# fASLR: Function-Based ASLR for Resource-Constrained IoT Systems

Xinhui Shao[1], Lan Luo[3], Zhen Ling[2(✉)], Huaiyu Yan[2], Yumeng Wei[1], and Xinwen Fu[3,4]

[1] School of Cyber Science and Engineering, Southeast University, Nanjing, China
{xinhuishao,yumeng5}@seu.edu.cn
[2] School of Computer Science and Engineering, Southeast University, Nanjing, China
{zhenling,huaiyu_yan}@seu.edu.cn
[3] University of Central Florida, Orlando, USA
lukachan@knights.ucf.edu
[4] University of Massachusetts Lowell, Lowell, USA
xinwen_fu@uml.edu

**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 return-to-libc. 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) for IoT runtime security utilizing the ARM TrustZone-M technique and the memory protection unit (MPU). fASLR loads a function from the flash and randomizes its entry address in a randomization region in RAM when the function is called. We design novel mechanisms on cleaning up finished functions from the RAM and memory addressing to deal with the complexity of function relocation and randomization. Compared with 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. fASLR achieves high randomization entropy and incurs runtime overhead of less than 10%.

**Keywords:** Function-based randomization · IoT · ASLR · CRA · ROP

## 1 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++, and is vulnerable to memory corruption attacks [5]. One typical memory corruption attack is the code reuse attack (CRA), which hijacks the control flow and reuses the application code [3]. Memory corruption

X. Shao and L. Luo—Contribute equally to this work.


V. Atluri et al. (Eds.): ESORICS 2022, LNCS 13555, pp. 531–548, 2022.
https://doi.org/10.1007/978-3-031-17146-8_26

attacks and defenses have been actively studied for mainstream operating systems such as Windows, macOS, Lniux, Android and iOS.

In this paper, we focus on defending against memory corruption attacks for resource-constrained IoT devices, particularly those running on micrcontrollers (MCUs). It is an intuitive idea to port existing security schemes to IoT platforms. We study the use of ASLR in memory-constrained IoT devices to mitigate CRAs such as the return-oriented programming (ROP) 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 [20] 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 [2] to protect MCU-based IoT devices from code reuse attacks that require knowledge of locations of executable code snippets, such as ROP and JOP. The runtime fASLR is located in the Secure World (SW) of TrustZone-M. The application is in the Non-secure World (NSW) flash and denoted as the NS app, which is protected by the memory protection unit (MPU). When a function call occurs, MPU redirects the function call to the runtime fASLR for callee randomization. Compared to the most recent related work [19] 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.

Our major contributions are summarized as follows. (i) 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. (ii) Novel schemes are designed for fASLR to perform memory management and addressing. Finished functions are efficiently removed from RAM when there is no RAM for storing more functions. We carefully address the issue of addressing since functions are randomly moved around. (iii) We implement fASLR with a TrustZone-M enabled MCU, SAM L11, and validate the feasibility and performance of fASLR with 21 applications. fASLR incurs a runtime overhead of less than 10% for all the applications.

## 2 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 [14] based on what is diversified. ASLP [12] is a code permutation scheme which applies function level permutation to the code segment and object permutation to the data segment without knowledge of source code. In [22], the original binary code is partitioned into small blocks of which the addresses are decided

when the application is loaded. Xifer [8] 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 [11] is an instruction-based randomization scheme which relocates every instruction thereby achieving high randomization entropy.

Snow et al. [20] 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 code reuse attack. Accordingly a fine-grained randomization approach named Isomeron [7] is proposed as the countermeasure to JIT-ROP attacks. Combining fine-grained ASLR with execution path randomization, Isomeron makes any gadgets unpredictable. Related research has been performed to overcome newly emerging code reuse attacks and meet increasing compatibility requirements [13,17,21,23].

Shi et al. [19] leverage 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 promptly cleans up finished functions from RAM at runtime. fASLR requires smaller RAM and achieves larger randomization entropy for resource constrained IoT devices. [19] rewrites 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%.

## 3 System Architecture

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

### 3.1 Threat Model

fASLR leverages ARM Cortex-M processors and hardware-based techniques including TrustZone-M, memory protection unit (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 Secure World (SW) and the Non-secure World (NSW).

