fASLR: Function-Based ASLR via TrustZone-M and MPU for Resource-Constrained IoT Systems

Lan Luo, Xinhui Shao, Zhen Ling, *Member, IEEE*, Huaiyu Yan, Yumeng Wei, and Xinwen Fu, *Senior Member, IEEE*

***Abstract*—The address space layout randomization (ASLR) has been widely deployed on modern operating systems against code reuse attacks (CRAs), such as return-oriented programming (ROP) and jump-oriented programming (JOP). However, porting ASLR to resource-constrained IoT devices is a great challenge due to the limited memory space for randomization. We propose a function-based ASLR scheme (fASLR) for IoT runtime security utilizing the ARM TrustZone-M technology and the memory protection unit (MPU) supported by ARM Cortex-M processors. fASLR loads a function from the flash and randomizes its base address in a randomization region in RAM when the function is being called. We design novel mechanisms on cleaning up finished functions from the RAM and memory addressing to tackle the complexity of function relocation and randomization. Optimizations are applied to effectively reduce overhead introduced by runtime memory management. We also formally prove that user applications will run correctly with fASLR enabled. Compared with the related work, a prominent advantage of fASLR is that fASLR can run an application even if the application code cannot be completely loaded into RAM for execution. We test fASLR with 21 applications. The experimental results show that fASLR achieves a high randomization entropy and incurs a runtime overhead of less than 10%.**

***Index Terms*—Address space layout randomization (ASLR), code reuse attacks (CRAs), Internet of Things, microcontroller, TrustZone.**

Manuscript received 30 May 2022; revised 18 June 2022; accepted 26 June 2022. Date of publication 13 July 2022; date of current version 7 September 2022. This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB2100300; in part by the National Natural Science Foundation of China under Grant 62022024, Grant 61972088, Grant 62072103, Grant 62102084, Grant 62072102, Grant 62072098, and Grant 61972083; in part by the U.S. National Science Foundation (NSF) under Award 1931871 and Award 1915780; in part by the U.S. Department of Energy (DOE) under Award DE-EE0009152; in part by the Jiangsu Provincial Natural Science Foundation for Excellent Young Scholars under Grant BK20190060; in part by the Jiangsu Provincial Natural Science Foundation of China under Grant BK20190340; in part by the Jiangsu Provincial Key Laboratory of Network and Information Security under Grant BM2003201; in part by the Key Laboratory of Computer Network and Information Integration of Ministry of Education of China under Grant 93K-9; and in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization. *(Corresponding author: Zhen Ling.)*

Lan Luo is with the Department of Computer Science, University of Central Florida, Orlando, FL 32816 USA (e-mail: lukachan@knights.ucf.edu).

Xinhui Shao and Yumeng Wei are with the School of Cyber Science and Engineering, Southeast University, Nanjing 210096, China (e-mail: xinhuishao@seu.edu.cn; yumeng5@seu.edu.cn).

Zhen Ling and Huaiyu Yan are with the School of Computer Science and Engineering, Southeast University, Nanjing 210096, China (e-mail: zhenling@seu.edu.cn; huaiyu_yan@seu.edu.cn).

Xinwen Fu is with the Department of Computer Science, University of Massachusetts Lowell, Lowell, MA 01854 USA (e-mail: xinwenfu@cs.uml.edu).

Digital Object Identifier 10.1109/JIOT.2022.3190374

I. Introduction

With the booming IoT industry, there are rising concerns on the security and privacy of IoT devices. IoT application code is often written in unsafe programming languages, such as C and C++, thus, tends to be vulnerable to memory corruption attacks [1]. Exploiting program bugs, memory corruption attacks aim at corrupting sensitive data, such as function pointers, to achieve various attack purposes. One typical memory corruption attack is the code reuse attack (CRA), which hijacks the control flow and reuses the application code [2]. Memory corruption attacks and defenses have been actively studied for mainstream operating systems, such as Windows, macOS, Lniux, Android, and iOS.

In this article, we focus on defending against CRAs for resource-constrained IoT devices, particularly those running on microcontrollers (MCUs). It is an intuitive idea to port existing security schemes to IoT platforms. We study the use of address space layout randomization (ASLR) in memory-constrained IoT devices to mitigate CRAs, such as the return-oriented programming (ROP) and jump-oriented programming (JOP) by randomizing the memory layout of code and data. Modern operating systems often implement the following ASLR scheme. When an executable is loaded into RAM, its base (start) address is randomly chosen while the executable structure is kept almost intact. Fine-grained ASLR strategies have been proposed and randomize executable code at fine levels of basic blocks, functions, or instructions [3] within a loaded application image. However, porting ASLR to resource-constrained IoT devices is a great challenge due to the limited memory space.

We propose a function-based ASLR scheme (fASLR) based on the ARM Cortex-M processor with TrustZone-M enabled [4] to protect MCU-based IoT devices from CRAs that require the knowledge of locations of executable code snippets, such as ROP and JOP. fASLR takes advantage of hardware-based isolation provided by TrustZone-M. The runtime fASLR is located in a trusted execution environment (TEE), namely, the secure world (SW) of TrustZone, while the application code is in a rich execution environment (REE), namely, the nonsecure world (NSW), and denoted as the NS app. The NS app is protected by the memory protection unit (MPU). When a function in the MPU protected region is called, a hardware exception is raised and the control flow of the function call is redirected to our custom exception handler, which is used by the runtime fASLR for callee randomization. Compared to the

most recent related work [5] that requires loading the whole application code into RAM, fASLR can run an application even if the application on flash is too large to be completely loaded into RAM.

fASLR is user friendly and does not require any code instrumentation for the user code. A programmer only needs to compile the NS code via the GCC compiler with specific compiler flags. fASLR implements a block-based memory management strategy to manage randomized functions in RAM. To reduce overheads introduced by runtime fASLR, three optimizations are adopted: 1) fASLR cleans up finished functions from RAM only when the randomization region (RR) is full; 2) a novel stack unwinding mechanism is devised to precisely find all functions that are safe to be cleaned; and 3) function call rewriting is used to replace the destination addresses of call instructions with the base addresses of the corresponding randomized loaded callees so that the call instructions can jump directly to the target loaded callees without raising exceptions.

A conference version of this article [6] mainly focuses on the optimized fASLR. In this journal version, we add a basic memory management strategy and compare it with the optimized fASLR in the conference paper. In addition, we formally prove mechanisms adopted by fASLR do not affect execution correctness of the NS app. We also port three test apps with relatively high time overheads evaluated in the conference version to a more powerful TrustZone-M-enabled MCU and compare the overheads at different MCUs. Finally, we discuss the compatibility of using fASLR with other protection mechanisms for securing the runtime execution of MCU-based IoT devices.

Our major contributions are summarized as follows.

1) We propose a *function-based ASLR* scheme for resource-constrained IoT devices with limited RAM and flash. fASLR dynamically loads only needed functions into RAM and randomizes their entry addresses so as to achieve large randomization entropy.
2) Novel schemes are designed for fASLR to perform memory management and addressing. We carefully address the issue of addressing since functions are randomly moved around. Finished functions are removed from RAM when there is no RAM to execute new function calls. Therefore, our scheme can run an NS app that is larger than the RAM.
3) We formally prove that the NS app still runs correctly with fASLR via logical reasoning.
4) We implement fASLR with a TrustZone-M-enabled MCUs, SAM L11, and STM32. We validate the feasibility and performance of fASLR with 21 applications on SAML11 and three applications on STM32. fASLR incurs a runtime overhead of less than 10% for all the applications. We also compare the performance of fASLR on SAM L11 and STM32 and show larger RAM can reduce the overhead as expected.

Roadmap: The remainder of this article is structured as follows: we first discuss the background of TrustZone-M-enabled processors in Section II. The threat model, design goals, and system architecture of ASLR are then presented in Section III. We also demonstrate the workflow of fASLR and two technical challenges in this section. In Section IV, we discuss the technical challenges and present our solutions. We prove the execution validity of the NS app with fASLR in Section V. In addition, we analyze the effectiveness and performance of fASLR in Section VI, and present experimental results in Section VII. Finally, we discuss the compatibility of fASLR with other security mechanisms in Section VIII, present related work on fine-grained ASLR techniques for embedded systems in Section IX, and conclude this article in Section X.

II. Background

In this section, we introduce ARM Cortex-M MCUs and TrustZone-M, which is used in this article. ARM Cortex-M is a series of processors optimized for MCUs. Such processors come equipped with MPU, specific exception model, and different processor modes for security concerns.

Memory Protection Unit: The MPU is the security extension that enforces memory access permissions (i.e., read, write, and execute) for memory regions. Any access violation at memory address protected by MPU will trigger the ARM HardFault exception handling. Once such an exception is triggered, the processor will stop the current execution and execute the exception handler in the SW to respond to the exception. Before the execution of the exception handler, the processor context is first preserved in the call stack. The stack frame of the exception context is composed of the `status registers` (`xPSR`), `program counter` (`PC`), `link register` (`LR`), and general-purpose registers `R12` and `R0` to `R3`. At the same time, `LR` is set with `EXC_RETURN`, which is the address where the exception occurs.

Processor Mode: ARM Cortex-M processors support two processor modes, i.e., thread mode and handler mode. While the thread mode is for normal program execution and can be either privileged or unprivileged, the handler mode is for exception handling and only supports privileged software execution.

TrustZone-M-Enabled MCU: MCU often runs either a bare-metal-embedded application that usually consists of an infinite loop performing a sequence of operations or a lightweight real-time operating system (RTOS) such as FreeRTOS [7]. The applications or RTOS are stored in the MCU on-chip memory. There are two types of MCU on-chip memory: 1) flash or EEPROM as the nonvolatile memory and 2) RAM as the volatile memory. Usually, an MCU program is programmed into the flash and executed directly in the flash, though MCUs allow running code snippets in RAM for performance concerns.

TrustZone-M is a hardware-based security technique designed for MCUs, providing two isolated execution environments named SW as the TEE, and NSW as the REE. The on-chip resources, such as memories and peripherals are divided into the two worlds as well. For simplicity, we use the word "Secure" to describe resources in the SW and use "Nonsecure" for those belonging to the NSW. Secure application (abbreviated as app), for example, is an app in the SW. In a TrustZone-enabled system, an SW program is able to access

resources in both worlds, while a program in the NSW is considered to be untrusted and can only access resources in the NSW directly.

