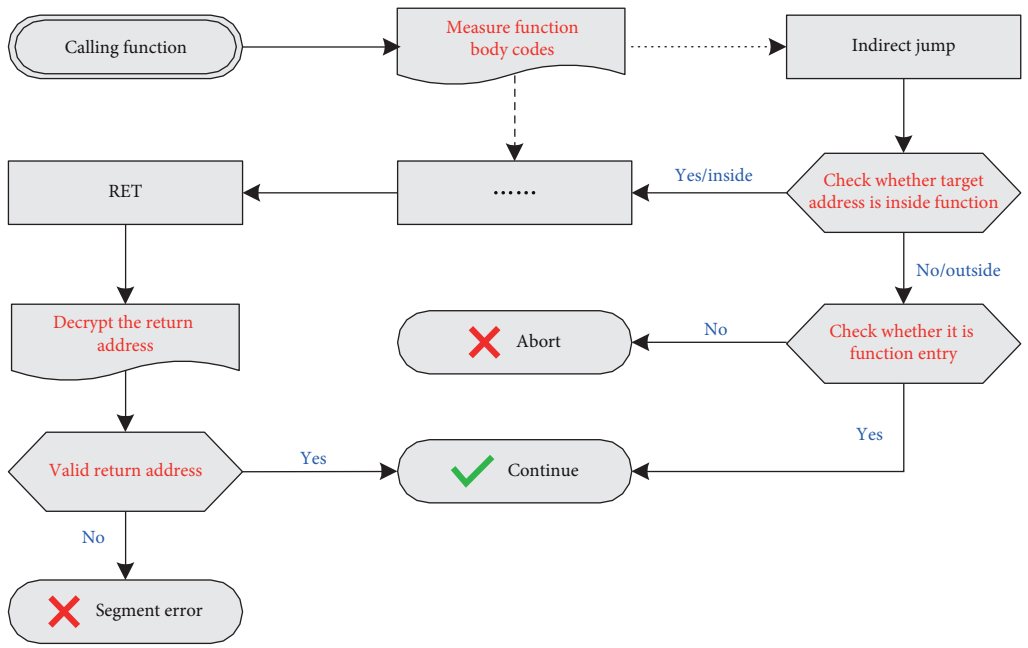

Figure 3: The process to determine validity of function return address.

Server is responsible for releasing software package by rewriting binary codes according to device key. It verifies devices identity and distributes trusted software stub to device. Initiator may be a simulation device or a real device in IoT network. It assists server to extract sample software and ensures the subs integrity and trustworthiness. Target device is the target platform on which distributed software is installed and executed. Figure 4 shows whole distribution protocol, which can be divided into 2 sub protocols: extraction and distribution.

3.3.1. Extraction

(1) Initiator executes the sample software and measures and attests its integrity. Initiator signs the measurement aggregation of the software with device key derived from PUF. The signature is $\text{sig} = \text{HMAC}(K, \text{Hash}(\chi \| \text{nonce} \| t_i \| v))$.

(2) Initiator sends integrity proof and signature (sig, log) to server with for device attestation. Server verifies the signature and measurement according to log.

(3) Server rewrites software using device key K of initiator, that is to say, Server encrypts the software return address with device key and measurement. Server generates the signature for the rewritten software and returns software package with signature to Initiator.

(4) Initiator verifies software integrity by servers signature, installs or upgrades the distributed sub, and runs testing software correctness.

3.3.2. Distribution

(1) Target device encrypts its device key with Server's public key and requests software's upgrade or install with the current counter

(2) Server decrypts device's key and rewrites new software based on it. Then Server signs the software and device counter by its private key and distribute software with its signature.

(3) Device verifies the nonce, counter, and signature. Then if verification is passed, Device upgrades software, and also increases counter.

From the above protocol figure, we consider that software integrity is ensured by servers' signature, and the rewritten software are different from each device because of different PUF device keys so that the distribution protocol mainly focuses on distributing same version software but different code copy to all IoT devices. Device identity must be verified before server distributes software for device. In order to prevent the rollback of old version software, devices must verify the counter bounded with software before installing or upgrading.

4. Security Evaluation

4.1. Bootkit and Firmware Tampering. The illegal firmware tampers the correct one on the device and injects malicious codes at the system initialization. This is the basic security problem in IoT devices. The Bootkit is a kernel-mode rootkit variant at boot stage, and it may copy, substitute, and compress the memory containing malicious codes so as to bypass checksum verification in system secure boot. The lightweight trust establishment is employed to achieve secure boot from PUF hardware. The attackers cannot intrude the trust root in secure world of TrustZone, so they cannot bypass verification on bootloader, os kernel, and other softwares, which is efficient to prevent Bootkit.

