# AVRAND: A Software-Based Defense Against Code Reuse Attacks for AVR Embedded Devices

Sergio Pastrana[1(✉)], Juan Tapiador[1], Guillermo Suarez-Tangil[2], and Pedro Peris-López[1]

[1] Department of Computer Science, University Carlos III de Madrid, Leganés, Spain
{spastran,jestevez,pperis}@inf.uc3m.es
[2] Information Security Group, Royal Holloway University of London, Egham, UK
guillermo.suarez-tangil@rhul.ac.uk

**Abstract.** Code reuse attacks are advanced exploitation techniques that constitute a serious threat for modern systems. They profit from a control flow hijacking vulnerability to maliciously execute one or more pieces of code from the targeted application. ASLR and Control Flow Integrity are two mechanisms commonly used to deter automated attacks based on code reuse. Unfortunately, none of these solutions are suitable for modified Harvard architectures such as AVR microcontrollers. In this work, we present a code reuse attack against embedded AVR devices that shows how an adversary can execute arbitrary code reused from the firmware and other external libraries. We then propose a software-based defense based on fine-grained random permutations of the code memory. Our solution is installed in the bootloader section of the embedded device and thus executes during every device reset. We also propose a self-obfuscation technique to hinder code-reuse attacks against the bootloader.

**Keywords:** Code reuse attacks · Return Oriented Programming · AVR · Internet-of-things · Embedded devices · Memory randomization

## 1 Introduction

The widespread adoption of communicating technologies such as smart or wearable devices enables users to interconnect their systems world-widely. The so-called Internet of Things (IoT) represents the integration of several computing and communications paradigms that facilitate the interaction between these devices. In this context, security and privacy play an important role as many of these devices incorporate sensors that could leak highly sensitive information (e.g., location, behavioral patterns, and audio and video of the device' surroundings). Moreover, embedded devices are frequently connected to the Internet, so they are valuable targets for malicious activities, such as botnets or spammers.

One common architecture for embedded devices is AVR[1], which is a modified Harvard architecture that physically separates the flash memory from the SRAM

[1] http://www.atmel.com/products/microcontrollers/avr/.


J. Caballero et al. (Eds.): DIMVA 2016, LNCS 9721, pp. 58–77, 2016.
DOI: 10.1007/978-3-319-40667-1_4

memory. While the former contains the executable binary, the latter stores the program data, heap, and stack. Flash memory can only be re-programmed from a special section called *bootloader*, and applications cannot be modified at runtime without flashing the entire memory. In addition, the number of times a memory can be flashed (namely cycles) is limited.

Memory corruption vulnerabilities have been widely explored as a strategy to hijack the execution control flow for a huge variety of systems, including embedded and mobile devices [6,12,15]. In the past, once the adversary gained control of the execution, the immediate next step was to directly jump into its own malicious payload, which was already injected in the exploit [8]. However, Data Execution Prevention (DEP) techniques turn code injection useless. AVR—together with other Harvard architectures—incorporate a type of hardware based DEP defense. This avoids the flash memory (where the executable code resides) being written from anywhere else except from the bootloader section, which also resides in the flash memory. Thus, the only means to exploit AVR devices is by reusing existing software from the flash memory [12,15].

***Related Work.*** Code reuse attacks were first implemented by reusing different functions imported from various libraries (such as `libc` [27]). Well-known countermeasures such as Address Space Layout Randomization (ASLR) [5] modify the memory layout of the function libraries during the loading process to effectively hinder these return-to-lib attacks. However, modern code reuse attacks can arbitrarily perform certain operations to carefully chain different pieces of code (called gadgets) based on the Return Oriented Programming (ROP) paradigm [17,20]. In fact, code reuse attack are still feasible in ASLR-based defenses using ROP due to memory leakage vulnerabilities [24]. For example, the JIT-ROP attack in [23] disassembles pages obtained from the leaked address to build a gadget chain at runtime. The exploitation of memory leakages assumes that the adversary can use large payloads, and that she can exploit the vulnerability several times. However, these assumptions are not generally valid for AVR devices, and the threat model is different from other less constrained architectures such as ARM or x86.