We assume a TrustZone-M enabled device has the following security features. (i) 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. (ii) The NS app is located at a fixed address in the NS flash and is executed from the flash (instead of RAM) by default. (iii) The device supports the memory protection mechanisms such as the MPU.

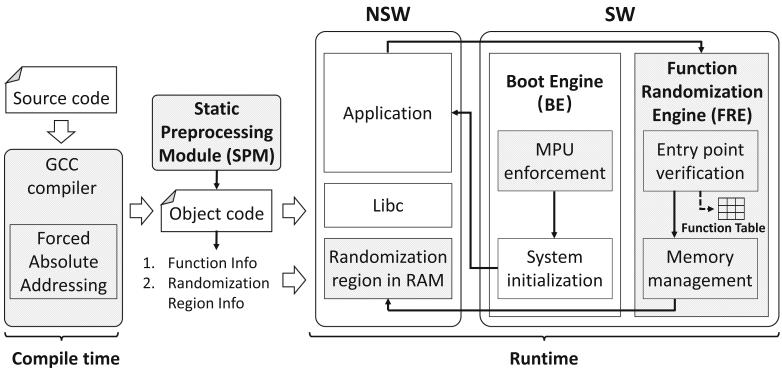


**Fig. 1.** fASLR architecture.

We assume an adversary has the following capabilities. (i) The NS app may be subject to CRAs such as the ROP attack. (ii) The adversary can obtain the binary of the NS app, disassemble the binary, and obtain code gadgets for CRAs.

### 3.2 Design Goals

fASLR is designed to achieve the following goals.

- **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.
- **Usability**. The scheme shall be user-friendly and will not add much burden of programming.
- **Low overhead**. The proposed scheme cannot introduce large overhead in terms of time and space and affect the NS app performance much.

### 3.3 System Architecture

As illustrated in Fig. 1, fASLR has three key components. (i) the ***Static Preprocessing Module (SPM)*** for compilation time preparation, (ii) The ***Boot Engine (BE)*** for boot time configuration, and (iii) The ***Function Randomization Engine (FRE)*** for runtime function-level randomization.

**Static Preprocessing Module (SPM).** The *SPM* serves two major purposes: (i) Creating the *Function Table*. 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 `.debug_frame` section of the ELF file. Function information is recorded in a data structure called *Function Table*. (ii) Profiling randomization region. Extracting RAM usage information

from the compiler output file, the *SPM* determines the size and location of the largest unused RAM space as the *randomization region*. Users can also set a smaller randomization region by manually modifying related configurations.

**Boot Engine (BE).** 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 non-executable for two purposes: (i) MPU prevents the NS app in the NS flash from being exploited by CRAs. (ii) Once the NS code is set as non-executable, any attempt to execute the NS code triggers a hardware exception, which is handled by the `HardFault` exception handler in the SW [1].

**Function Randomization Engine (FRE).** 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 *Function Table* until there is a match. After a match, the *FRE* obtains the size of the corresponding function from the *Function Table* 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 *randomization region*. After the function randomization, we need to carefully handover the control flow from the exception handler to the relocated function. We overwrite 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.

### 3.4 Workflow

**Offline—Compilation and Flashing.** After the NS app is compiled and linked by the GCC compiler at compile time, the *SPM* creates the *Function Table* offline according to the NS app ELF file. The *Function Table* 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.** Figure 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 [24]. 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 *randomization region* in the RAM for execution. During the execution of *main()* function, the control flow can divert to a callee on the MPU-protected flash only if the callee is invoked by *main()*.