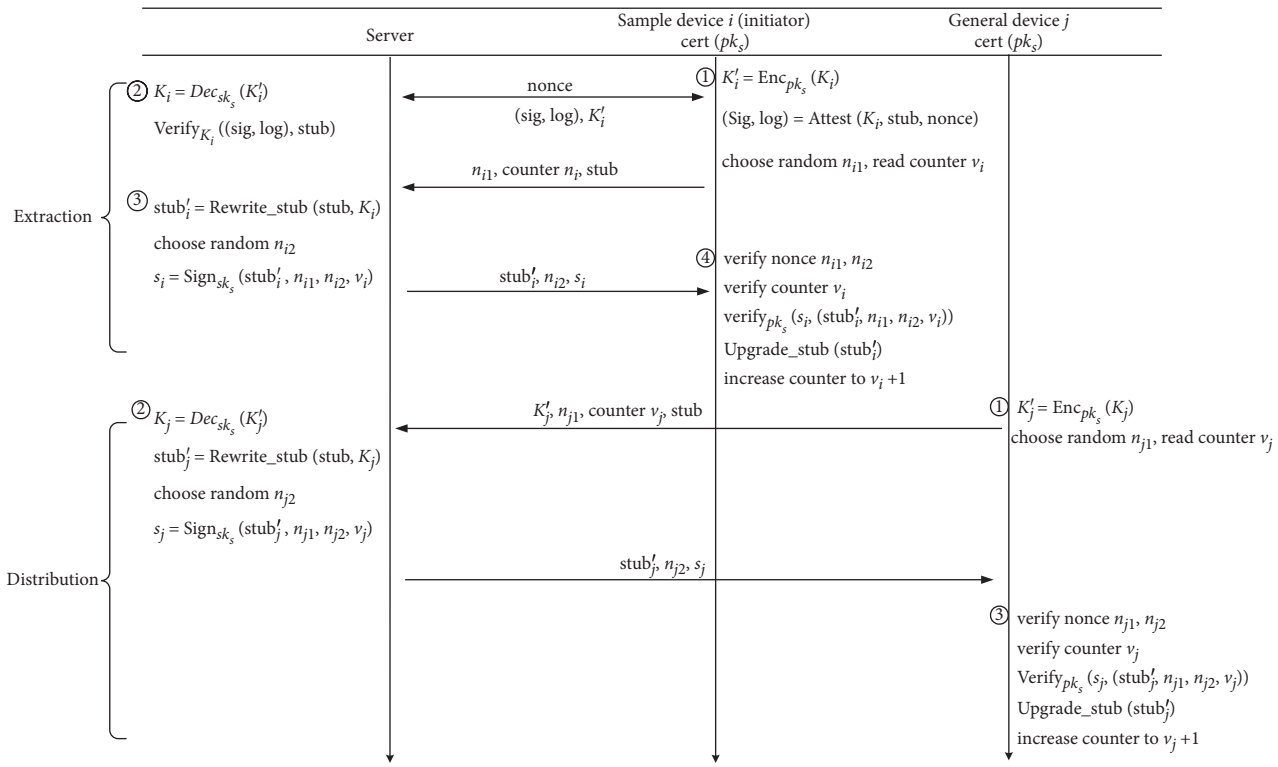

Figure 4: Distribution protocol for device software in IoT.

4.2. Code Reuse Attack. The return address encryption is implemented to prevent code reuse attack like ROP and JOP in our scheme. As DEP is enabled, our system will reject code injection. But CRA modifies the return address in stack, jump to malicious gadgets in system library, and executes the malicious behaviors. For example, ROP gadget is a piece of code located in the existing program and shared library code, which typically ends in a return instruction. The return address is encrypted with checksum of function codes by PUF key in our system. Therefore, any tampering of return address will be checked while decrypting return address, and CRA can be prevented by address checking modules resided in TrustZone secure world.

4.3. Dynamic-Link Library Hijacking. The attacker may exploit system vulnerabilities to tamper the dynamic-link library (dll) as a system service. Though the application software is healthy, it triggers the execution of malicious codes when invoking dynamic-link library hijacked by attacker. In our scheme, measurement engine (ME) will measure the target function defined in dll while control flow transfers to dll and verify the integrity of target function. If it is the same one according to the digest of function codes, the dll's invocation will be terminated immediately. Owing to the enforcement in function level, the dynamic measurement and verification can prevent this kind of attack efficiently in our scheme.

4.4. TOCTOU Attack. TOCTOU attack is a time interval attack to bypass security checking mechanism. The threat is relatively high under the traditional static measurement and checking at software loading time. Our dynamic measurement can reduce the security risk significantly through shortening the time interval between measurement and checksum. Although periodic monitor of codes' integrity can mitigate this attack, the performance will be obviously affected. The function codes are checked while it is called; thus, TOCTOU attack is prevented with high probabilities by measuring the integrity of the current running codes at runtime.

4.5. Network Replay & MITM Attack. The software distribution protocol uses monotonic counter to prevent replay attack and version rollback attack. Besides counters, the devices and server both use fresh random number and message signature to ensure message integrity and identity authentication. These security mechanisms can efficiently prevent resist replay, man-in-the-middle, and impersonation attacks.

4.6. Device Compromising Attack. The same hardware and software configuration are deployed commonly in IoT network, especially in swarm devices. If one device is affected by worm and virus, other devices will be attacked soon, so that the hackers will intrude and control whole IoT network. What is worse, the attackers will manipulate all device nodes to conduct large-scale botnet attack. In our scheme, every device has a unique PUF key and the instrumentation code on each device has difference on the return address. Consequently, the other software is immune to the same kind of

attack without knowing its device PUF key, although one device is compromised and its PUF key is exposed to attacker. The attacker can obtain the function measurement and original return address from the compromised device, but he cannot crack software return address on other device without device PUF key and construct the dedicated shell codes for the code injection, buffer overflow, and code reuse attack like ROP and JOP. This efficiently reduces security risk for the reason that the same kind of software attack is spread to IoT network by compromised device.