Countermeasures against code reuse attacks have been widely explored recently [4,6,7,9,10,12,15,18,22]. Current defenses can be classified as follows:

1. Memory randomization [4,6,9,25] obfuscates the layout of the program binary. To overcome memory leakages, this technique relies on certain Execute-only-Memory (XoM) areas, which can neither be read nor written. These areas can be used to store trampolines to real, randomized areas of code. Many of these solutions rely on hardware-specific properties, such as Intel Extended Tables [9,25], which obviously are not applicable to AVR. A recent work by Braden et al. [6] performs a software-based XoM for ARM embedded devices. However, the authors also rely on a specific hardware component, namely the link register used in ARM, to prevent address disclosure.
2. Control Flow Integrity (CFI), which typically determines which are the valid targets for each control flow statement (e.g., jumps or returns), and prevents non-valid flows. CFI usually incurs an expensive overhead [18], which is not suitable for resource-constrained systems such as AVR.

Most of the attacks and defenses so far target either x86 or ARM architectures. In these cases, the adversarial model and the defense capabilities are radically different from those applicable to AVR. Current approaches aiming at hindering code reuse attacks in Harvard-based architectures rely on adding additional hardware [15] or modifying the existing one [13]. Such countermeasures introduce additional costs to these devices that cannot be overlooked. This is especially critical in scenarios where devices are expected to be inexpensive, as it usually happens with many IoT deployments. Furthermore, there are settings where the hardware is already given "as it is", such us in industrial environments [21], vehicular systems, and home automation projects [26], to name a few.

***Contribution.*** In this work, we demonstrate code reuse attacks against AVR devices and provide a software-based defense named AVRAND. The novelty of our work lies in providing an inexpensive solution targeting endpoint users and distributions rather than manufacturers, vendors, or hardware architects. We argue that the capabilities of an attacker are much more limited when dealing with hardware constrained devices such as an Arduino. Based on this, we balance the trade-off between its capabilities and the level of protection implemented to provide a practical and robust countermeasure. To the best of our knowledge, this is the first work looking at this problem from this viewpoint that proposes a software-based defense for AVR-based devices. Our randomization engine is encoded in the bootloader section of the device and, thus, it is executed after every reboot. Moreover, since the bootloader itself is a potential target for code reuse attacks, AVRAND applies an obfuscation technique using an XOR-based self encryption function. To facilitate reproducibility of our results and foster research in AVR security, we provide functional prototypes of the attack and the proposed defense for an Arduino Yun device (Sect. 5), which is an emerging platform widely used in the IoT arena.

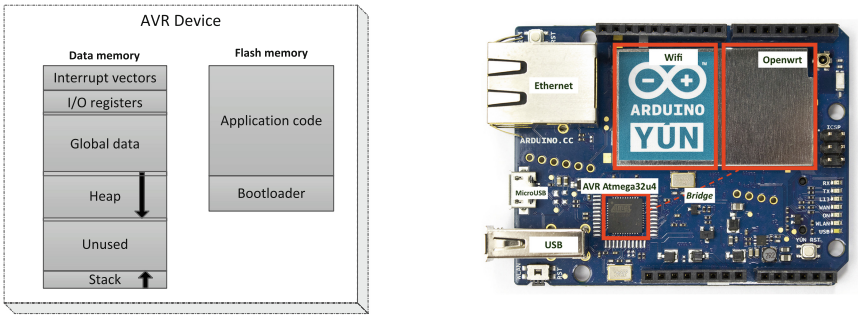
## 2 Background

In this section, we provide a brief background on the target systems studied in this work: the AVR architecture and the Arduino Yun, which is the platform used during our experimentation.

### 2.1 The AVR Architecture

AVR is a modified Harvard architecture implemented by Atmel in 1996. AVR is widely deployed in embedded devices due to its simplicity and low cost, and it is present in a variety of applications, including automotive systems [3], the toy industry, and home automation systems [26].

AVR devices store code and data in memories that are physically separated, i.e., the flash memory and the data or SRAM memory (see Fig. 1a). To allow self-programming, two special instructions are provided to load data from flash to SRAM memory (Load Program Memory, LPM), and to store data in the flash memory (Store Program Memory, SPM). The latter can only be invoked from




(a) Schematic view of AVR memories. (b) Arduino Yun board.