III. fASLR-Enabled System

In this section, we first present the threat model and design goals of our ASLR scheme—fASLR. We then introduce the architecture of an fASLR-enabled system and the workflow of fASLR. Finally, we discuss challenges for implementing a practical fASLR.

A. Threat Model

fASLR leverages ARM Cortex-M processors and hardware-based techniques, including TrustZone-M, MPU, and exception handling mechanism. Based on the hardware isolation provided by TrustZone-M, on-device system resources are divided into two worlds, namely, the SW and the NSW.

We assume a TrustZone-M-enabled device has the following security features.

1) Main components of fASLR reside in the SW and can be fully trusted. The application (denoted as NS app) is located in the NSW and may be vulnerable.
2) The NS app is located at a fixed address in the NS flash and is executed from the flash (instead of RAM) by default.
3) The device supports the memory protection mechanisms such as the MPU.

We assume an adversary has the following capabilities.

1) The NS app may be subject to CRAs such as the ROP attack.
2) The adversary can obtain the binary of the NS app, disassemble the binary, and obtain code gadgets for CRAs.

B. Design Goals

fASLR is designed to achieve the following goals.

1) *Mitigating CRAs:* The scheme shall provide dynamic function-level code randomization for resource-constraint IoT devices to mitigate CRAs, which require a certain chain of gadgets found in the NS app. The randomization shall achieve high entropy to defeat brute-force guessing attacks.
2) *Usability:* The scheme shall be user friendly and will not add much burden of programming.
3) *Low Runtime Overhead:* The proposed scheme cannot introduce large overhead in terms of time and space and affect the NS app performance much.

C. System Architecture

As illustrated in Fig. 1, fASLR has three key components: 1) the *static preprocessing module* (SPM) for compilation time preparation; 2) the *boot engine* (BE) for boot time configuration; and 3) the *function randomization engine* (FRE) for runtime function-level randomization.

Static Preprocessing Module: The SPM serves two major purposes.

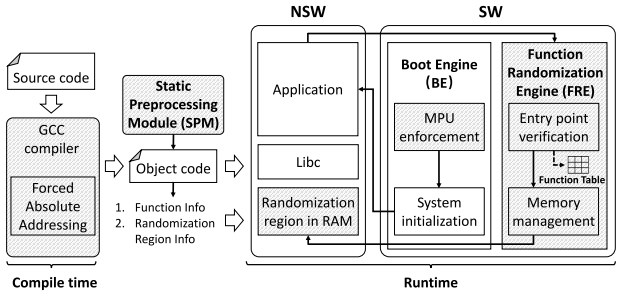


Fig. 1. fASLR architecture.

1) *Creating the Function Table (FT):* Once the NS app code is compiled via a GCC compiler, the SPM tries to extract needed information of all functions in the ELF object file, including function entry point addresses and function sizes from the symbol table, and function stack frame sizes from the `.debug_frame` section of the ELF file. Function information is recorded in a data structure called FT.
2) *Profiling the RR:* Extracting RAM usage information from the compiler output file, the SPM determines the size and location of the largest unused RAM space as the RR. Users can also set a smaller RR by manually modifying related configurations.

Boot Engine: When a TrustZone-M-enabled device boots, the boot flow is the Secure bootloader, Secure app, and then the NS app. The NS app starts with the `reset` handler that calls the first function, e.g., `main()`. The BE is a part of the Secure bootloader stored in the SW flash. It configures and enables the MPU to mark the NS app code in the flash as nonexecutable for two purposes: 1) MPU prevents the NS app in the NS flash from being exploited by CRAs and 2) once the NS code is set as nonexecutable, any attempt to execute the NS code triggers a hardware exception, which is handled by the HardFault exception handler in the SW [8].

Function Randomization Engine: The FRE is a part of the HardFault exception handler and handles invoked functions in the NSW flash protected by the MPU. It serves two purposes, i.e., *function entry point verification* and *memory management*.

When a HardFault exception occurs, the FRE fetches the return address of the exception, which is the entry point of the invoked function, through the NSW exception stack frame. Then, the function entry point verification is performed by comparing the return address to all legitimate function entry point addresses in the FT until there is a match. After a match, the FRE obtains the size of the corresponding function from the FT for later use in randomization. A HardFault exception may be caused by other reasons, for instance, memory access violation when an adversary launches CRAs trying to execute an instruction not at legitimate function entry points. In such a case, a security alert shall be raised.

After the function entry point verification, memory management is performed. Specifically, the invoked function is randomly relocated to a RAM region within the RR. After the function randomization, we need to carefully handover the control flow from the exception handler to the relocated

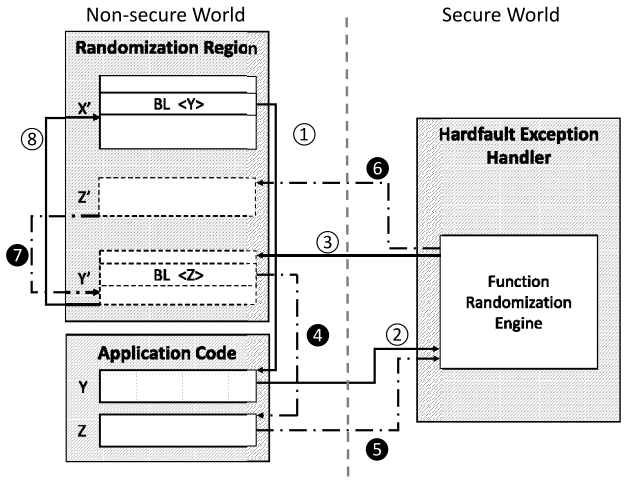


Fig. 2. Program flow of function X, Y, and Z. For any function F in the NS app, we use F' to represent its corresponding duplicate in the RR.

function. We deliberately manipulate this by overwriting the return address of the exception handler on stack with the new function entry point so that the execution mode will change back to the mode of the NS app with correct privileges.

D. Workflow

Offline—Compilation and Flashing: After the NS app is compiled and linked by the GCC compiler at compile time, the SPM creates the FT offline according to the NS app ELF file. The FT and NS app image are then flashed to the SW and NSW flash, respectively. The BE and FRE are also flashed to the SW flash.

Runtime: Fig. 2 shows the program flow, which is an iterative sequence of function calls (e.g., ① and ❹), MPU violation exception (e.g., ② and ❺), runtime function randomization, function execution (e.g., ③ and ❻), and function return (e.g., ❼ and ⑧).

Once we turn on the device power supply, the BE boots the system and then the NS app initiates the `reset` handler [9]. After a sequence of initialization operations, the `reset` handler branches to the main application code, i.e., `main()`. Both attempts of executing the reset handler and `main()` trigger runtime fASLR, and their code is loaded by the FRE to the RR in the RAM for execution. During the execution of the `main()` function, the control flow can divert to a callee on the MPU-protected flash only if the callee is invoked by `main()`.

In an fASLR-enabled system, when a function call occurs in the RR, it jumps to the entry point of the callee in the original MPU-protected application and, thus, triggers the MPU violation exception. The FRE then conducts runtime function randomization for the callee and diverts the control flow to the callee relocated in the RAM. During the execution of the callee, any function call in the callee can also trigger a MPU violation exception. The FRE handles function calls and randomization in such an iterative way above. The control flow returns to the caller in the RAM once its callee is finished. fASLR does not interfere with the function return mechanism, and the function in the RAM returns normally as functions in a system without fASLR do. Note that in an fASLR enabled system, a callee returns to the relocated caller in the RR since that is where the function call really occurs.

Fig. 2 presents an exemplary program flow for the call path $X \rightarrow Y \rightarrow Z$ of functions X, Y, and Z. A call path illustrates the calling relationship. Starting from the leftmost one, each function in the path calls the function right after it. Suppose that function X has been loaded to the RR and the program flow starts from the relocated function X'. When X' calls Y, the attempt of executing Y (①) results in an MPU violation exception (②), handled by the FRE inside the HardFault exception handler. Y is then relocated to the RR as Y' and, consequently, the control flow is redirected to Y' (③). During the execution of Y', Y' attempts to call Z (❹), and the MPU violation exception (❺) is triggered again and is then handled by the FRE (❻). Finally, the control flow returns from Z' to Y' (❼) and Y' to X' (⑧) when Z' and Y' finish execution.

E. Challenges

A practical fASLR faces the following challenges. We address these challenges in detail in Section IV.

Memory Management: We target MCUs with limited RAM and the whole NS app may not be loaded into the RAM for execution. Therefore, a memory management strategy is needed to dynamically trim loaded functions, assuring free space for subsequent function randomization. *Ancestor functions* are defined as direct or indirect callers of the current running function. Such functions are awaiting returns from some ongoing function calls, and shall not be trimmed before their descendants return. *Finished functions* are those that have finished execution and are not ancestors of any running function. They can be disposed safely. The runtime FRE is supposed to distinguish finished functions and select an appropriate timing to trim them from the RR.

Memory Addressing: A function in the RR may contain branches that use absolute or relative addresses. All absolute branches in the ARMv8-M architecture compiled by GCC are function calls, which would not be affected by the relocation and will function normally. Relative branches within a function can work normally as well since a relative position would not change when the function is relocated as a whole. However, relative branches may be used to jump between functions. In an fASLR-enabled system, those relative branches can lead the control flow to branch to an unexpected destination as function-based randomization changes the relative position of two functions.

IV. Memory Management and Addressing

In this section, we address the challenges of fALSR raised in Section III-E and present our memory management and addressing schemes.

A. Memory Management

The dynamic memory management of fASLR positions functions randomly within the RR. Function cleaning strategies are utilized to remove finished functions from the RR.