5. Implementation and Performance Evaluation

Following the above method, we implement RIPTE prototype based on lightweight trust architecture. In this section, a simple example is given out to illustrate the implementation principle; then, we summarize the performance results from the experiments of test evaluation.

5.1. System Implementation. The original software must be rewritten to target one on specified IoT device by network administrator. After that, the program can run with integrity protection on the target device. First of all, the device registers its valid identity with device key derived from PUF and server allows the device to join the IoT network. Then administrator prepares the original software to instrument target one by rewriting tools. The tools analyze binary codes, especially function calling in program, add the codes of measurement function, and insert hook modules with hook instruction and encrytion/decrytion. When target program executes, the function return address will be encrypted with the device key and measurement value. Moreover, considering the binding relationship between target software and IoT device, instrumented software can only be executed correctly on the specified device. Finally the instrumented programs are distributed to designated device and installed on it. As a result, the target program can execute on the specified device with integrity protection at runtime, preventing the code tampering, and code reuse attack.

Figure 5 illustrates the secure execution principle of a simple program within our scheme in ARM platform. The control flow of original binary codes may be hijacked by codes tampering or code reuse attack like ROP and JOP. The function return address is in the form of plaintext in the stack at program runtime, such as "00032468." Instrumented binary codes by rewriting tools has 2 kinds of hook functions (entry hook and exit hook), which intercept the special instructions about function jumping, stack push/pop. In the simple example, entry hook is function hook_*bl* and exit hook is hook_pop_*fp_pc*. Entry hook first measures codes integrity of called function, then uses device key to encrypt function return address, next outputs encrypted address to lr register, and pushes it in stack. At the moment, the function return address is in the form of ciphertext in the stack, such as "*A*1*B*1*E*7*BC*." Exit hook first pops the cihpertext of return address to *lr* register and decrypts *lr* with function measurement using device key and then outputs plaintext of function return address to *lr*. After that, the regular return will be transferred without any change.

5.1.1. Considerations in Implementation. The choice of encryption algorithm is a tradeoff between security and efficiency. We implement AES as encryption algorithms in our system. And we also test two other candidate algorithms XOR and RC4, respectively. XOR is just a simple and efficient algorithm in address encryption, which requires less power and fewer codes. Its problem is that the encryption strength is not high and the key may be analyzed by differential attack. RC5 can work well as lightweight algorithm in wireless sensor network, which generally performs on 16/32/64 bits messages. But RC5 is also vulnerable to differential attack. AES is a usual algorithm in IoT application, especially built-in implementation in COAP application layer. The hardware accelerations of AES algorithm are supported in various CPU like Intel AES-NI instruction set. So we recommend AES to implement our scheme. In general, the specified application scenarios and security requirements determine the concrete algorithm choice in IoT network.

The function return address on stack is 32 bits in ARM 32 bits platform. Since enough security strength is provided in our scheme, we implement address encryption by 128 bits key with the algorithms of AES and RC5. The push operation on stack limited the ciphertext lengths, so we consider alternative method which truncates the address ciphertext into two parts. The first part is 32 bits pushed on stack, and the remainder with size of 96 bits is stored in shared memories as a lookup table, which index is first part's contents. While function is called, the plaintext of function return address is encrypted, then the first part of ciphertext is pushed into stack, and the remainder of ciphertext is stored into a hash table. While called function returns, address checking module fetches the ciphertext on stack, lookups the whole ciphpertext of return address in a hash table, decrypts it, and pops the correct address to the instruction pointer register.

The performance can be greatly influenced because of frequent encryption/decryption during function calling and returning, such that we consider the optimization calculation by precomputing encryption address to improve performance. The IoT software is relatively simple, in which the target jumping address can be easily deduced before instrumenting codes. The function measurement is specific value, and the encryption key is known to server in safe mode before server rewrites software codes. The server can precompute encryption return address according to above definite values and store it into a function address table. While the program is executed on the device, checking address module does not need to encrypt the return address and directly search the ciphertext of return address from the function address table. It pushes the ciphertext on the stack when entering function. Furthermore, the decryption of return address can be similarly done in function address table according to the method of encryption. Through analyzing the sample programs in an embedded system, we find out that function pointers are seldom used in program, so that almost all the return address can be precomputed