**Fig. 1.** AVR and Arduino Yun boards.

a special memory region called the *bootloader*, and thus all the reprogramming code must reside in this region. The flash memory in AVR is structured in pages, which are addressed different than the SRAM. Actually, the program counter (PC) does not hold the actual address, but a page-based index. Specifically, the most significant bits of the PC are mapped to the page number, while the less significant bits are mapped to the offset within the page. As shown throughout this paper, AVRAND uses this property to manage the memory randomization efficiently. AVR has 3 special registers, called X, Y and Z, that are used for direct and indirect addressing and have added properties such as automatic increment (e.g., Y++) or fixed displacement (e.g., Y+k). These special registers are mapped with 8-bit general purpose registers (e.g., Y is the concatenation of r28 and r29).

The SRAM contains the program data, the heap and the stack, which are unique as AVR runs a single process at a time. A property of AVR is that the stack starts at the highest address and grows towards lower addresses (i.e., a *PUSH* instruction stores a new byte in the stack and decreases the stack pointer), while the heap grows towards higher addresses and can eventually collide with the stack. Additionally, the data memory also contains I/O registers such as the status register or the stack pointer. This implies that the stack pointer is directly mapped in program memory and can be read and write by load and store instructions, respectively.

Code running in embedded AVR devices may contain a huge amount of firmware and library functions required to integrate and operate different sensors, such as thermometers, motion sensors, cameras, etc. Since AVR does not provide dynamic loading of libraries, integrated libraries are statically linked at compilation time. AVR binaries follow the Hexadecimal Object File (HEX) format [16]. These binaries must be uploaded (flashed) to program memory using either an In-System Programming interface (ISP) or by communicating with the bootloader using a universal asynchronous receiver/transmitter (UART) [2].

### 2.2 Arduino Yun

Arduino[2] is an open-source platform originally proposed to be used in electronics and microcontroller projects. With the increasing interest in the IoT, the Arduino Yun has been designed specifically to run IoT applications, by combining both the low-level electronics originally present in other Arduino devices with higher level architectures running a Linux based operating system. Specifically, the Arduino Yun contains a board based on two chips (see Fig. 1b). One is the Atmel ATmega32u4 (AVR MCU) and the other is an Atheros AR9331. The Atheros processor holds a Linux distribution based on `OpenWrt` and has built-in Ethernet and WiFi support.

The AVR chip and the `OpenWrt` are connected through a *Bridge*, i.e., a logical component programmed in the `OpenWrt` which communicates with the AVR chip using a serial port. An Arduino Bridge library provides the required functionality to communicate applications running in the AVR chip with the `OpenWrt`, including a *Process* object that allows to run shell commands in the `OpenWrt` *shell* or a *HttpClientd* that allows to connect the AVR to internet. As shown in Sect. 3.3, the proposed exploit uses functions from the Bridge library to compromise the `OpenWrt` shell.

## 3 Code Reuse Attacks in AVR

In this section, we demonstrate code reuse attacks in AVR binaries using ROP and other similar exploiting techniques [27]. We first present the adversarial model assumed and then provide a general description of the attack. Finally, we describe the implementation of a prototype for Arduino Yun devices.

### 3.1 Assumptions and Adversarial Model

In this work, we consider the following assumptions and adversarial settings:

- The targeted embedded device is based on the AVR architecture and it is not tamper-proof. Thus, if physically accessible, the adversary can dump all the contents from the data and code memories at any time.
- The adversary cannot inject arbitrarily large payloads. We elaborate more on this limitation in Sect. 3.2. However, an adversary could inject relatively large payloads in memory by using software resets and multiple runs.
- The adversary could gain the control of the program flow by remotely exploiting a memory corruption vulnerability on the device, for example a stack or heap overflow.
- The program includes library functions that are useful for the adversary. For example, we assume that the program includes the *Bridge* lib that allows communication between the AVR and `OpenWrt` chips in Arduino Yun.

[2] https://www.arduino.cc.