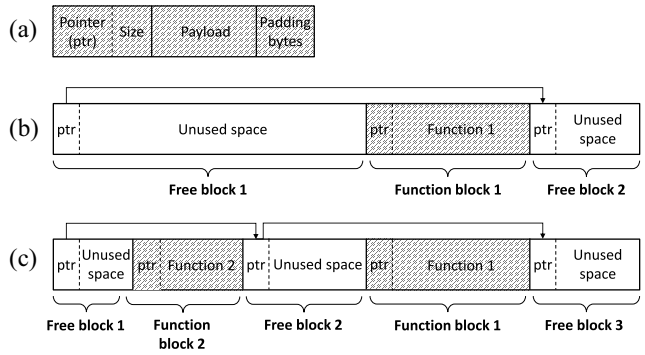


Fig. 3. Memory fragmentation. (a) Structure of the RR. (b) Memory layout before loading function 2. (c) Memory layout after loading and randomizing function 2.

1) Memory Fragmentation Management: In the RR, each loaded function occupies a function block. A disposed function block becomes a free block. If there are any adjacent free blocks, the FRE merges them into one big free block. All free blocks are managed by a linked list. A function block, as presented in Fig. 3(a), consists of a two-word metadata, a payload, and padding bytes for memory alignment. The metadata contain the size of the block and the pointer which points the next free block. The payload region is used to store the randomized function.

When the system starts, fASLR initializes the whole RR as a big free block since no function has been allocated yet. Once a function is called in the NSW, the FRE allocates a function block for the target function. Specifically, it first scans the linked list and finds out all free blocks larger than the target function in the payload region. The FRE randomly selects one block among the discovered free blocks and then randomly allocates the target function to the selected free block. After the allocation, new free blocks may be generated and the linked list will be updated accordingly. Fig. 3(a) and (b) illustrates the case of randomizing function 2 when there are two free blocks. Function 2 is consequently allocated to the middle of free block 1. The new free block 1 and free block 2 are then formed.

2) Baseline Function Cleaning: We now present an intuitive baseline function cleaning scheme which dynamically cleans up finished functions from the RR. To find out finished functions for cleaning, we create a stack-like data structure named *Trace Stack* to store metadata, which is denoted as a *function record*, of every loaded function within the RR. A function record is created and pushed into the trace stack by the FRE when a new function is called. A function record contains the information of the callee function, mainly including: 1) *loadAddress*—the new entry point of the callee in the RR and 2) *size*—the size of the callee. The sequence of function records in the trace stack, from bottom to top, reflects the order of function calls. It can be observed from Fig. 4 that if a function's record is above function A's record in the trace stack, that function is either directly called by A (e.g., function B), or its ancestor is called by A (e.g., function C). Based on this observation, we create the algorithm for cleaning finished functions from the RR as shown in Algorithm 1.

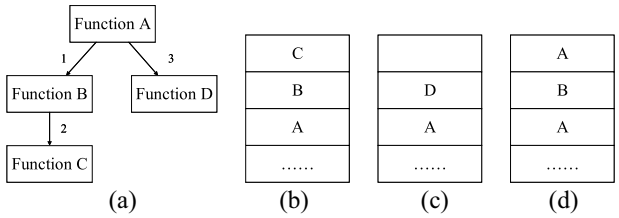


Fig. 4. Function call graph and the *Trace Stack*. (a) Call graph. (b) *Trace Stack* before D is called. (c) *Trace Stack* after D is called. (d) *Trace Stack* contains multiple As.

Algorithm 1 Function Cleaning Algorithm

```
returnAddress ← readException(sp)
for i = 1; i < stackLen; i++ do
    if (returnAddress ≤ stack[i].endAddress) &
(returnAddress ≥ stack[i].loadAddress) then
        for j = i; j < stackLen; j++ do
            funcRecord ← stackPop()
            freeFunction(funcRecord)
        end for
    end if
end for
```

Algorithm 1 runs every time the Hardfault exception raises and the FRE has verified the entry point of the callee. The FRE first locates the caller record, e.g., at position i of the trace stack, by comparing the return address of the callee (`returnAddress`) with the function entry address (`stack[i].loadAddress`) and the function end address (that is `stack[i].loadAddress+size`) of each record in the trace stack. According to the aforementioned analysis of the function record order, any records above the ith record map to functions which were called after the current caller. Since the caller is running at the moment, it can be deduced that functions which have records above the caller record have already returned, so it is safe to remove all of those functions from RAM. For example, when function D is being called by function A in Fig. 4, the current caller is A. Any of its descendants (B and C) in the call graph have already returned, therefore, can be removed before loading D. In Algorithm 1, once the caller record in the trace stack is found, all function records above it are popped out via `stackPop()` and the corresponding function space in the RR is cleared via `freeFunction(funcRecord)`.

Note that this cleaning strategy is a "conservative" strategy that only ensures all removed functions are finished but may miss some finished functions in certain cases. Fig. 4(d) presents an instance. When the trace stack contains multiple records of the same function, suppose the current caller is the bottom A, the function cleaning strategy, however, will treat the top A as the caller and miss cleaning top A and B. The top A, B and bottom A will not be cleaned until an ancestor function of bottom A calls a new function.

3) Optimized Function Cleaning: The major issue of the baseline function cleaning is that the exception handler is triggered every time there is a function call. Too many exceptions cause large overheads. We improve the memory management

of fALSR with respect to the timing of function cleaning and calling loaded functions.

1) *Call Stack Unwinding:* Finished functions are found through unwinding the Nonsecure call stack.
2) *Cleaning on Demand:* Finished functions are cleaned up only if the available RR space is not large enough for the callee.
3) *Call Instruction Rewriting:* We further reduce the runtime overhead by overwriting a call instruction in a loaded function if the callee of that call instruction has already been loaded into RAM.

Call Stack Unwinding: The key of function cleaning is to distinguish finished functions from all loaded functions in RAM. However, it is difficult to trace all finished functions at runtime because fASLR runtime does not capture any function return information. Instead, our approach finds ancestor functions of the current callee, and records all loaded functions. Any function that is a loaded function but not an ancestor function is a finished function that can be disposed. Now, the problem is decomposed to record all loaded functions and find all ancestor functions.

Like the trace stack working for the baseline memory management, a queue structure named the loading queue (LQ) is used to store metadata, namely, *function record*, of all loaded functions in the RAM. The FRE pushes a function record into the LQ when an unloaded function is called.

We also need to find out all ancestor functions. Modern computer system uses the call stack to retain return addresses of functions that have been called but have not returned yet. Such functions are direct or indirect callers of the current running function, which is also the callee when program execution is trapped in the FRE in our system. Therefore, functions which have their stack frames in the call stack are ancestor functions of the callee. To figure out all functions in the call stack, *stack unwinding* is needed. Basically, stack unwinding helps to locate *all return addresses in the call stack*. A return address then tells where the caller is within the RAM. If we can obtain all return addresses on the call stack, by comparing each return address with the function records in the LQ, we are able to identify all ancestor functions.

Frame pointer is an intuitive approach of unwinding the call stack [10]. However, the Armv8-M architecture only implements the Thumb instruction set, which does not support the frame pointer convention. To achieve stack unwinding without frame pointers, we devise a stack unwinding method utilizing the stack top address and the stack frame sizes of all functions to resolve return addresses on the call stack. Recall that the stack frame size information of each function is extracted offline by the SPM from the `.debug_frame` section of the ELF file and stored in the FT.

In ARM, by convention, return address is the first object pushed onto the stack when there is a function call, and is at the bottom of the callee's stack frame. Once a function call triggers the HardFault exception and the program execution is trapped by the FRE, the top stack frame is the exception stack frame of the HardFault exception handler and has a fixed length s_e. The current stack top can be obtained through the `SP` register of the NSW. The frame top of the first function f_1 (namely, the current caller) is $T_1 = SP + s_e$. According to the `LR` register, which stores an address within f_1, FRE is able to search the frame size s_1 of f_1 from the function records in the LQ. To access the return address of f_1, the frame bottom B_1 is calculated by $B_1 = T_1 + s_1$. Following this procedure, the FRE is able to resolve return address of every stack frame from the stack top to bottom. Algorithm 2 presents our stack unwinding procedure.

Algorithm 2 Call Stack Unwinding Algorithm

```
nsSp = getNSSp()
returnAddress = readExceptionStackFrame(nsSp)
funcSp = nsSp + sizeof(ExceptionStackFrame)
for i = 1; i < loadingQueue.size; i++ do
    if (returnAddress ≤ loadingQueue[i].endAddress) &
(returnAddress ≥ loadingQueue[i].loadAddress) then
        funcRecord = loadingQueue[i]
        funcRecord.state = UNFINISHED
        funcSp = funcSp + funcRecord.callFrameSize
        returnAddress = getReturnAddress(funcSp)
    end if
end for
```

Note that recursion is compatible with our function cleaning strategy. A recursive function is the function that calls itself. In our compilation environment, a recursive function uses a relative branch instruction. Therefore, when a recursive function is loaded to the RR, it can still call itself with the relative branch without triggering the MPU violation exception.

Cleaning on Demand: The FRE removes functions only when the RR does not have enough space to load a new function. Before loading a function to RAM, the FRE checks if there is enough memory space for it. If not, the FRE first recovers rewritten call instructions in the loaded functions as introduced below. It then unwinds the call stack, finds out all ancestor functions, and marks those functions in the LQ as unfinished. According to the marked LQ, the FRE disposes all finished functions and updates the LQ. The call instruction rewriting mechanism, which will be introduced next, ensures that any function pointers pointing to a trimmed function will be restored to point to the original function in flash.

Call Instruction Rewriting: Function call rewriting optimizes the memory management scheme so that finished but not disposed functions in RAM can be called again without triggering the HardFault exception. Specifically, when a function call occurs and the control flow is trapped in the HardFault exception handler, the FRE first checks if the callee is in the RR. If so, the FRE overwrites that call instruction (in the loaded caller) to change the destination address of the call (i.e., the entry point of the callee in flash) with the entry of the loaded callee in RAM. Thus, the caller will directly jump to the loaded callee next time this call instruction executes. The rewriting history, including which instruction is rewritten and what the original instruction is, is recorded in the rewriting list (RL). Such records are used to recover the call instructions with callees' flash entry points before function cleaning, since the loaded callees of those call instructions might be disposed.