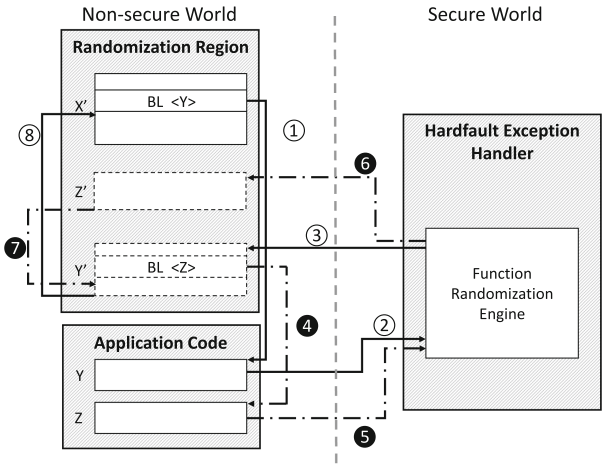


**Fig. 2.** Program flow of function $X$, $Y$ and $Z$. For any function $F$ in the NS app, we use $F'$ to represent its corresponding copy in the randomization region.

In a fASLR-enabled system, when a function call occurs in the randomization region, 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 a fASLR enabled system, a callee returns to the relocated caller in the randomization region since that is where the function call really occurs.

Figure 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 randomization region and the program flow starts from the relocated function $X'$. When $X'$ calls $Y$, the

attempt of executing $Y$ (①) results in a MPU violation exception (②), handled by the *FRE* inside the `HardFault` exception handler. $Y$ is then relocated to the randomization region 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.

### 3.5 Challenges

A practical fASLR faces the following challenges. We address these challenges in detail in Sect. 4.

**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 finished functions. *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* needs to distinguish finished functions and select an appropriate timing to trim them from the randomization region.

**Memory Addressing.** A function in the randomization region 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 a 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.

## 4 Memory Management and Addressing

In this section, we discuss the challenges of fASLR and present our key memory management and addressing schemes.

### 4.1 Memory Management

We devise a function cleaning scheme that dynamically cleans up finished functions from the *randomization region* with the following mechanisms: (i) *Call stack unwinding*. Finished functions are found through unwinding the Non-secure call stack. (ii) *Cleaning on demand*. Finished functions are cleaned up only if the available randomization region space is not large enough for the callee. (iii) *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.

**Algorithm 1.** Call Stack Unwinding

```
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
```

A queue structure named *Loading Queue* is used to store meta data, denoted as function record, of all loaded functions in the RAM. The *FRE* pushes a function record into the *Loading Queue* when an unloaded function is called. A function record includes the following information of the callee, (i) *loadAddress*– the new entry point of the callee in the randomization region, (ii) *size* of the callee, and (iii) *stack frame size* of the callee.

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 *Loading Queue*, 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 mechanism. 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 of each function is extracted offline by the *SPM* from `.debug_frame` section of the ELF file and stored in the *Function Table*.

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 *Loading Queue*. 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 1 presents our stack unwinding procedure.

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 randomization region, 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 randomization region 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 *Loading Queue* as unfinished. According to the marked *Loading Queue*, the *FRE* disposes all finished functions and updates the *Loading Queue*. 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* 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*. Such records are used to recover the call instructions with callees' original flash entry points before function cleaning, since the loaded callees of those call instructions might be disposed.

**Memory Fragmentation Management.** In the randomization region, 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 contains the size of the block and the pointer which points the next free block. The payload region is used to stores the randomized function.

When the system starts, fASLR initializes the whole randomization region 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. Figure 3 (a) and (b) illustrate 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.

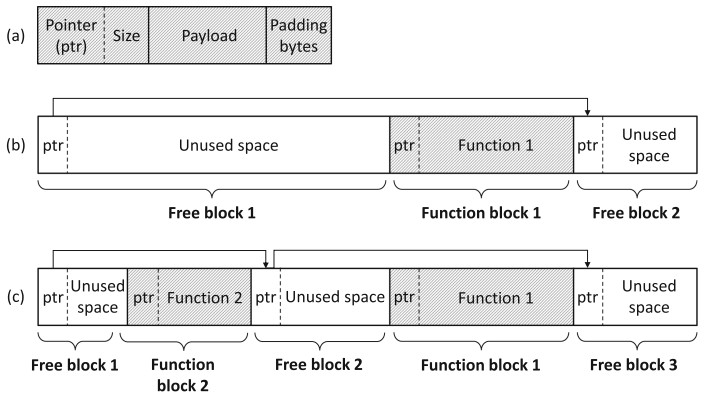


**Fig. 3.** Memory management. (a) Structure of the randomization region; (b) Memory layout before loading Function 2; (c) Memory layout after loading and randomizing Function 2

## 4.2 Memory Addressing

Control flow instructions using relative addresses in the ARMv8-M instruction set include *B* (branch), *BL* (branch with link), and *CBNZ/CBZ* (conditional branches), among which the *BL* (branch with link) 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 change 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, 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.