### 3.2 Attack Overview

In this section we present a code reuse attack for AVR devices. Due to the limited capacity of the AVR memory, the adversary is not able to use large exploiting payloads, and thus she has to inject additional data into the SRAM. This is also used when a function library function is *called by reference*, i.e., when the arguments are passed as pointers to data memory. Contrarily to other architectures, function arguments in AVR are passed via registers whenever possible, and through the stack only when the arguments are larger than the length of the registers. An adversary may also be able to change any data from the SRAM memory. For example, Habibi et al. [15] proposed an attack that modifies the registers of an Unmaned Aerial Vehicle (UAV) gyroscope to control its flight.

**Injecting Data into the SRAM.** Injecting data into the SRAM is limited by the amount of memory available for the exploit. The main idea is to use a set of gadgets that, when chained together, could potentially store data into non-volatile areas of the SRAM memory [12,15]. We call this chain of gadgets *Store_data*. Ideally, the fewer the number of gadgets used the better, as each gadget may require to include its pointer in the exploit. During our experimentation, we have found a pair of gadgets that allow an adversary to build a payload that loads several values in memory recursively. We provide more details of these gadgets and how they are used in our prototype in Sect. 3.3.

Since the stack is located at the highest address of the SRAM memory, the space available to inject a payload after overflowing the stack is significantly limited. When a buffer is locally declared in a function, the return address is stored at a higher position of the memory allocated in the stack. This position may be close to the end of the SRAM address space (see Fig. 1a). Thus, the adversary is not able to send large attack payloads as it is usually done in ROP attacks against conventional architectures [23]. To partially overcome this issue and provide more space, the stack pointer can be moved to the beginning of the buffer as proposed in [15]. In this way, the buffer itself can be fully used to allocate the payload, and the size of the payload injected by the adversary intrinsically depends on the available buffer size. We call the gadgets that allow to move the stack *Stack_move*.

Given that the amount of injected data is limited, exploiting the same vulnerability multiple times could place the attacker in an advantageous position. However, exploiting a buffer overflow usually leaves the memory in a non-deterministic state and the attacker is usually forced to reset the device each time to maintain the device functional and/or resume its normal operation. To this end, existing works proposed to repair the stack right after the attack succeeds [14,15]. While this is useful to modify a few memory data bytes (such as the UAV gyroscope), repairing the stack does not provide the adversary with extra data space since the payload is always limited by the memory size—in fact, using the gadgets that repair the stack requires additional space in the payload. In this regard, Francillon and Castellucia [12] proposed to perform a software reset by directly jumping to the address $0x0000$ (i.e., the reset vector).

However, this approach is not suitable for modern AVR chips since it does not guarantee that the I/O registers are restored to their initial state[3]. In this work, we propose the use of a gadget, namely *Reset_chip*, that uses a *watchdog reset*, which is one of the reset sources used in AVR. More precisely, the gadget first establishes a watchdog timer and then jumps to an infinite loop. When the timer expires, the watchdog causes a software reset.

Figure 2 shows a schematic view of a generic data injection attack. When the vulnerable function is called, the return address is pushed on the stack. The attack starts by overwriting this address with the address of the *Stack_move* gadget (Step 1), which pops the new address and stores it in the memory address corresponding to the stack pointer (SP). From there on, the buffer constitutes the new stack (Step 2). Then, the address of the next gadget is popped from the stack, so the first bytes of the buffer must point to the *Store_data* gadget (Step 3) that stores the data at a given address (Step 4). As showed in Sect. 3.3, both the stored data and the SRAM memory addresses must be included in the payload. Finally, when the *Store_data* gadget returns (Step 5), the program jumps to the *Reset_chip* gadget (Step 6), which performs a clean software reset of the AVR chip. The adversary, while needed, may send a new payload to exploit the vulnerability and store additional data in consecutive addresses. In every reboot, the *.data* and *.bss* sections (i.e., data and heap) of the SRAM memory are cleared and reloaded, so if the adversary stores data in a memory area different from these (e.g., the region tagged as *unused* in Fig. 1a), then such data will persist across reboots.

**Calling Library Functions.** Once the required data are stored in memory, the adversary is ready to use library functions. The idea is to perform a similar approach to classical "return-to-lib" attacks [27]. Arguments are passed through registers, which can be easily loaded by using gadgets that pop values from the stack and stores them in registers. During our experimentation we have observed that these gadgets are frequent in many AVR binaries.