TABLE I
ATOMIC OPERATIONS OF RUNTIME FASLR

Operation A: Act-1 → Act-2A → Act-3 **Operation B:** Act-1 → Act-2B → Act-3 **Operation C:** Act-1 → Act-2C → Act-3	
Action-1	Obtaining the base address of the callee in flash and querying the corresponding function record from the *Loading Queue*
Action-2A	**Condition** *If the function record is not in the Loading Queue and the randomization region has available space for the callee:* Loading the callee into the randomization region and pushing its record into the *Loading Queue*
Action-2B	**Condition** *If the function record is in the Loading Queue:* Rewriting the call instruction so that its destination parameter is the entry point of the loaded callee, and adding the rewriting information to the *Rewriting List*
Action-2C	**Condition** *If the function record is not in the Loading Queue and the randomization region is full:* Finding all finished functions via stack unwinding. For each finished function, querying it from the *Rewriting List*. If a record is found, restoring all call instructions that are related to the finished function and delete the rewriting record. Cleaning up all finished functions and delete corresponding function records from the *Loading Queue*
Action-3	Recovering normal program execution to the entry point of the loaded callee in RAM.

B. Memory Addressing

Control flow instructions using relative addresses in the ARMv8-M instruction set include branch (*B*), branch with link (*BL*), and conditional branches (*CBNZ/CBZ*), among which the *BL* is used to branch between functions. It is difficult for fASLR to handle such relative addressing without instruction patching, namely, runtime instruction update. Note that the relative positions of two functions changes after function randomization. Recalculating all the relative addresses used in the randomized function and updating the related instructions with the new relative addresses will result in unacceptable overhead in performance.

fASLR eradicates relative addressing at compile time. A user needs to access the source code of the app (including libraries) and compile the app with specific compilation flags (i.e., *-mlong-calls*, *-fno-jump-tables*). As a result, the original relative function calls now use absolute addressing. It is worth noting that compiling with such flags would not break the normal build process or affect runtime behavior of the original program.

V. EXECUTION VALIDITY OF NS APP WITH FASLR

In this section, we prove that an NS app can correctly execute with the optimized fASLR enabled. That is, the program's functionality represented by the control flow and data flow remains unaltered with or without fASLR.

A. Problem Analysis

fALSR can be decomposed as compile-time fASLR and runtime fASLR. Compile-time fASLR applies to program compiling time with no attempt to change the program logic, and is guaranteed to have no influence on the NS app runtime program logic. This leaves our proof to only runtime fASLR. Unless otherwise specified, fASLR refers to runtime fASLR.

With runtime fASLR, the overall running system can be seen as a series of fASLR operations being inserted into the original execution of the NS app using serial execution. Therefore, from the perspective of execution timeline, proving the execution correctness of NS app with fASLR is equivalent to proving that each fASLR operation during program execution has no influence on the NS app program logic.

The program logic for the whole app is still a complex concept. Whenever discussing the influence of an fASLR operation on NS app program logic, we attempt to dissect the program logic via logical analysis so that the influence can be effectively depicted and the possible range of affected program logic can be narrowed down.

Third, we are also aware that runtime fASLR depends on a few auxiliary data structures, such as the FT and LQ. In the proof, we ensure that operations of runtime fASLR do not harm the validity of such data structures at any moment, which are also essential for maintaining the original program logic.

B. fASLR Operations

Runtime fASLR has three operations which are basic atomic interactions of runtime fASLR with the system. Each operation can be further decomposed into a few actions. Table I lists all fASLR operations and actions. Some actions like Action-1 and Action-3 are shared among different operations. These operations are atomic such that no overlapping between operations or overlapping between operation and program executions are possible. We can analyze the NS app program logic under fASLR by investigating influence of each operation separately. Fig. 5 shows an overview of the execution flow of fASLR operations.

C. Definitions, Concepts, and Assumptions

1) Definitions:

Definition 1 (Correctness of Program Execution): We say an NS app with fASLR executes correctly if the app's functionality is the same as the NS app without fASLR.

Definition 2 (Static Call information): refers to the hard-coded destination addresses in the program used for function calls.

2) Concepts: We first introduce the auxiliary data structures used by fASLR and then define the validity of the RR, function calls and function returns. Such concepts will be used in our following proof.

Auxiliary Data Structures: fASLR employs four auxiliary data structures to maintain information used at runtime.

1) *Loading Queue:* LQ maintains function records for each loaded function currently in the RR. For a loaded function f, its function record can be denoted as

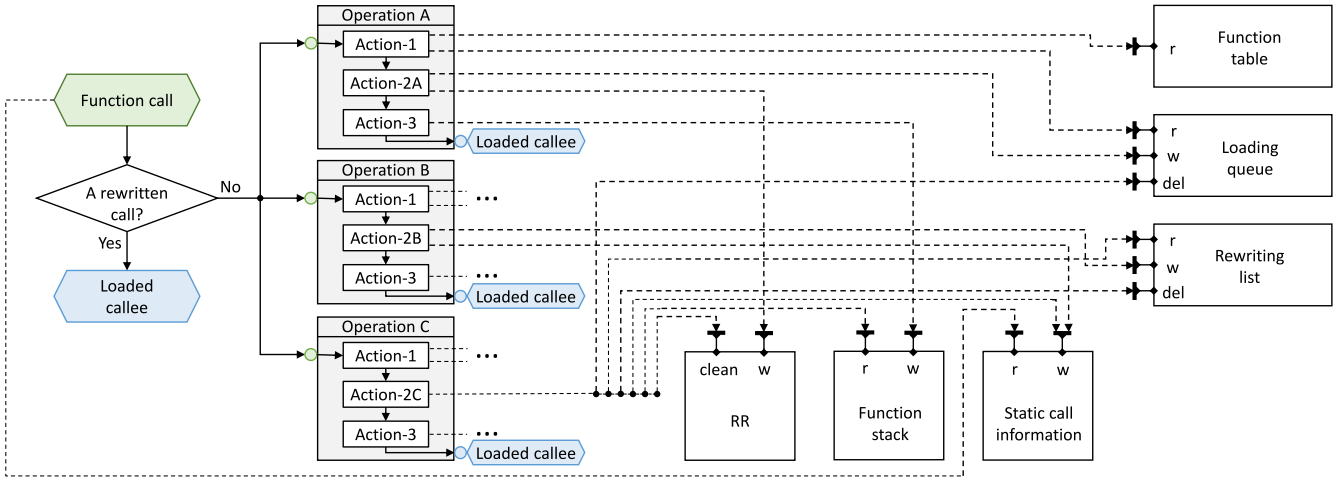


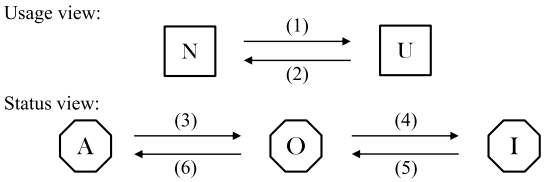
Fig. 5. Execution flow of fASLR operations. Solid lines represent control flow, and dotted lines are accesses to memory or data structures. Accesses from Action-1 and Action-3 in Operation B and C to memory and data structures are omitted since they are exactly the same as in Operation A.

$\{\text{Addr}_f, \text{Addr}'_f, S_f\}$, where Addr_f is the original base address of f in flash, Addr'_f is the randomized base address of f in the RR, and S_f is the size of f. In our implementation, the function record may contain other elements solely for convenience of programming. We ignore them here since they are irrelevant to the proof.

2) *Rewriting List:* RL traces call instructions which have been rewritten by fASLR. For a rewritten instruction i, a rewriting record is created in RL as $\{\text{Dest}'_i, \text{Addr}_i, \text{Dest}_i\}$, where Addr_i is the address of instruction i, Dest'_i the current destination address of i after rewriting, and Dest_i the original destination before rewriting.
3) *Function Table:* FT is a static table storing function information generated at compile time.
4) *Free Block List (FBL):* FBL is a linked list that dynamically links all free blocks within the RR.

Among these four data structures, FT is static and will not be changed by fASLR at runtime, therefore, it remains valid during the whole execution. FBL is designed for fast searching for free blocks within the RR, and contains part of information in LQ. If LQ is valid, FBL must be valid at the same time. Given LQ, it is not necessary to discuss FBL separately. Therefore, we will focus on analyzing the validity of LQ and RL in this section.

Validity of the Randomization Region: The RR is the RAM space allocated for runtime randomization. In fASLR, we use LQ to help record the usage of RR. By checking LQ, we could categorize memory in the RR into two states: 1) used region and 2) not used region, thus forming a $\{U|N\}$ partition of RR. We refer to this as the *usage view* of RR. The region here is a general notation for a continuous memory space sharing the same state. By further cross-checking both the LQ and the function stack, we could categorize the memory of RR into exactly one of the three states: 1) *in-use:* a region is in-use if it holds a function block and the function execution is unfinished; 2) *occupied:* a region is occupied if it is a function block but the function execution is finished; and 3) *available:* a region is available if it is a free block. This forms a $\{I|O|A\}$ partition of RR. We refer to this as the *status view* of RR.



(1) Function is loaded into region **(4) Function becomes in-use**
(2) Function is cleaned **(5) Function is finished but not cleaned**
(3) Function is loaded but not in-use **(6) Function is cleaned**

* Transition (1) – (2) are verified by checking function LQ;
Transition (3) – (6) are verified by checking LQ and the stack jointly.

Fig. 6. State transitions in usage view and status view under different transition rules.

A state in the usage view and Status view can transition to another state under state transition rules defined in Fig. 6.

RR is a special component in fASLR as it is the only structure containing function code duplicates that will be used for program execution. Consequently, we specially discuss the requirements for the validity of RR. In order to maintain the validity of RR, we require that all state transitions in both views at any moment must follow the transition rules as shown in Fig. 6, and verify the validity of RR at any moment by checking the satisfaction of the following two conditions.

1) For any function record $\{\text{Addr}_f, \text{Addr}'_f, S_f\}$ in LQ, it should be consistent with the actual memory content stored in the RR. That is, the memory region ranging from Addr'_f to $\text{Addr}'_f + S_f$ shall contain the duplicate of function f.
2) The usage and status views of RR should be consistent, i.e., they should satisfy $N = A$ and $U = I + O$.

Validity of Function Calls and Returns: Since fASLR not only affects the current trapped function call, it may also have influence on function calls and returns that will occur in the future, we first categorize function calls and returns that are related to the proof. If not explicitly stated otherwise, here, the function calls (or returns) refer to active and future function calls (or returns), not finished ones. An active function call is a function call that is intercepted by fASLR and there is an

ongoing fASLR operation with regard to the trapped function call. A future function call is one that will occur after the active function call. A function return will definitely occur after the active function call is executed.