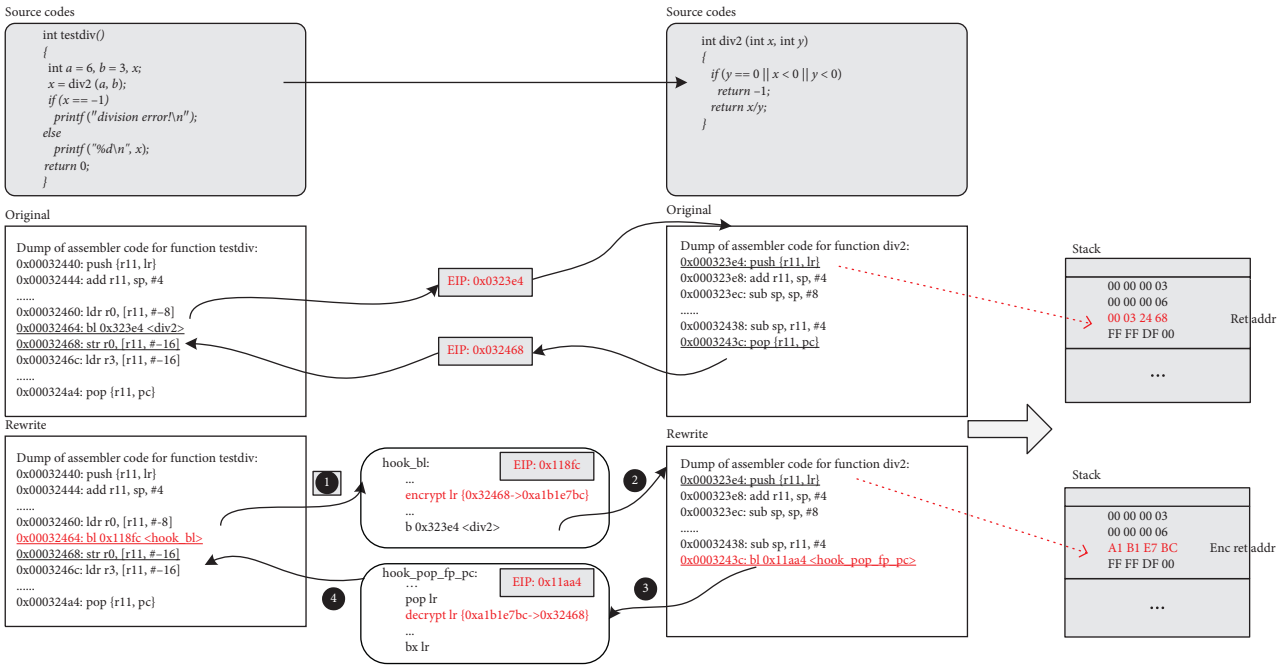

Figure 5: Control flow transferring in simple example with/without RIPTE.

before instrumenting codes. It greatly improves the performance while precomputing on address encryption and decryption, especially in the program of frequent function calling.

5.2. Performance Evaluation. In order to evaluate the performance of our RIPTE prototype implementation (source code is available at https://github.com/mars208/RIPTE), we choose the benchmark suite MiBench [27] as overall performance for commercial embedded softwares and cube algorithm in basicmath of MiBench as single and loop performance test.

All benchmarks are executed on embedded development boards HiKey, CPU ARM Cortex-A53 1.2 GHz with 8 Cores, SOC HiSilicon Kirin 620, RAM 2 GB. The code size of the main modules in our system are summarized in Table 1, which indicates that just a few codes (1075 LOC) newly added are instrumented to protect software except PUF and secure boot (2170 LOC). PUF and secure boot are the basic building blocks executing once to establish trust while the system starts up and the total size of increasing TCB is 313.8 kB. We choose SRAM PUF to develop terminal system which meets robustness and uniqueness the most important properties of PUF. We use the noise present in the SRAM start-up value to accumulate entropy for the secure random seed. For FE.Gen and FE.Rec operations on PUF, we adopt the code-offset mechanism using BCH code [28] based encryption/decryption. These properties guarantee that errors between the generate and reproduce procedures of PUF can be corrected by the BCH code; thus, the device key can be correctly derived from PUF key extraction. Table 2 shows the scale of test samples on an embedded device and performance comparison with/without code instrumentation. The time cost per function is less than 1 ms under the stress test of more than 100 thousands function calling. And the main performance cost is the overhead of trustzone context switch, so that it has less performance impact under condition of great security improvement.

Our system implements AES algorithm and also tests the other two encryption algorithms, XOR and RC4. Their performance evaluation is shown in Figure 6 on decryption procedure. The encryption of the return address is finished by the administrator before software distribution; thus, only the decryption may influence the software's runtime performance. The test results show that XOR algorithm is the most efficient among three algorithms, RC4 comes second, and AES costs 1.628 s in 10000 function callings. But AES is more secure than other two algorithms, and the cost of AES decryption is not expensive in high-end IoT device.

Precomputing is the most important method to optimize runtime performance in our implementation. The spending time is 86.2 ms with precomputation in 100 function calling, as shown in Figure 7. Most of the function calling can be predicated according to CFG (control flow graph), particularly in IoT software, so ACM (Address Checking Module) in our system can precompute the return address of target function. The average performance improvement is 24.5% by precomputing optimization from statistics of experiment results. Figure 8 shows performance of encryption, decryption, and measurement in multiple loop testing. With the increase of the loop time, the performance of encryption and decryption has minimal time growth, almost little change. But the performance of measurement becomes linear increase. The more the functions are calling, the more the measurement time is spent during runtime execution.

Table 1: Module size in RIPTE implementation.