The adversary is now ready to call the library function using a chain of gadgets that performs the desired operation. First, she must load the arguments and prepare the required data (e.g., pointers to objects) using the data injection scheme explained above. Next, the program flow must jump to the desired function itself.

### 3.3 Attack Implementation in Arduino Yun

In this section, we describe and implement an attack that targets Arduino Yun devices, allowing an adversary to execute remote commands in the `OpenWrt` environment of these devices (i.e., bypassing the *Bridge* between the two chipsets). The attack comprises two phases: *injection* and *invocation*. First, it starts by injecting the command into SRAM memory as a *String* object, and then

[3] http://www.atmel.com/webdoc/AVRLibcReferenceManual/FAQ_1faq_softreset.html.

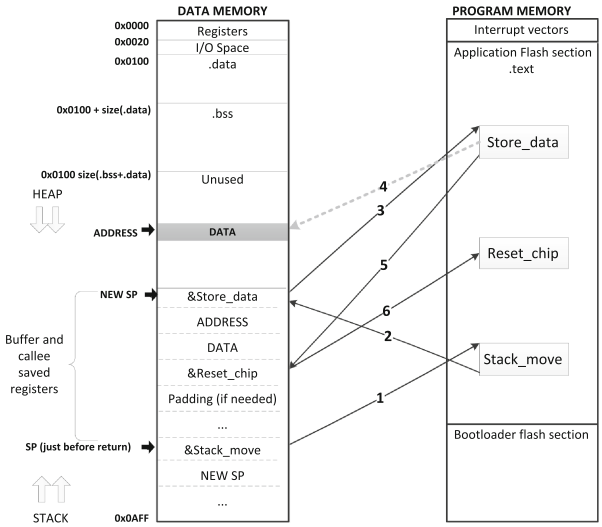



**Fig. 2.** Scheme of the data injection ROP attack.

forces the execution of the function *runShellCommand(String* cmd)* from the Bridge Library[4] by passing as argument the pointer to the injected object.

We assume that the adversary is able to exploit a memory corruption vulnerability and hijack the control flow. In this work, we have exploited a function (implemented ad-hoc for the prototype) that receives data from the serial port and stores it into a buffer, without checking its bounds. By sending crafted data, we are able to overwrite the return address of the function and take control of the program flow. We next explain the implementation details of the attack.

**Command Injection into the SRAM.** Table 1a shows a pair of gadgets that chained together move the stack pointer (SP) to a given address. The first gadget loads the new SP to registers r28 and r29, while the second gadget stores the SP in 0x3e and 0x3f, which are actually the positions mapping the SP. This is possible because AVR uses fixed positions of data memory to store I/O registers, including the SP. Gadgets used to move the stack are very frequent in AVR binaries, since they are used to save and restore the stack within the called functions.

To store the data in SRAM, we have found an optimal pair of gadgets (see Table 1b) that are included with the *String* library (imported by default in all Arduino programs). As these gadgets are consecutive in the code, they can be used recursively. In the first interaction, the gadget *Load_data* at address 0x2c00 loads data in registers r16 and r17, and the destination address in registers r28 an r29. As explained in Sect. 2.1, registers r28 and r29 are mapped to the register Y used for direct addressing. Here, the gadget *Store_data* showed in Table 1b uses

[4] https://www.arduino.cc/en/Reference/YunProcessConstructor.

**Table 1.** Gadgets used to move the stack to a desired position (a) and to inject data in SRAM (b).

| Address | Instructions | Description |
|---|---|---|
| (a) Stack_mov_1 | | |
| 0x0c84 | pop r29<br>pop r28<br>ret | Loads the new stack pointer in registers r28 and r29 |
| (a) Stack_mov_2 | | |
| 0x39e4 | in r0, 0x3f<br>cli<br>out 0x3e, r29<br>out 0x3f, r0<br>out 0x3d, r28<br>movw r28, r26<br>ret | Stores the new address in the SRAM memory addresses mapping the stack pointer (i.e. 0x3e and 0x3f) |