As an fASLR operation performs when an active function call occurs, the operation is in full control of the execution of the active function call. The validity of the active function call depends on how the operation affects the program logic of the call. On the other side, any future function call that may occur after the current operation is generated by the execution of a call instruction. The validity of such a call depends on whether the call instruction points to the correct callee function. In other words, the memory contents at the location pointed by the destination address of the call instruction (i.e., static call information) must be the correct callee function (either the original callee in flash or the corresponding callee duplicate in RR).

A function return obtains the return location through return address. In ARM, leaf subroutine and nonleaf subroutine use different ways to obtain the return address. For a leaf subroutine, the return address is stored in the `lr` register, while a nonleaf subroutine uses the return address stored in the stack frame. Because fASLR does not intervene in the process of putting a return address into the `lr` register or onto a stack frame, it can be affirmed that all return addresses are generated correctly as the original NS app (without fASLR) does. After the generation of the return addresses, fASLR never changes the lr register and stack frames used by normal program execution. So, until a function returns, the return address, no matter it is in `lr` or on the stack, remains unchanged. The validity of future function returns therefore requires that the memory content pointed by the return addresses is valid for function returns.

In summary, the validity of the active function call depends on if the program logic is affected by an fASLR operation; the validity of future calls is determined by static call information and content validity of RR; Similarly, the validity of future returns relies on the content validity of RR. In addition, the auxiliary data structures, as we introduced in concepts, must remain valid throughout the program execution so that any fASLR operation can perform as designed. Thus, we can conclude that the validity of function calls and returns critically relies on the validity of the following four objects throughout the program execution.

1) Program logic of the active call.
2) Static call information.
3) Auxiliary data structures (LQ and RL).
4) Randomization region.

We name the union of these four objects as the *critical reliance set*. The validity of the critical reliance set is in fact the necessary and sufficient condition for the validity of the function calls and returns. In later proof, we check the validity of the critical reliance set whenever the validity of function calls and returns need to be verified.

3) Assumptions:

Assumption 1 (Correct Initial System Status): When the system starts, the initial status of the whole system, including the program logic, static call information, auxiliary data structures, and memory state, is correct.

Assumption 2 (Serial Execution System): The MCU that runs the program with fASLR is a serial execution system, in which one computation can begin only after the previous computation completes without parallelization.

D. Propositions

Proposition 1: Any operation of fASLR affects only function calls and returns of the NS app execution.

Proof: Program logic, which refers to the implementation of the program's design, is mutually determined by control flow and data flow at runtime. For the operations listed in Table I, the runtime fASLR does not modify any data flow during execution. We thus focus on the influence on the control flow. The control flow of a program can be divided into control flow within a function (i.e., branches inside a function and nonbranch execution) and control flow between functions (i.e., function calls and returns). It can be seen that operations listed in Table I do not affect any execution within a function. Hence, all Operations A, B, and C may affect only function calls and returns of the NS app execution. ■

E. Lemmas and Theorem

Lemma 1: The correctness of the NS app execution maintains when Operation A finishes.

Proof: According to Proposition 1, Operation A can only affect function calls and returns of the NS app. The validity of function calls and returns, as we introduced in concepts, can be verified by checking the validity of the critical reliance set. Therefore, we prove Lemma 1 through justifying that the critical reliance set remains valid when Operation A finishes. Since Operation A is the ordered combination of three actions, we first analyze whether each action affects the validity of the critical reliance set. We assume that the critical reliance set is valid upon the entry of Operation A. Such an assumption is natural and common, and will be consistently used among the proofs of all lemmas.

Action-1 only reads the entry address of the callee from the stack and search the function record from the LQ. It never writes any values or memory contents, and would not affect the validity of the critical reliance set.

The condition in Action-2A first guarantees the callee can be loaded into the RR. The loading action changes the status of the RR and LQ. Note that before this action, both RR and LQ are valid. So, we focus on the changes applied on them. After loading the callee (denoted as x) into the RR, for the occupied memory region (denoted as m) of the callee, indeed a new function record $\{\text{Addr}_x, \text{Addr}'_x, S_x\}$ is created in the LQ to record that this region is being used. These are the solely changes to the RR and LQ, and these two changes are entirely correspondent. Conditions for the validity of RR are satisfied. Therefore, both LQ and RR remain valid.

When Action-3 is applied, the RR contains x, which is the duplicate of the callee. Action-3 forwards the control flow to this duplicate so the program logic is exactly the same as before.

So far, we have proved that any action of *Operation A* does not affect the validity of the critical reliance set. Hence, the critical reliance set will remain valid at the exit of Operation A. The correctness of the NS app execution holds at the exit of Operation A. ■

Lemma 2: The correctness of the NS app execution maintains when Operation B finishes.

Proof: Compared to Operation A, the only difference of *Operation B* is that it performs Action-2B instead of Action-2A. Therefore, we focus on the changes brought by Action-2B. The effects of other actions of Operation B are the same as Operation A.

Action-2B solely changes the static call information, i.e., the destination address of the active function call and adds the corresponding rewriting record to the RL. The new destination address Addr'_y obtained from LQ is the correct base address of the duplicate callee because LQ is valid upon the entry of this operation. So, it is straightforward to see that both the static call information and RL remain valid. The correctness of the NS app execution is kept when Operation B finishes. ■

Lemma 3: The correctness of the NS app execution holds when Operation C finishes.

Proof: We focus on Action-2C since it is the only difference between Operations C and A.

Action-2C involves function cleaning and call instruction restoring. Function cleaning changes RR and LQ. As for the memory content of RR, *Action-2C* solely cleans the memory with state O from the status view according to LQ and stack, and deletes corresponding function records in LQ. This means LQ remains valid, and the state transition of such memory region is $O \rightarrow A$, which follows the previously defined transition rule. The consistency between LQ and RR is kept and satisfies the first requirement for the validity of RR. Based on the function cleaning process, the status view of the RR before and after function cleaning can be presented as $\{I|O|A\}$ and $\{I'|O'|A'\}$, where $I' = I$, $O' =$ Empty, and $A' = A + O$. Similarly, from the usage view, we have $N' = N + O$, $U' = U - O$. Recall the second condition of a valid RR ensures $N = A$ and $U = I + O$. After cleaning, we have

$$U' = U - O = I + O - O = I = I' + O'$$
$$N' = N + O = A + O = A'.$$

This means that the RR after cleaning satisfies the second condition as well. Now, we can conclude that RR and LQ are still valid after cleaning.

Function cleaning may affect static information and RL. When loaded functions are cleaned from RAM, rewritten call instructions with the destination addresses pointing to the cleaned functions need to be restored to point to their original callees in flash. This is exactly what we do in this action. While cleaning a function f, by scanning all corresponding function records in RL, fASLR can precisely identify in static information of the set of call instructions pointing to the cleaned function. For each found function record $\{\text{Addr}'_f : \text{Addr}_i, \text{Addr}_f\}$, fASLR deletes the rewriting record and restores the instruction i to use the original address Addr_f of the callee as the destination address. So, RL remains accurate and all static call information remains valid after Action-2C.

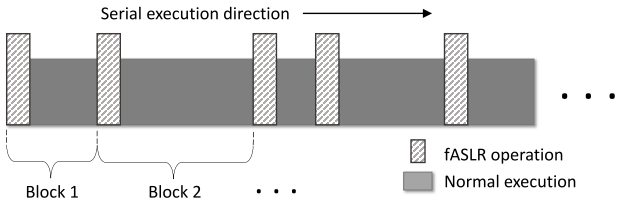


Fig. 7. Execution model of the NS app with fASLR.

We can conclude that the critical reliance set affected by Operation C remains valid. So, the NS app can execute correctly when Operation C is applied. ■

Lemma 4: The correctness of the NS app execution after any operation in runtime fASLR will hold until the next occurrence of a runtime fASLR operation, regardless of the in-between program execution.

Proof: So far, we have proved that the occurrence of any operation in runtime fASLR does not affect the validity of the critical reliance set. Hence, the correctness of the program execution holds at the exit of each runtime fASLR operation. Because only an operation in fASLR could possibly change the validity of the critical reliance set, such validity after an operation in fASLR holds until the next operation happens. Such validity will not be affected by the specific program execution during these two operations either because the critical reliance set cannot be changed by any program execution.

Recall that the validity of the critical reliance set is equivalent to the validity of the function calls and returns and, thus, equivalent to the overall correctness of the program execution. Based on the observations above and Lemmas 1–3, the validity of the critical reliance set prevails between consecutive runtime fASLR operations. We can deduce that the correctness of the NS app execution, not only holds after any operation in runtime fASLR but also holds until the right next occurrence of a runtime fASLR operation. Lemma 4 is proved. ■

Theorem 1: fASLR does not affect the correctness of the NS app execution.

Proof: As we analyzed at the beginning of this section, the execution of the NS app with runtime fASLR can be seen as inserting several fASLR operations into the original execution of the NS app, as it runs in a serial execution system as we assume in Assumption 2. Such an execution model is illustrated in Fig. 7. For each operation, we combine the operation with the normal program execution right after it, until the occurrence of the next operation, as an execution block. So, the whole program execution can be seen as a chain of such execution blocks.

We have proved that the NS app execution will remain correct when Operation A, B, or C completes in Lemmas 1–3 separately, and proved that the correctness of the NS app execution after any operation prevails until the right next operation occurs in Lemma 4. According to Assumption 1, the whole system is initialized correctly. Thus, the NS app execution is correct at the beginning of block 1 in Fig. 7, and remains

correct at the end of this block, which is also the beginning of block 2. Similarly, the correctness of the program execution prevails until block 2 completes. The correctness will not be affected by specific operations or specific normal executions. Therefore, no matter what operations and what normal executions compose the execution blocks after block 2, the program execution will remain correct until the program execution finishes. In other words, fASLR does not affect the correctness of the NS app execution. Theorem 1 is proved. ■

VI. Security and Performance Analysis

In this section, we first analyze the effectiveness of fASLR against ROP, a representative CRA. Entropy is computed to quantify the randomness of gadgets required for the ROP attack, which indicates the difficulty of guessing the gadget locations in a brute-force way. We also study time and memory overheads introduced by fASLR.