Module	LOC	Binary size (kB)
HOOK	905	5.7
Encryption	25	58.9
Decryption	58	60.9
ME	12	58.6
ACM	75	95.2
PUF	1747	10.7
Secure boot	423	23.8
Total	3245	313.8

Table 2: Performance comparison with/without instrumentation on different types of embedded programs.

Category	Program	Func	Func call	Binsize (kB)	Org time/func (ms)	Instr time/func (ms)
Automotive	Basicmath	5	140481	12.5	0.088881	0.723067
Consumer	JPEG	552	18017 1	232.4	0.002962	0.849559
Network	Dijkstra	6	273610	7.1	0.031748	0.557701
Office	Stringsearch	10	2781	44.3	0.014717	0.137495
Security	Sha	8	111681	5.2	0.015941	0.518042
Telecomm	crc32	3	3	3	2172.80	2774.55

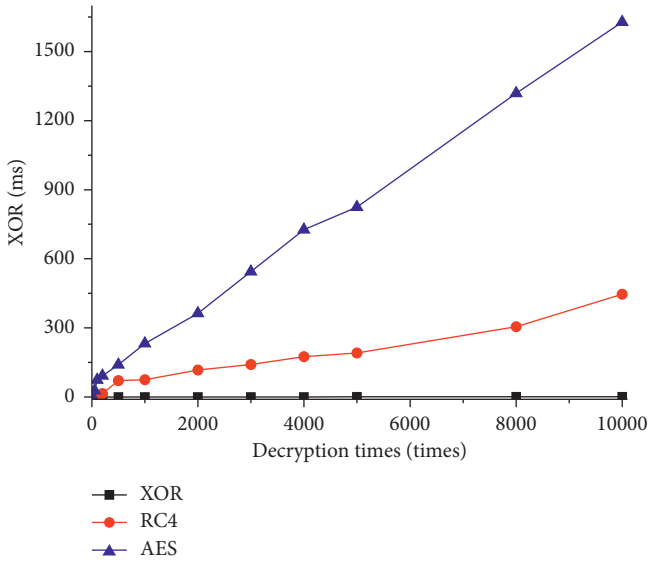

Figure 6: Decryption time for different algorithms.

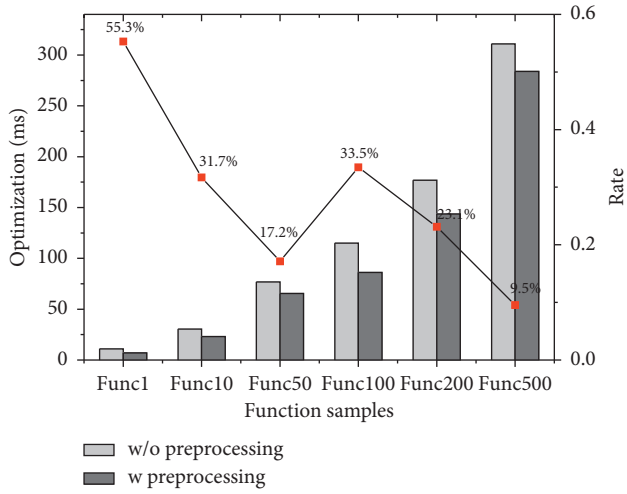

Figure 7: Performance optimization with precomputation.

6. Related Work

6.1. Integrity Measurement. The most typical scheme of integrity protection is IMA [26] (Integrity Measurement Architecture) in Trusted Computing, which uses TPM to measure integrity of kernel and executable memory images at loading time. PRIMA [29] scheme is an improvement combining the static measurement with access control model on information flow, and it reduces measurement range and promotes the efficiency. Afterwards, schemes of LKIM, HIMA, HyperSentry, etc. ([30~33]) were presented to extend IMA to other application systems. In brief, these schemes are static integrity measurement, and it is not suitable to protect the runtime integrity for a single application in IoT device.

6.2. Software Attestation. Software attestation is a well-known and popular technique to verify security states of IoT devices under the resource constraints [34]. It prevents the attacks like software tampering and malicious code injection by verifying software checksum. The concept of software attestation was firstly presented in SWATT [35] for embedded system, i.e., checksum on the whole memory. The typical scheme in early study is CMU's Pioneer [36], in which the remote verifier can check software integrity of embedded system by challenge-response protocol on software attestation. Following the thought of Pioneer, many software attestation schemes ([37~39]) were designed based

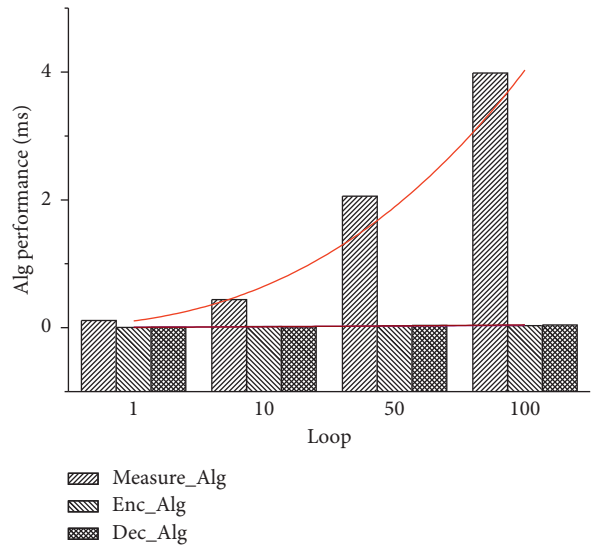

Figure 8: Performance on measurement, encryption, decryption.