## 5 Security and Performance Analysis

In this section, we analyze the effectiveness of fASLR against ROP, a representative code reuse attack. 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.

### 5.1 Effectiveness Against ROP

The prerequisite of ROP is that the adversary knows where the ROP gadgets are. In a fASLR enabled system, an adversary can only use ROP gadgets in randomized functions relocated to the randomization region. Gadgets in the NS app stored in flash are non-executable, so it is hard for adversaries to use them. Recall that any MPU violation triggers the `HardFault` exception. As discussed in Sect. 3.3, the *FRE* validates the return address of the exception by using the *Function Table*. 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 [4,9] 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.

### 5.2 Randomization Entropy

fASLR mitigates the brute-force guessing attack as follows. (i) fASLR restricts the number of functions that can be reused at a time. This is achieved by configuring the whole app image as non-executable. The only code snippets that can be utilized are functions relocated to the randomization region in the RAM. (ii) 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 randomization region.

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

where $k$ is the number of functions in the randomization region, and $V$ is the size of unused randomization space divided by two since the ARMv8-M architecture only allows even function addresses. Note 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 the randomization region 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. For example, if $k = 5$ and $V = 100$, there are $5!\binom{100+5}{5} = 1.159\text{e+}10$ combinations in total.

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

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

### 5.3 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 $i$th 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.

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

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: (i) address verification, which involves looking up the *Function Table*; (ii) function cleaning on demand, which looks up the call stack and cleans up finished functions; (iii) randomization, which selects a free block to rebase the callee; (iv) function loading, which reads and writes the function body; (v) function rewriting, which overwrites the destination of the call instruction with the entry point of loaded function.

### 5.4 Memory Overhead

The components of fASLR deployed in the SW include the *BE* code, *FRE* code, *Function Table*, *Loading Queue*, and *Rewriting List*. The *Function Table* 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 *Loading Queue* and *Rewriting List* are dynamic data structures that contain function records and rewriting records respectively. Each function record has three 4-bytes metadata and a rewriting record contains double 4-bytes data. The maximum number of records that the *Loading Queue* may contain at runtime is equal to the number of functions in the NS app, while the maximum number of rewriting records in the *Rewriting List* is the total number of call instructions. Formula (4) presents the size of the *Function Table* (i.e., $MO_t$), *Loading Queue* (i.e., $MO_q$), and *Rewriting List* (i.e., $MO_l$),

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

$$MO_q = N_f \times 3 \times 4 = 12N_f, \tag{5}$$

$$MO_l = N_c \times 2 \times 4 = 8N_c, \tag{6}$$

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.

### 5.5 Size Requirement of the Randomization Region

fASLR will run out of memory (OOM) if a new function cannot fit into the randomization region and no function can be trimmed. To avoid such an OOM issue, there is a size requirement of the randomization region for a certain application. We define call path size as the total size of all functions on a call path. The randomization region 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 randomization region if needed.

## 6 Evaluation

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

### 6.1 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 to make C functions compatible with fASLR. A Python script runs during the compilation time to collect function metadata and saves them in the *Function Table*. fASLR program and the *Function Table* are 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. 4, 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 twenty apps including the CoreMark benchmark [6], two micro benchmarks *Cache Test* and *Matrix Multiply* created based on [18], nine benchmarks of BEEBS [16], and eight SAM L11 demo apps obtained from Atmel Start [15].