A. Effectiveness Against ROP

The prerequisite of ROP is that the adversary knows where the ROP gadgets are. In an fASLR enabled system, an adversary can only use ROP gadgets in randomized functions relocated to the RR. Gadgets in the NS app stored in flash are nonexecutable, so it is hard for adversaries to use them. Recall that any MPU violation triggers the HardFault exception. As discussed in Section III-C, the FRE validates the return address of the exception by using the FT. Therefore, the FRE is incapable of identifying exceptions triggered by a ROP attack if the adversary targets the entry point of a function since normal function calls will trigger such exceptions as well. In other words, the adversary will succeed in reusing a whole function as a gadget for ROP attack. However, such gadgets are often of very low quality [11], [12] containing too many instructions. It is almost impossible for an adversary to assemble a chain of gadgets with such low-quality gadgets to achieve certain malicious goal.

An adversary may also guess the addresses of randomized functions in a brute-force way. However, our runtime randomization approach rebases a function every time as long as it has not been loaded into RAM and achieves high randomization entropy as analyzed below.

B. Randomization Entropy

fASLR mitigates the brute-force guessing attack as follows.

1) fASLR restricts the number of functions that can be reused at a time. This is achieved by configuring the whole app image as nonexecutable. The only code snippets that can be utilized are functions relocated to the RR in the RAM.
2) Even if all the required gadgets can be found from the relocated functions, the adversary has to guess locations of all those functions at once. Formula (1) gives the total number (denoted as C) of possible function layouts in the RR

$$C = k!\binom{V+k}{k} \tag{1}$$

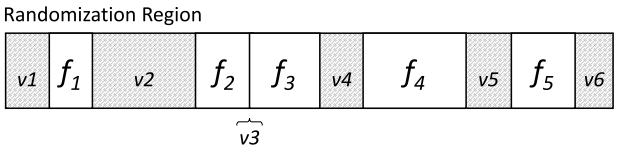


Fig. 8. Randomization layout with 5 loaded functions $f_1, f_2, \ldots, f_5$. The 5 functions form 6 vacancies $v_1, v_2, \ldots, v_6$ where free randomization units can be placed.

where k is the number of functions in the RR, and V is the size of unused randomization space divided by two since the ARMv8-M architecture only allows even function addresses. Note that the ARMv8-M architecture only allows an function to be loaded to an even address. Thus, the randomization space can be treated as V free randomization units and each unit is 2 bytes. We assume that the RR is large enough to accommodate k functions. If all free blocks are too small to fit the upcoming function, defragmentation can be applied. We calculate the maximum possibility of arranging k distinguished functions among V free units since from an attacker's perspective, any combination of k functions and V free units is possible. The combinations can be counted by the binomial coefficient $\binom{V+k}{k}$ multiplied by $k!$ because the k functions are distinguished.

Fig. 8 illustrates a randomization layout with five loaded functions. The shaded portions in the RR are free spaces. In this example, $k = 5$, and we suppose $V = 100$, there are $5!\binom{100+5}{5} = 1.159\text{e}+10$ combinations.

The probability of a layout is the reciprocal of C, i.e., $P = 1/C$. Formula (2) gives the entropy H of function randomization

$$H = -\sum_{i_1=1}^{C} P \log_2 P = -\sum_{i_1=1}^{C} \frac{1}{C} \log_2 \frac{1}{C} = \log_2 C. \tag{2}$$

C. Time Overhead

fASLR introduces runtime overhead when it hijacks a function call for function randomization via hardware exception. According to fASLR runtime mechanism, we consider three factors that affect the program runtime performance, namely, the number of function calls N_c that trigger HardFault exceptions, function randomization processing time for the ith function call denoted as $T_R(i)$, and hardware exception processing time T_E. Formula (3) gives the relationship between the time overhead TO and the three factors

$$\text{TO} = \sum_{i=1}^{N_c} (T_R(i) + T_E). \tag{3}$$

Intuitively, $T_R(i)$ would be much larger than T_E since T_R involves several time-consuming operations, such as memory write and table scanning, while T_E is accomplished by hardware. The overhead from the function randomization process primarily comes from the following aspects: 1) address verification, which involves looking up the FT; 2) function cleaning on demand, which looks up the call stack and cleans up finished functions; 3) randomization, which selects a free block to

rebase the callee; 4) function loading, which reads and writes the function body; and 5) function rewriting, which overwrites the destination of the call instruction with the entry point of loaded function.

D. Memory Overhead

The components of fASLR deployed in the SW include the BE code, FRE code, FT, LQ, and RL. The FT is a static table with three 4-byte attributes and its size is linear to the total number of functions in the NS app. The LQ and RL are dynamic data structures that contain function records and rewriting records, respectively. Each function record has four 4-bytes and one 1-byte metadata, and a rewriting record contains four 4-bytes data. The maximum number of records that the LQ may use at runtime is equal to the number of functions in the NS app, while the maximum number of rewriting records in the RL is the total number of call instructions. Formula (4) presents the size of the FT (i.e., MO_t), LQ (i.e., MO_q), and RL (i.e., MO_l)

$$MO_t = N_f \times 3 \times 4 = 12N_f \tag{4}$$

$$MO_q = N_f \times (4 \times 4 + 1) = 17N_f \tag{5}$$

$$MO_l = N_c \times 4 \times 4 = 16N_c \tag{6}$$

where N_f is the number of functions in the NS app, and N_c is the number of function calls in the NS app.

E. Size Requirement of the Randomization Region

fASLR will run out of memory (OOM) if a new function cannot fit into the RR and no function can be trimmed. To avoid such an OOM issue, there is a size requirement of the RR for a certain application. We define call path size as the total size of all functions on a call path. The RR should be no less than the largest call path of the application when fragmentation compaction is applied by the memory management scheme. We can calculate the size requirement by statically analyzing the application code and perform defragmentation to the RR if needed.

VII. Evaluation

In this section, we first present the experimental setup. We then present the evaluation of randomization entropy, runtime overhead, and memory overhead.

A. Experiment Setup

fASLR is implemented and deployed on the SAM L11 Xplained Pro Evaluation Kit, a MCU development board using the ARM Cortex-M23 core with TrustZone-M enabled. SAM L11 has a 64-kB flash and a 16-kB SRAM.

Software in SAM L11 is built with the GNU Arm Embedded Toolchain. User code, namely, the NS app code, is compiled with two flags, *-mlong-calls* and *-fno-jump-tables*, to eliminate instructions using relative addressing. We recompile the C library with the same compiler flags. A Python script runs during the compilation time to collect function metadata and saves them in the FT. fASLR program and the FT are

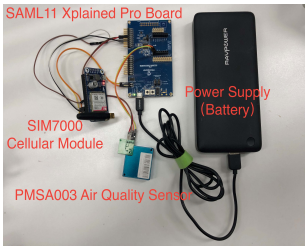


Fig. 9. Air quality monitoring device.

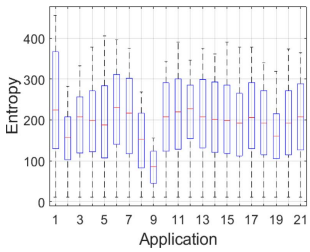


Fig. 10. Entropy distribution.

part of the Secure application placed in the SW flash, while the user app is deployed in the NSW flash.

We evaluate the performance of fASLR with 21 applications, including our own air quality monitoring system (*AirQualityMonitor*). The air quality monitoring device, as shown in Fig. 9, consists of a SAM L11 development board, a PMSA003 air quality sensor module, and a SIM7000 cellular module. The NS app in SAM L11 periodically receives air quality data from PMSA003 and sends the data to SIM7000, which then transfers the data to the AWS IoT platform via secure MQTT protocol. The other 20 apps, including the CoreMark benchmark [13], two microbenchmarks *Cache Test* and *Matrix Multiply* created based on [14], nine benchmarks of BEEBS (with the prefix *Beebs-*) [15], and eight SAM L11 demo apps (with the prefix *AS-*) obtained from Atmel Start [16].

B. Randomization Entropy

The entropy of function randomization changes dynamically when a function call occurs. We explore the entropy for all test applications. For each measured pair of k and V, we calculate the corresponding entropy of function randomization according to (1) and (2). Fig. 10 is the box plot demonstrating the entropy distribution for each app. The smallest average entropy is around 80 which is still considered to be large enough to defend against brute-force guessing.

C. Runtime Overhead

fASLR introduces runtime overhead since it intercepts every function call of the NS app for function randomization. We evaluate the time overhead by measuring and comparing the execution time of an application with and without fASLR. We use the internal `systick` timer of the Cortex-M core to record the execution time with precision of 0.01 s. Since the main program of an IoT application is usually a big loop, in the experiments we measure the execution time of 1000 loops

TABLE II
TOTAL EXECUTION TIME (IN SECOND) OF 1000 LOOPS AND OVERHEADS

Application	# of cleanings	w/o fASLR	with fASLR	Overhead
AirQualityMonitor	1	324.79	327.50	0.83%
CoreMark	4	15.62	15.78	1.02%
Cache Test	2	2.13	2.26	6.10%
Matrix Multiply	1	24.47	26.13	6.78%
AS-SecureDriver	1	12.56	12.64	0.64%
AS-ADC Event	2	12.41	12.54	1.04%
AS-Calendar	0	50.36	50.33	-0.06%
AS-Light Sensor	1	24.77	25.36	2.38%
AS-Low Power	0	14.60	14.60	0%
AS-ADP Hello	1	9.93	10.88	9.57%
AS-CRYA	1	6.79	7.35	8.25%
AS-TrustRAM	1	1.14	1.25	9.65%
Beebs-crc	1	7.44	7.73	3.90%
Beebs-aha-mont64	1	7.30	7.56	3.56%
Beebs-aha-compress	1	4.50	4.67	3.78%
Beebs-bs	1	0.28	0.29	3.57%
Beebs-bubblesort	1	0.33	0.35	6.06%
Beebs-compress	2	2.02	2.18	7.92%
Beebs-md5	2	0.42	0.44	4.76%
Beebs-levenshtein	1	17.42	17.97	3.16%
Beebs-edn	2	15.96	16.24	1.75%

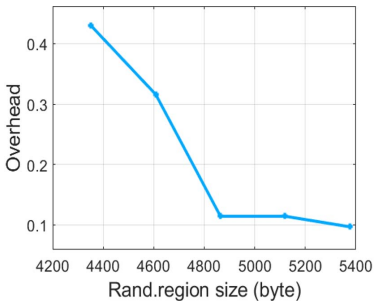


Fig. 11. Time overhead of *AS-TrustRAM* application versus RR size.

for each testing application. We comment out all `delay` functions inside the loop for a better estimation of time overhead introduced by fASLR. Table II presents the total execution time of 1000 loops for each application. The runtime overhead of fASLR is less than 10% for all apps, and 14 apps achieve time overheads below 5%. We also count for the occurrence of function cleaning for each app. The result shows that 19 apps have exhausted memory space during execution and triggered at least one function cleaning.