on different constructions of checksum function, in which designer considered the factors such as pseudorandom memory traversal, antireplay attack, memory detection with minor change, and low network delay. Software attestation can be categorized into time-based attestation and memory-based attestation as usual [40]. The former checks response timing to identify compromised software; the later fills memory with randomness to prevent malicious software in the attestation. The software attestation for peripherals was presented to prove the firmware integrity of peripheral in 2010 [41], which can prevent malicious code injection during firmware update. Asokan et al. proposed SEDA scheme [42] for IoT cluster devices and constructed attestation tree from the initial device to check all cluster devices in IoT network. Asokan et al. [43] designed and implemented a control flow attestation for embedded device through logging branch path of simple program execution. The scheme overcomes the shortages of static measurement and attestation, which can efficiently prevent control flow hijacking attacks.

With deep development on software attestation, the researchers found out some disadvantages of embedded software attestation. For example, many schemes of software attestation cannot prevent code substitution and ROP attack [44, 45]; time-based attestation cannot resist the TOCTOU (Time of Check, Time of Use) attack [46]. Therefore, we attempt to solve these security issues of software attestation from the point of code integrity protection based on trust establishment.

6.3. Control Flow Integrity. One key challenge is the vulnerability of IoT devices to malware. Hacker often exploits buffer overflow vulnerability to inject malicious code, while simultaneously rewriting the function return address and jump the entry address of injected codes [47, 48]. Control flow integrity [49] is an important security method to guarantee software trust execution in IoT device. The key idea of CFI is to enforce the control flow at runtime according to control flow graph (CFG), which includes hardware-based CFI and software-based CFI. Many CFI research works follow the ID-based scheme presented by Abadi et al. [49]. And its method assigns a label to each indirect control flow transfer or potential target in the program. G-Free [50] is a compiler-based approach to eliminate ROP attack by protecting free-branch instructions in binary executable. G-Free can be applied in compiling GNU libc and many real-world applications. Recent work from Google [51] and Microsoft [52] has moved beyond the ID-based schemes to optimize set checks. Data execution prevention (DEP) assisting CFI can efficiently prevent codes from injection attack for stack or data ([53, 54]). The research trend of CFI [55] is to improve analysis and enforcement algorithms and to extend protection scope such as just-in-time code, os kernel code. In embedded system, CFI can be applied to prevent code injection attack and code reuse attack like ROP and JOP. Because IoT environment has small amount of code, the program can easily be examined by CFG. But CFI enforcement certainly affects system performance owing to its limited resource.

6.4. Secure Code Updates. Besides protection of software runtime execution, secure code update is the important aspect in software trust execution and software attestation in IoT. Secure code update in an embedded system is referred to software update, firmware update, over the air update (OTA), software attestation, and so on. It must ensure the integrity and authenticity of code updates before installation and prove the integrity of newly installed software on IoT device. PoSE-based (Proofs of Secure Erasure) methods ([39, 56, 57]) are applied widely in commodity devices, with the strong assumption that attacker cannot communicate with IoT device during software attestation. Perito and Tsudik [40] used PoSE to propose a practical approach to secure erasure and code update in embedded devices and evaluated the scheme's feasibility in commodity sensors. Kohnhauser and Katzenbeisser [58] presented a code update scheme which verifier can enforce the correct distribution and installation of code updates on all devices in the network. Especially the scheme allowed the administrator to provide secure code updates for a group of devices.

7. Conclusion and Outlook

The IoT devices currently have many known and unknown vulnerabilities, which cause a large-scale swarm, botnet, and DDoS attack targeting IoT. We propose RIPTE, a novel and practical scheme for software trusted execution in IoT device relying on secure hardware PUF and trust architecture. The lightweight trust is established by PUF and secure boot. Software codes and IoT device are bound together to prevent software tampering and version rollback. It eliminates dynamic-link library hijacking and TOCTOU attack by runtime measurement in granularity of function. Encryption protection of return address prevents the code reuse attack like ROP and JOP. The test results of our prototype indicate that the runtime execution protection is efficiently applied to

IoT software. The hardware cryptographic support is considered to enhance our scheme in future work. In our scheme, TEE with TrustZone is utilized to assure trust service executing in secure world with high isolation security. The TEE context switch between normal world and secure world is the important performance factor, so we will research the architecture and performance optimization as the future work.

Data Availability

The data used to support the findings of this study are included within the article.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

The research presented in this paper was supported by National Key R&D Program of China (2018YFB0904903 and 2020YFE0200600) and National Natural Science Foundation of China (Nos. 61872343 and 61802375).

References

[1] K. E. Defrawy, D. Perito, and G. Tsudik, "SMART: secure and minimal architecture for (establishing a dynamic) root of trust," in *Proceeding of Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, 2012.

[2] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: a security architecture for tiny embedded devices," in *Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*, Amsterdam, Netherlands, 2014.

[3] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: tiny trust anchor for tiny devices," in *Proceedings of the 52nd Annual Design Automation Conference (DAC '15)*, San Francisco, CA, USA, 2015.