| Address | Instructions | Description |
|---|---|---|
| (b) Store_data | | |
| 0x2bf6 | std Y+3, r17<br>std Y+2, r16<br>ldi r24, 0x01<br>rjmp .+2 | Stores the values from r17 and r18 in addresses Y+3 and Y+4 (mapped to r29 and r28) and jumps to 0x2c00. |
| (b) Load_data | | |
| 0x2c00 | pop r29<br>pop r28<br>pop r17<br>pop r16<br>ret | Loads the new values at r17 and r16 and new addresses at r28 and r29 |

the fixed displacement of the Y register to store the values from r16 and r17 in addresses Y+2 and Y+3 respectively. Because the end of the gadget *Store_data* directly jumps to the gadget *Load_data*, they can be used repetitively, as shown in Fig. 3.

To perform a software reset of the AVR chip, we use one of the reset sources provided by the AVR architecture, the *watchdog reset*, which establishes a time-out and resets the chip when it expires. Table 2a shows the gadgets used. A first gadget enables the watchdog and sets a timeout to 120 ms. This gadget is present in all Arduino programs since it belongs to one of its core libraries, CDC (the USB Connected Device Classes). The second gadget performs an infinite loop and is intended to wait until the timer expires. This gadget, which consists of just one instruction, is the last instruction of every Arduino program and represents the "stop-program" instruction that maintains the device in an *idle* state. By chaining these two gadgets, the chip automatically resets and the normal operation of the Arduino device is restored. Then, the adversary may send a new exploit to store more data, depending on what she wants to inject.

**Command Invocation.** In the previous section we have described how an adversary can store any data in the SRAM. Now, we show how she could use such data to execute commands in the `OpenWrt` of an Arduino Yun. Using the data injection process, the adversary writes in memory the raw sequence of characters of the command (e.g., "curl", as shown in Fig. 2). Then, a *String* object pointing to such sequence must be created. A *String* object has three components. First, a pointer to the sequence of characters (2 bytes); second, the length of the sequence (2 bytes); and, finally, its capacity (2 bytes).

To execute the inserted command, we call the function *runShellCommand* of the Bridge Library. This function takes as argument the address of the *String* object that represents the command, which is provided in registers. *Load_arguments* gadget, showed in Table 2b performs such loading. In many AVR binaries it is frequent to find pop instructions before a return, and thus

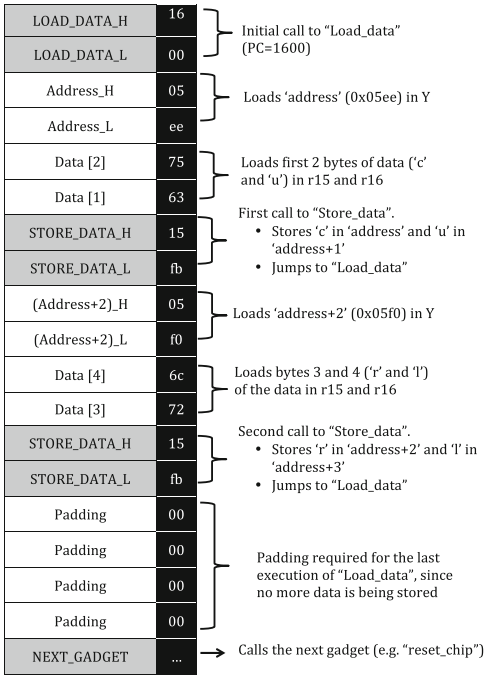



**Fig. 3.** Schematic view of a payload that inserts the command "curl" (0x63,0x75,0x72,0x6c) into a the address 0xef00 of SRAM memory using the gadgets from Table 1b.

it can be assumed that this gadget can be easily obtained. Finally, after the *Load_arguments* gadget is executed, the program should directly jump to the *runShellCommand* function which uses the Bridge between the two chips to execute the desired command in the `OpenWrt`.

## 4 Design and Overview of AVRAND

In order to defeat code reuse attacks, we propose AVRAND, a solution that randomizes the layout of the flash memory where the binary code resides and obfuscates the randomization engine. Since the core of AVRAND resides in the bootloader of the flash memory, it re-randomizes the complete program memory after every software reset, thus preventing attacks that exploit the vulnerability several times (e.g., brute force attacks) and requiring adversaries to use one-shot clean attacks (i.e., attacks that do not rely on software resets). Moreover, as we discuss in Sect. 6, AVRAND could be configured to defeat other exploitation techniques that do not require to reset the device.