We evaluate the influence of the RR size on time overhead with *TrustRAM*, the app with the largest time overhead in Table II. Fig. 11 illustrates that fASLR tends to perform better with a larger RR. This is mainly because fASLR with a larger RR will less likely apply function cleaning and function loading during program execution.

We also examine the overheads of all 21 applications when fASLR is enabled with and without optimized memory management strategy. The test results demonstrated in Fig. 12 show that our optimization scheme achieves great improvements for all applications in terms of time overhead.

We further test three selected applications on another MCU, STM32 [17], to examine the overhead performance with and without fASLR, and enable comparisons with performances on SAML11. These three applications are those show relatively

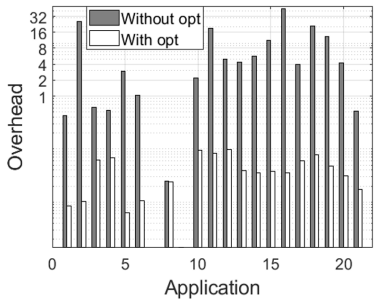


Fig. 12. Time overhead of test applications with versus without optimized memory management.

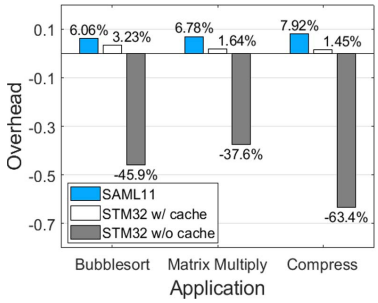


Fig. 13. Time overheads of three selected NS apps tested on SAML11, STM32 with cache, and STM32 without cache.

large overheads in Table II, namely, Matrix Multiply, Beebs-bubblesort, and Beebs-compress. Here, STM32 is a much more powerful TrustZone-M-enabled microcontroller with 512-kB flash memory, 256-kB SRAM, and cache support. Specifically, the SRAM of STM32 is 16 times the size of SAML11. The overheads of these selected apps on SAML11, STM32 with cache enabled, and STM32 without cache enabled are compared in Fig. 13. For all applications, the overhead performances are better on STM32 than on SAML11. This is because STM32 has a much larger SRAM, which provides more randomization space. With the same reason as aforementioned, this leads to less chances to evoke fASLR operations such as function cleaning, thus reducing overheads.

We then examine the overhead performance on STM32 with and without cache. It is interesting to observe that without cache, enabling fASLR speeds up the program execution significantly, thus presenting negative overheads. This is because although fASLR operations inevitably introduce overheads, it also executes code in RAM. In the case without cache, execution in RAM leads to a large performance improvement that not only compensates the overhead from fASLR operations but also yields execution speedup. This reveals another benefit of fASLR besides function randomization, that fASLR could also generate an execution speedup in certain systems. In the case of STM32 with cache, because the cache mechanism can also speedup code execution in flash, the benefit from enabling fASLR becomes much less apparent. In our experiment, since the overhead introduced by fASLR operations becomes dominant over the benefit of execution in RAM, the overheads are positive. Overall results in Fig. 13 consistently demonstrate that fASLR achieves less overhead with more powerful devices.

TABLE III
NS APP SIZE (IN BYTE) AND OVERHEADS

Application	# of functions	Size of the rand. region	App size w/o fASLR	App size with fALSR	Overhead
AirQualityMonitor	148	6144	41092	43036	4.73%
CoreMark	174	6144	46048	47648	3.47%
Cache Test	140	5632	40228	41844	4.02%
Matrix Multiply	145	6144	40728	42404	4.12%
AS-SecureDriver	139	6144	39544	41184	4.15%
AS-ADC Event	173	6144	43036	44640	3.73%
AS-Calendar	97	6144	36780	36808	0.08%
AS-Light Sensor	132	6144	40496	40528	0.08%
AS-Low Power	67	6144	34136	34164	0.08%
AS-ADP Hello	99	6144	38072	38316	0.64%
AS-CRYA	143	7168	41368	43012	3.97%
AS-TrustRAM	142	6144	39896	41500	4.02%
Beebs-crc	138	6144	39944	41492	3.88%
Beebs-aha-mont64	142	6144	40476	42028	3.83%
Beebs-aha-compress	140	6144	39944	41492	3.88%
Beebs-bs	137	6144	39344	40896	3.94%
Beebs-bubblesort	137	6144	39932	40892	2.40%
Beebs-compress	143	5632	40808	42360	3.80%
Beebs-md5	137	6144	41552	43100	3.73%
Beebs-levenshiein	138	6144	39708	41348	4.13%
Beebs-edn	144	6144	42112	43736	3.86%

D. Memory Overhead

For each tested application, we measure the total number of functions and the memory overhead of NS apps before and after deploying fASLR, as illustrated in Table III. In the SW, the code overhead is caused by the program of fASLR with a fixed code size of 3.45 KB, and the data overhead is mainly introduced by the static FT, dynamic LQ and RL and, thus, depends on the number of functions in the NS app. The size of the NS app is changed because of the compilation with specific compiler flags. Table III shows little memory overhead below 5% for all tests.

VIII. DISCUSSION

In this section, we discuss how fASLR can be used together with other security defense mechanisms to protect runtime execution of MCU-based IoT devices.

Data Execution Prevention (DEP): fASLR can work with DEP, which is supported by many commercial MCUs. DEP prevents execution from the data region, such as stack and heap, and can effectively mitigate against code injection attacks. It is widely believed that combining ASLR with DEP is more effective than using ASLR or DEP alone [18].

The ARM's DEP technology is called execute-never (XN) bits [19] which are used to mark certain memory regions as nonexecutable. Any attempt of executing an instruction from the region tagged as XN will result in a HardFault exception. Our scheme requires an executable RR for runtime function randomization and execution. Therefore, any other memory region can be set to XN for security concerns. Although the violation of XN triggers the same fault exception in ARMv8-M architecture, it is trivial for fASLR to distinguish such exceptions at the verification stage by judging if the exception return address is within the application code region or not since fASLR only handles exceptions triggered by attempts of executing functions in the application region.

Control Flow Integrity: Control flow integrity (CFI) is another countermeasure against CRAs. Unlike ASLR which makes the locations of gadgets unpredictable, CFI prevents the control flow hijack by monitoring any control flow changes, ensuring branch instructions branch as intended. CFI technique is not as mature as ASLR and has not been widely used in industry due to its runtime overhead. Although some CFI implementations have been deployed, they cannot mitigate all CRAs independently. For example, Windows 10 supports both ASLR and control flow guard (Windows' implementation of CFI) as exploit protection mechanisms [20]. CaRE [21] is a CFI implementation for IoT devices based on the Cortex-M processors. CaRE intercepts any direct and indirect branches by replacing those instructions with the supervisor call (*svc*) so that the control flow is trapped to its monitor code.

fASLR can work with CFI-based mechanisms with a few modifications. Once a function is randomized or cleaned by fASLR, the control flow graph or other assisted data structures which reflect the control flow should be modified by the runtime fASLR correspondingly.

IX. RELATED WORK

Compared to conventional ASLR which rebases the whole executable, fine-grained ASLR strategies achieve higher randomization entropy, change the structure of the executable and, thereby, are considered to be more effective against CRAs and brute-force attacks. Code randomization can have different granularities [22] based on what is diversified. ASLP [23] is a code permutation scheme which applies function-level permutation to the code segment and object permutation to the data segment without the knowledge of source code. In [24], the original binary code is partitioned into small blocks of which the addresses are decided when the application is loaded. Xifer [25] achieves fine-grained randomization by splitting code into arbitrary small pieces, spreading the code pieces within the address space, and rewriting the code to preserve

its semantics. ILR [26] is an instruction-based randomization scheme which relocates every instruction thereby achieving high randomization entropy.

Although fine-grained ASLR is effective in mitigating a single-memory disclosure attack, Snow *et al.* [3] found that multiple memory disclosures are promising in bypassing fine-grained randomization techniques. Motivated by this observation, they introduce an attack framework which bypasses fine-grained randomization via just-in-time code reuse (JIT-ROP). With the knowledge of a single-memory disclosure, the framework is able to excavate memory contents of multiple memory pages at runtime, search and assemble gadgets on-the-fly, and then launch CRA. Accordingly a fine-grained randomization approach named Isomeron [27] is proposed as the countermeasure to JIT-ROP attacks. Combining fine-grained ASLR with execution path randomization, Isomeron makes any gadgets unpredictable. Specifically, it generates diversified application code using fine-grained ASLR, and loads both original code and diversified code to the virtual address space at runtime. During execution, a coin-flip decision is made upon each function call to select the destination from either original or diversified code. Related research has been performed to overcome newly emerging CRAs and meet increasing compatibility requirements [28]–[31].

Shi *et al.* [5] leveraged the TrustZone-M hardware extension to enable a function-level ASLR scheme for ARM-based MCUs. The proposed system loads the NS code to NS RAM and periodically reordering all functions at runtime. Compared with our work, this scheme loads the whole application code to RAM. Instead of loading the whole NS app code, our mechanism—fASLR—only loads functions in use and cleans up finished functions from RAM at runtime. fASLR requires smaller RAM and achieves a larger randomization entropy for resource-constrained IoT devices. Shi *et al.* [5] rewrote binaries of the NS code offline and introduces a code size overhead of about 10%–15%, while fASLR has a code size overhead below 5%.