[4] A. R. Sadeghi and D. Naccache, *Towards Hardware-Intrinsic Security*, Springer, Berlin, Germany, 2010.

[5] U. Ruhrmair, S. Frank, S. Jan et al., "Modeling attacks on physical unclonable functions," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, USA, 2010.

[6] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "FPGA intrinsic PUFS and their use for IP protection," in *Proceedings of the 2007 Cryptographic Hardware and Embedded Systems-CHES 2007*, Vienna, Austria, September 2007.

[7] D. E. Holcomb, W. P. Burleson, and K. Fu, "Power-up SRAM state as an identifying fingerprint and source of true random numbers," *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1198–1210, 2009.

[8] R. Pappu, B. Recht, J. Taylor, and N. Gershenfeld, "Physical one-way functions," *Science*, vol. 297, no. 5589, pp. 2026–2030, 2002.

[9] G. E. Suh and S. Devadas, "Physical unclonable functions for device authentication and secret key generation," in *Proceedings of the 44th Annual Design Automation Conference*, San Diego, CA, USA, 2007.

[10] Y. Wang, W.-k. Yu, S. Wu, G. Malysa, G. E. Suh, and E. C. Kan, "Flash memory for ubiquitous hardware security functions: true random number generation and device fingerprints," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP)*, Washingtom, DC, USA, 2012.

[11] J. P. Linnartz and P. Tuyls, "New shielding functions to enhance privacy and prevent misuse of biometric templates," in *Audio-and Video-Based Biometric Person Authentication*, pp. 393–402, Springer, Berlin, Germany, 2003.

[12] S. Zhao, Q. Zhang, G. Hu, Yu Qin, and D. Feng, "Providing root of trust for ARM trustzone using on-chip SRAM," in *Proceedings of the 4th International Workshop on Trustworthy Embedded Devices*, Scottsdale, AZ, USA, 2014.

[13] ARM, "TrustZone technology," 2020, http://www.arm.com/products/security-on-arm/trustzone.

[14] Samsung, "White paper: an overview of Samsung KNOX," 2013, http://www.samsung.com/global/business/business-images/resource/white-paper/2013/06/Samsung_KNOX_whitepaper_June-0.pdf.

[15] Intel, "Trusted execution technology," 2018, http://www.intel.com/technology/security/downloads/TrustedExec_Overview.pdf.

[16] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: an execution infrastructure for TCB minimization," in *Proceedings of the 3rd ACM European Conference on Computer Systems*, Glasgow, Scotland, 2008.

[17] A. Seshadri, M. Luk, N. Qu, and A. Perrig, "SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 335–350, 2007.

[18] J. M. McCune, Y. Li, N. Qu et al., "TrustVisor: efficient TCB reduction and attestation," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2010.

[19] P. Bryan, J. M. McCune, and P. Adrian, "Bootstrapping trust in commodity computers," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy (SP '10)*, Oakland, CA, USA, 2010.

[20] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune, "Trustworthy execution on mobile devices: what security properties can my mobile platform give me?" in *Trust and Trustworthy Computing*Vol. 7344, Springer, Berlin, Germany, 2012.

[21] M. Schunter, "Intel software guard extensions: introduction and open research challenges," in *Proceedings of the 2016 ACM Workshop on Software Protection (SPRO '16)*, Vienna, Austria, 2016.

[22] McK. Frank, I. Alexandrovich, I. Anati et al., "Intel software guard extensions (Intel? SGX) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016 (HASP 2016)*, Seoul, Republic of Korea, 2016.

[23] L. Richter, J. G. Tzfried, and T. Mller, "Isolating operating system components with Intel SGX," in *Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX '16)*, Trento, Italy, 2016.

[24] S. W. Smith, *Trusted Computing Platforms—Design and Applications*, Springer, Berlin, Germany, 2005.

[25] L. Chen, *NIST Special Publication 800-108: Recommendation for Key Derivation Using Pseudorandom Functions*, Computer Security Division. Information Technology Laboratory, NIST, Gaithersburg, MD, USA, 2009.

[26] R. Sailer, X. Zhang, T. Jaeger et al., "Design and implementation of a TCG-based integrity measurement architecture," in *Proceedings of USENIX Security '04*, pp. 223–238, San Francisco, CA, USA, 2004.

[27] M. R. Guthaus, J. S. Ringenberg, D. Ernst et al., "MiBench: a free, commercially representative embedded benchmark suite," in *Proceedings of the 2002 IEEE International Workshop on Workload Characterization*, Austin, TX, USA, 2002.

[28] Y. Dodis, L. Reyzin, and A. Smith, "Fuzzy extractors: how to generate strong keys from biometrics and other noise data," *SIAM Journal on Computing*, vol. 38, no. 1, 2008.

[29] T. Jaeger, R. Sailer, and U. Shankar, "PRIMA: policy-reduced integrity measurement architecture," in *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies*, pp. 19–28, Lake Tahoe, CA, USA, 2006.

[30] Z. Xu, Y. He, and L. Deng, "An integrity assurance mechanism for run-time programs," in *Proceedings of the 2009 of Information Security and Cryptology*, pp. 389–405, Ankara, Turkey, 2009.