**Fig. 4.** Our air quality monitoring device developed with SAML11

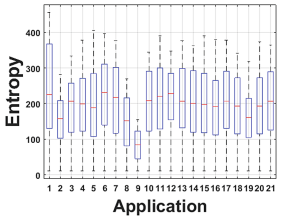


**Fig. 5.** Entropy distribution

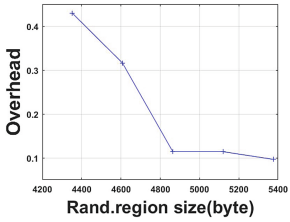


**Fig. 6.** Time overhead of *TrustRAM* application vs. randomization region size

### 6.2 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 Formulas (1) and (2). Figure 5 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.

### 6.3 Runtime Overhead

fASLR introduces runtime overhead since it intercepts every non-rewritten 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 0.01s. Since the main program of an

IoT application is usually a big loop, in the experiments we measure the execution time of 1000 loops for each testing application. We comment out all `delay` functions inside the loop for better estimation of time overhead introduced by fASLR. Table 1 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 also evaluate the influence of the randomization region size on time overhead with *TrustRAM*, the app with the largest time overhead in Table 1. Figure 6 illustrates that fASLR tends to perform better with a larger randomization region. This is mainly because fASLR with a larger randomization region will less likely apply function cleaning and function loading during program execution.

**Table 1.** Total execution time (in second) of 1000 loops and overheads.

| Application | # of cleanings | Baseline | 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% |
| SecureDriver | 1 | 12.56 | 12.64 | 0.64% |
| ADC event system | 2 | 12.41 | 12.54 | 1.04% |
| Calendar | 0 | 50.36 | 50.33 | –0.06% |
| Light sensor | 1 | 24.77 | 25.36 | 2.38% |
| Low power | 0 | 14.60 | 14.60 | 0% |
| ADP Hello | 1 | 9.93 | 10.88 | 9.57% |
| CRYA | 1 | 6.79 | 7.35 | 8.25% |
| 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% |

### 6.4 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 2. 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 *Function Table*, dynamic *Loading Queue* and *Rewriting List*, 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 2 shows little memory overhead below 5% for all tests. It can be observed the app sizes in Table 2 are larger than the RAM size (16 KB). This shows the strength of fASLR, which can run an applications that is too large to be completely loaded into RAM compared with related work [19].

**Table 2.** NS app size (in byte) and overheads.

| Application | # of Functions | Size of rand. region | App size (baseline) | 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% |
| SecureDriver | 139 | 6144 | 39544 | 41184 | 4.15% |
| ADC Event System | 173 | 6144 | 43036 | 44640 | 3.73% |
| Calendar | 97 | 6144 | 36780 | 36808 | 0.08% |
| Light Sensor | 132 | 6144 | 40496 | 40528 | 0.08% |
| Low Power | 67 | 6144 | 34136 | 34164 | 0.08% |
| ADP Hello | 99 | 6144 | 38072 | 38316 | 0.64% |
| CRYA | 143 | 7168 | 41368 | 43012 | 3.97% |
| 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% |

## 7 Conclusion

In this paper, we propose fASLR, a function-based ASLR scheme 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 Non-secure world 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 is designed for allocating and cleaning up functions in the randomization region. fASLR is user friendly and only requires a user compiling the app with specific flags. We implement 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.

**Acknowledgment.** This research was supported in part by National Key R&D Program of China 2018YFB2100300, National Natural Science Foundation of China Grant Nos. 62022024, 61972088, 62072103, 62102084, 62072102, 62072098, and 61972083, by US National Science Foundation (NSF) Awards 1931871, 1915780, and US Department of Energy (DOE) Award DE-EE0009152, by Jiangsu Provincial Natural Science Foundation for Excellent Young Scholars Grant No. BK20190060, Jiangsu Provincial Natural Science Foundation of China Grant No. BK20190340, Jiangsu Provincial Key Laboratory of Network and Information Security Grant No. BM2003201, Key Laboratory of Computer Network and Information Integration of Ministry of Education of China Grant Nos. 93K-9, and Collaborative Innovation Center of Novel Software Technology and Industrialization. Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