X. Conclusion

In this article, we propose fASLR for runtime software security of resource-constrained IoT devices, particularly those based on microcontrollers. fASLR leverages hardware-based security provided by the TrustZone-M technique as the trust anchor. It uses MPU and prevents direct code execution of the application image in the NSW flash. Instead, it traps control flow in an exception handler and relocates functions to be executed to a randomly selected location within the RAM. A memory management strategy was designed for allocating and cleaning up functions in the RR. We also optimized the baseline function cleaning scheme to largely decrease runtime overhead. fASLR is user friendly and only requires a user compiling the app with specific flags. We formally prove that fASLR will not affect the correctness of the NS app execution. We implemented fASLR with a TrustZone-M-enabled MCU—SAM L11. fASLR achieves high randomization entropy with acceptable overheads. We will release fASLR to GitHub for broad adoption and refine the implementation to further reduce the overhead.

References

[1] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer, "Defeating memory corruption attacks via pointer taintedness detection," in *Proc. Int. Conf. Depend. Syst. Netw. (DSN)*, Jul. 2005, pp. 378–387. [Online]. Available: https://doi.org/10.1109/DSN.2005.36

[2] T. K. Bletsch, X. Jiang, and V. W. Freeh, "Mitigating code-reuse attacks with control-flow locking," in *Proc. 27th Annu. Comput. Security Appl. Conf. (ACSAC)*, Orlando, FL, USA, Dec. 2011, pp. 353–362. [Online]. Available: https://doi.org/10.1145/2076732.2076783

[3] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Proc. IEEE Symp. Security Privacy (SP)*, Berkeley, CA, USA, May 2013, pp. 574–588. [Online]. Available: https://doi.org/10.1109/SP.2013.45

[4] "TrustZone for cortex-M." ARM. [Online]. Available: https://www.arm.com/why-arm/technologies/trustzone-for-cortex-m (Accessed: Jun. 18, 2022).

[5] J. Shi, L. Guan, W. Li, D. Zhang, P. Chen, and P. Chen, "HARM: Hardware-assisted continuous re-randomization for microcontrollers," in *Proc. IEEE Eur. Symp. Security Privacy (EuroS P)*, 2022, pp. 520–536.

[6] X. Shao, L. Luo, Z. Ling, H. Yan, Y. Wei, and X. Fu, "fASLR: Function-based ASLR for resource-constrained IoT systems," in *Proc. ESORICS*, 2022.

[7] "freeRTOS—Market Leading RTOS (Real Time Operating System for Microcontrollers)." [Online]. Available: https://www.freertos.org/ (Accessed: Jun. 18, 2022).

[8] "ARMv8-M Fault Handling and Detection." ARM. [Online]. Available: https://developer.arm.com/documentation/100691/0200/Fault-exceptions (Accessed: Jun. 18, 2022).

[9] J. Yiu, "Chapter 2—Getting started with cortex-M programming," in *Definitive Guide to Arm® Cortex®-M23 and Cortex-M33 Processors*, J. Yiu, Ed. Cambridge, MA, USA: Newnes, 2021, pp. 19–51. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128207352000020

[10] S. M. Hejazi, C. Talhi, and M. Debbabi, "Extraction of forensically sensitive information from windows physical memory," *Digit. Investig.*, vol. 6, pp. S121–S131, Sep. 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287609000474

[11] A. Follner, A. Bartel, and E. Bodden, "Analyzing the gadgets," in *Proc. Int. Symp. Eng. Secure Softw. Syst.*, 2016, pp. 155–172.

[12] M. D. Brown and S. Pande, "Is less really more? why reducing code reuse gadget counts via software debloating doesn't necessarily indicate improved security," 2019, *arXiv:1902.10880*.

[13] "CPU Benchmark—MCU Benchmark—CoreMark." Embedded Microprocessor Benchmark Consortium. [Online]. Available: https://www.eembc.org/coremark/ (Accessed: Jun. 18, 2022).

[14] H. Quinn. "Microcontroller Benchmark Codes for Radiation Testing." Los Alamos National Security. [Online]. Available: https://github.com/lanl/benchmark_codes (Accessed: Jun. 18, 2022).

[15] J. Pallister, S. Hollis, and J. Bennett, "BEEBS: Open benchmarks for energy measurements on embedded platforms," 2013, *arXiv:1308.5174*.

[16] "ATMEL Start." Microchip. [Online]. Available: https://start.atmel.com/ (Accessed: Jun. 18, 2022).

[17] "STM32L562E-DK—Discovery Kit With STM32L562QE MCU." STMicroelectronics. [Online]. Available: https://www.st.com/en/evaluation-tools/stm32l562e-dk.html (Accessed: Jun. 18, 2022).

[18] SWIAT. "On the Effectiveness of DEP and ASLR." Microsoft Security Response Center. 2010. [Online]. Available: https://msrc-blog.microsoft.com/2010/12/08/on-the-effectiveness-of-dep-and-aslr/

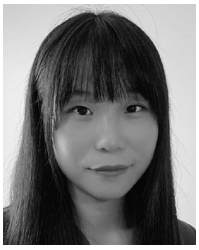
[19] "ARM11 MPCore Processor Technical Reference Manual." ARM. [Online]. Available: https://developer.arm.com/documentation/ddi0360/f/memory-management-unit/memory-access-control/execute-never-bits (Accessed: Jun. 18, 2022).

[20] "Apply Mitigations to Help Prevent Attacks Through Vulnerabilities." Microsoft Docs. 2021. [Online]. Available: https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/exploit-protection?view=o365-worldwide

[21] T. Nyman, J.-E. Ekberg, L. Davi, and N. Asokan, "CFI CaRE: Hardware-supported call and return enforcement for commercial microcontrollers," in *Proc. Int. Symp. Res. Attacks Intrusions Defenses*, 2017, pp. 259–284.

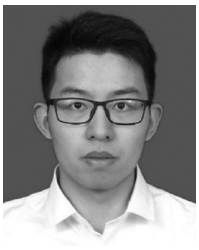
[22] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proc. IEEE Symp. Security Privacy*, 2014, pp. 276–291.

[23] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software," in *Proc. 22nd Annu. Comput. Security Appl. Conf. (ACSAC)*, 2006, pp. 339–348.
[24] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proc. ACM Conf. Comput. Commun. Security*, 2012, pp. 157–168.
[25] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and ARM," in *Proc. 8th ACM SIGSAC Symp. Inf. Comput. Commun. Security*, 2013, pp. 299–310.
[26] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *Proc. IEEE Symp. Security Privacy*, 2012, pp. 571–585.
[27] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming," in *Proc. NDSS*, 2015, pp. 1–15.
[28] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis, and M. Polychronakis, "Compiler-assisted code randomization," in *Proc. IEEE Symp. Security Privacy (SP)*, 2018, pp. 461–477.
[29] F. Xuewei, W. Dongxia, L. Zhechao, K. Xiaohui, and Z. Gang, "Enhancing randomization entropy of x86-64 code while preserving semantic consistency," in *Proc. IEEE 19th Int. Conf. Trust Security Privacy Comput. Commun. (TrustCom)*, 2020, pp. 1–12.
[30] S. Priyadarshan, H. Nguyen, and R. Sekar, "Practical fine-grained binary code randomization," in *Proc. Annu. Comput. Security Appl. Conf.*, 2020, pp. 401–414.
[31] X. Wang, S. Yeoh, R. Lyerly, P. Olivier, S.-H. Kim, and B. Ravindran, "A framework for software diversification with ISA heterogeneity," in *Proc. 23rd Int. Symp. Res. Attacks Intrusions Defenses (RAID)*, 2020, pp. 427–442.



Lan Luo received the B.S. degree in electrical engineering from the Civil Aviation University of China, Tianjin, China, in 2015, and the M.S. degree in computer engineering and the Ph.D. degree in computer science with the University of Central Florida, Orlando, FL, USA, in 2018 and 2022, respectively.

Her research interests mainly cover security and privacy of Internet of Things, security of embedded system, network and software security, and trustworthy computing.



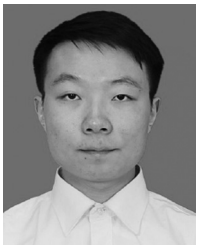
Xinhui Shao received the B.S. degree in communication engineering from Shanghai University, Shanghai, China, in 2019. He is currently pursuing the master's degree in cyber science and engineering with Southeast University, Nanjing, China.

His current research interests include Internet of Things and privacy and security.

Zhen Ling (Member, IEEE) received the B.S. degree from Nanjing Institute of Technology, Nanjing, China, in 2005, and the Ph.D. degree in computer science from Southeast University, Nanjing, in 2014.

He is a Professor with the School of Computer Science and Engineering, Southeast University. His research interests include network security, privacy, and Internet of Things.

Prof. Ling won the ACM China Doctoral Dissertation Award in 2014 and the China Computer Federation Doctoral Dissertation Award in 2015.



Huaiyu Yan received the B.S. degree in software engineering from Southeast University, Nanjing, China, in 2019, where he is currently pursuing the Ph.D. degree in computer science and engineering.

His current research interests include Internet of Things and privacy and security.

Yumeng Wei is currently pursuing the B.S. degree in cyberspace security with Southeast University, Nanjing, China.

Her research interests include software and network security of Internet of Things devices.

Xinwen Fu (Senior Member, IEEE) received the B.S. degree in electrical engineering from Xi'an Jiaotong University, Xi'an, China, in 1995, the M.S. degree in electrical engineering from the University of Science and Technology of China, Hefei, China, in 1998, and the Ph.D. degree in computer engineering from Texas A&M University, College Station, TX, USA, in 2005.

He is a Professor with the Department of Computer Science, University of Massachusetts Lowell, Lowell, MA, USA. He was a tenured Associate Professor with the Department of Computer Science, University of Central Florida, Orlando, FL, USA. He has published at prestigious conferences, including the four top computer security conferences (Oakland, CCS, USENIX Security, and NDSS), and journals, such as ACM/IEEE TRANSACTIONS ON NETWORKING and IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING. His current research interests are in computer and network security and privacy.

Dr. Fu spoke at various technical security conferences including Black Hat.