[31] P. A. Loscocco, P. W. Wilson, J. A. Pendergrass et al., "Linux kernel integrity measurement using contextual inspection," in *Proceedings of the 2nd ACM Workshop on Scalable Trusted Computing*, New York, NY, USA, 2007.

[32] A. M. Azab, P. Ning, E. C. Sezer, and X. Zhang, "HIMA: a hypervisor-based integrity measurement agent," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, Washington, DC, USA, 2009.

[33] A. M. Azab, N. Peng, Z. Wang et al., "HyperSentry: enabling stealthy in-context measurement of hypervisor integrity," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, Chicago, IL, USA, 2010.

[34] T. Abera, N. Asokan, L. Davi et al., "Invited—things, trouble, trust: on building trust in IoT systems," in *Procceeding of Design Automation Conference 2016*, Austin, TX, USA, 2016.

[35] A. Seshadri, A. Perrig, L. Van Doorn et al., "SWATT: softWare-based attestation for embedded devices," in *Proceedings of 2004 IEEE Symposium on Security and Privacy*, Berkeley, CA, USA, 2004.

[36] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. Van Doorn, and P. Khosla, "Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems," *ACM SIGOPS Operating Systems Review*, vol. 39, no. 5, 2005.

[37] Y. Li, J. M. McCune, and A. Perrig, "VIPER: verifying the integrity of peripherals firmware," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, Chicago, IL, USA, 2011.

[38] A. Seshadri, M. Luk, and A. Perrig, "SAKE: software attestation for key establishment in sensor networks," *Ad Hoc Networks*, vol. 9, no. 6, 2008.

[39] A. Seshadri, M. Luk, A. Perrig, L. V. Doorn, and P. Khosla, "SCUBA: secure code update by attestation in sensor networks," in *Proceedings of the 5th ACM Workshop on Wireless Security*, Los Angeles, CA, USA, 2006.

[40] D. Perito and G. Tsudik, "Secure code update for embedded devices via proofs of secure erasure," in *Proceedings of the 2010 European Conference on Research in Computer Security*, Athens, Greece, 2010.

[41] Y. Li, J. M. McCune, and A. Perrig, "SBAP: software-based attestation for peripherals," in *Proceedings of the 2010 International Conference on Trust and Trustworthy Computing*, Berlin, Germany, 2010.

[42] N. Asokan, F. Brasser, A. Ibrahim, and A.-R. Sadeghi, "SEDA: scalable embedded device attestation," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*, Denver, CO, USA, 2015.

[43] A. T. Asokan, N. Davi, L. Ekberg et al., "C-FLAT: control-flow attestation for embedded systems software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, Vienna, Austria, 2016.

[44] U. Shankar, M. Chew, and J. D. Tygar, "Side effects are not sufficient to authenticate software," in *Proceedings of the 2004 USENIX Security Symposium*, San Diego, CA, USA, May 2004.

[45] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, Chicago, IL, USA, 2009.

[46] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, "New results for timing-based attestation," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, San Francisco, CA, USA, 2011.

[47] Aleph One, "Smashing the stack for fun and profit," 1996, http://www.phrack.com/issues.html?issue=49&id=14&mode=txt.

[48] J. Pincus and B. Baker, "Beyond stack smashing: recent advances in exploiting buffer overruns," *IEEE Security and Privacy*, vol. 2, Article ID 208C27, 2004.

[49] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 2005 ACM Conference on Computer and Communications Security*, Alexandria, VA, USA, 2005.

[50] K. Onarlioglu, L. Bilge, A. Lanzi et al., "G-free: defeating return-oriented programming through gadget-less binaries," in *Proceedings of 2010 Computer Security Applications Conference*, Austin, TX, USA, 2010.

[51] C. Peter, "LLVM control flow integrity," 2015, http://clang/llvm/org/docs/ControlFlowIntegrity.

[52] Microsoft, "Visual studio 2015 compiler options enable control flow guard," 2015, https://msdn/microsoft/com/en-us/library/dn919635.

[53] P. Team, "Pax non-executable pages design & implementation," 2003, http://pax.grsecurity.net/docs/noexec.txt.

[54] M. Corp and S. Andersen, *Part 3: Memory Protection Technologies in Changes to Functionality in Microsoft Windows XP Service Pack 2*, Microsoft Corp, Washington, DC, USA, 2004, http://technet.microsoft.com/en-us/library/bb457155.aspx.

[55] N. Burow, S. A. Carr, J. Nash et al., "Control-flow integrity: precision, security, and performance," *ACM Computing Surveys*, vol. 50, no. 1, p. 16, 2016.

[56] F. Armknecht, A. R. Sadeghi, S. Schulz, and C. Wachsmann, "A security framework for the analysis and design of software attestation," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, Berlin, Germany, 2013.

[57] G. O. Karame and W. Li, "Secure erasure and code update in legacy sensors," in *Trust and Trustworthy Computing*Springer, Berlin, Germany, 2015.

[58] F. Kohnhuser and S. Katzenbeisser, "Secure code updates for mesh networked commodity low-end embedded devices," *Computer Security—ESORICS*, Springer, Berlin, Germany, 2016.