AVRAND is composed by two main modules: *preprocessing* and *runtime*, as depicted in Fig. 4. First, the *preprocessing module* modifies the HEX file that is being uploaded into the AVR device so that it can be randomized. This module

**Table 2.** Gadgets used to reset the microcontroller (a) and to load the arguments to the function *runShellCommand* (b).

| Address | Instructions | Description |
|---|---|---|
| a) Reset_chip_1 | | |
| 0x1c56 | ldi r18, 0x0B<br>ldi r24, 0x18<br>ldi r25, 0x00<br>in r0, 0x3f<br>cli<br>wdr<br>sts 0x0060, r24<br>out 0x3f, r0<br>sts 0x0060, r18<br>ret | Sets the timeout<br>to 120 ms, disables<br>interrupts and enables<br>the watchdog |

| Address | Instructions | Description |
|---|---|---|
| a) Reset_chip_2 | | |
| 0x3a0a | rjmp .-2 | Relative jump to itself<br>(i.e. infinitive loop) |
| b) Load_arguments | | |
| 0x2b52 | pop r25<br>pop r24<br>pop r23<br>pop r22<br>...<br>ret | Loads the arguments<br>into registers. Note that<br>some useless instructions<br>are omitted. Upon return<br>the program should jump<br>to *runShellCommand* |

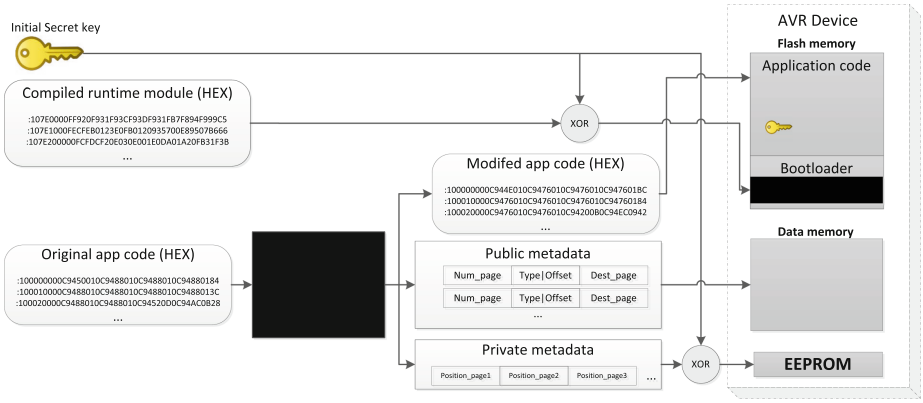



**Fig. 4.** AVRAND overview.

is executed once in an external computer, before uploading the binary to the device. Second, the *runtime module* is installed in the bootloader section of the device to perform the actual randomization of the flash memory after each device reset. Moreover, this module uses an obfuscation technique to prevent code reuse attacks on the bootloader, by applying XOR-based encryption.

**Preprocessing Module.** This module is executed once and prepares the code so that it can be randomized. First, it reads the original HEX file and gets a list of all the control-flow statements, including both absolute and relative pointers within the code (e.g., jumps and calls, conditional branches, etc.) and also indirect pointers that may be in the data section (e.g., C++ vtables). Using relative offsets is common in AVR binaries due to code-size optimization, but this is not compatible with a randomization approach since relative positions change from one layout to another. Thus, during the preprocessing module all the relative operations are replaced by their absolute versions (e.g., RJMP are substituted by JMP and RCALL by CALL instructions).

Since the flash memory in AVR is structured in pages, AVRAND performs randomization at a paged-grained level. However, in order to preserve the semantics of the entire code, pages are linked using JMP instructions. Thus, all control-flow statements in the code point to absolute positions and can be re-calculated at runtime during each randomization. Accordingly, the preprocessing module outputs a list of *public metadata* (i.e., we assume that an adversary may know this information) used to update the offsets during the randomization (see Sect. 5.1 for details). Furthermore, a list of initial page positions is also created to indicate the offsets of each page in the binary, which must be kept secret from adversaries and thus it is named *private metadata*.