## References

1. ARM. Armv8-m fault handling and detection
2. ARM. Trustzone for cortex-m
3. Bletsch, T.K., Jiang, X., Freeh, V.W.: Mitigating code-reuse attacks with control-flow locking. In: Zakon, R.H., McDermott, J.P., Locasto, M.E. (eds.) Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5–9 December 2011, pp. 353–362. ACM (2011)
4. Brown, M.D., Pande, S.: Is less really more? Why reducing code reuse gadget counts via software debloating doesn't necessarily indicate improved security. arXiv preprint arXiv:1902.10880 (2019)
5. Chen, S., Xu, J., Nakka, N., Kalbarczyk, Z., Iyer, R.K.: Defeating memory corruption attacks via pointer taintedness detection. In: 2005 International Conference on Dependable Systems and Networks (DSN 2005), 28 June–1 July 2005, Yokohama, Japan, Proceedings, pp. 378–387. IEEE Computer Society (2005)
6. EEMBC Embedded Microprocessor Benchmark Consortium. Cpu benchmark–mcu benchmark–coremark
7. Davi, L., Liebchen, C., Sadeghi, A.R., Snow, K.Z., Monrose, F.: Code randomization resilient to (just-in-time) return-oriented programming. In: NDSS (2015)
8. Davi, L.V., Dmitrienko, A., Nünberger, S., Sadeghi, A.R.: Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and arm. In: 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, pp. 299–310 (2013)

9. Follner, A., Bartel, A., Bodden, E.: Analyzing the gadgets. In: International Symposium on Engineering Secure Software and Systems, pp. 155–172 (2016)
10. Hejazi, S.M., Talhi, C., Debbabi, M.: Extraction of forensically sensitive information from windows physical memory. Digit. Investig. **6**, S121–S131 (2009). The Proceedings of the Ninth Annual DFRWS Conference
11. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., Davidson, J.W.: Ilr: where'd my gadgets go? In: 2012 IEEE Symposium on Security and Privacy, pp. 571–585. IEEE (2012)
12. Kil, C., Jun, J., Bookholt, C., Xu, J., Ning, P.: Address space layout permutation (ASLP): towards fine-grained randomization of commodity software. In: 2006 22nd Annual Computer Security Applications Conference (ACSAC 2006), pp. 339–348. IEEE (2006)
13. Koo, H., Chen, Y., Lu, L., Kemerlis, V.P., Polychronakis, M.: Compiler-assisted code randomization. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 461–477. IEEE (2018)
14. Larsen, P., Homescu, A., Brunthaler, S., Franz, M.: SoK: automated software diversity. In: 2014 IEEE Symposium on Security and Privacy, pp. 276–291. IEEE (2014)
15. Microchip. Atmel start
16. Pallister, J., Hollis, S., Bennett, J.: BEEBS: open benchmarks for energy measurements on embedded platforms. arXiv preprint arXiv:1308.5174 (2013)
17. Priyadarshan, S., Nguyen, H., Sekar, R.: Practical fine-grained binary code randomization. In: Annual Computer Security Applications Conference, pp. 401–414 (2020)
18. Quinn, H.: Microcontroller benchmark codes for radiation testing
19. Shi, J., Guan, L., Li, W., Zhang, D., Chen, P., Zhang, N.: Harm: hardware-assisted continuous re-randomization for microcontrollers. In: 2022 IEEE European Symposium on Security and Privacy (EuroS P) (2022)
20. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., Sadeghi, A.R.: Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In: 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, 19–22 May 2013, pp. 574–588. IEEE Computer Society (2013)
21. Wang, X., Yeoh, S., Lyerly, R., Olivier, P., Kim, S.H., Ravindran, B.: A framework for software diversification with {ISA} heterogeneity. In: 23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020), pp. 427–442 (2020)
22. Wartell, R., Mohan, V., Hamlen, K.W., Lin, Z.: Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In: 2012 ACM Conference on Computer and Communications Security, pp. 157–168 (2012)
23. Feng, X., Wang, D., Lin, Z., Kuang, X., Zhao, G.: Enhancing randomization entropy of x86–64 code while preserving semantic consistency. In: 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), pp. 1–12. IEEE (2020)
24. Yiu, J.: Chapter 2–getting started with cortex-m programming. In: Yiu, J. (ed.) Definitive Guide to Arm®Cortex®-M23 and Cortex-M33 Processors, pp. 19–51. Newnes (2021)