The modified binary code is then flashed onto the flash memory and the *public metadata* in the SRAM, while the *private metadata* is encrypted with the XOR key and flashed in a non-readable memory area of the embedded device. For example, many AVR devices are equipped with an external EEPROM memory that is not directly addressable without special functions in the program binary, so it can be used to store the *private metadata*. Finally, the initial private key is stored in a fixed position of the flash memory. Note that during each randomization a new key is generated which overwrites the previous one.

**Runtime Module.** This module is installed in the bootloader section of the device and it performs the actual randomization of the memory layout each time the device is reset. First, it reads the current page positions (i.e., the offset of each page) from the *private metadata* to get the actual memory layout of the device, and decrypts it using the secret key. Second, it generates a map of random swaps indicating couples of pages randomly paired that must be exchanged. This map is used to update the current page positions in the *private metadata*. Furthermore, the offsets of every control-flow statement in the program memory are re-calculated and updated by using the new positions and looking at the *public metadata*. Finally, the entire memory is re-flashed, swapping all the pages that purely contain code. To do this, both pages are temporary stored in the SRAM and then they are re-written into each others' offsets of the flash memory. Note that a complete random permutation of the memory layout would require to store an entire copy of the binary in SRAM, which demands much more memory than keeping only two pages at a time in memory.

The entire flash memory is structured in pages, but certain pages cannot be shuffled during the randomization. These are pages that contain data (which are either before or after the code, never interleaved) and the first two pages which contains the interrupt vectors. Pages containing data remain in constant memory offsets. However, two pages may contain both data and code (i.e., one page before the program and one page after the program), and code in these pages may be used in a code reuse attack. In the worst case, each of these two pages will have a single byte of data and code in the bottom part of the section (i.e., $page\_size - 1$). Thus, the maximum size of code that remains constant during randomization is $2 * (page\_size - 1)$ (i.e., 254 bytes in the Atmega32u4 chip).

Each page contains 128 bytes of code, i.e., approximately 42 instructions. Thus, gaining knowledge of a single page does not position the attacker in a privileged situation since she may not find enough gadgets to perform a code-reuse attack. Moreover, the probability of guessing a page in AVRAND is $1/N_p$, where $N_p$ is the number of swapped pages (which depends on the size of the program memory, as discussed below). This probability outperforms state-of-the-art solutions like Isomeron [10], which has a probability of 0.5 of being discovered at each gadget.

As stated before, the runtime module is compiled and uploaded into the bootloader section of the embedded system. Accordingly, this is the first piece of code being executed after every device reboot, which prevents code reuse attacks using software resets and reducing the chances for brute force attacks aiming to discover the memory layout. However, the bootloader itself could be the target of code reuse attacks (in our experiments, the bootloader contains around 4 KB of code) and thus it should be protected as well. AVRAND solves it by applying a simple obfuscation technique using an XOR based encryption. As such, most of the bootloader is stored encrypted. The runtime module uses a non-encrypted routine that is executed at the beginning to decrypt the bootloader and then jumps to its main function. Once the randomization is finished, and before jumping to the application section, a new random key is generated and used to re-encrypt the bootloader and the private data from the EEPROM.

## 5 Implementation

We have developed a freely available[5] prototype of AVRAND for the Atmel Atmega32u4 chip included in the Arduino Yun platform. In this section, we discuss its implementation details.

### 5.1 Preprocessing Module

We have implemented the preprocessing module in Python. It takes as input the HEX file of the original application and generates a modified HEX in such a way that it can be randomized at runtime by the bootloader. The initial list of control-flow statements is obtained from the assembly code, which is generated from the HEX file using the open source tools *avr-objcopy* and *avr-objdump* [19]. Control-flow statements may be one of the following: relative or absolute jumps (RJMP/JMP), relative or absolute calls (RCALL/CALL), conditional branches (BR), pointers to function prologues and epilogues (used by some functions to save registers in the stack), pointers to global variable constructors (CTORS) and C++ specific virtual pointers (vpointers). Also, a list of indivisible instruction sequences is obtained, in order to avoid placing jumps between them during the page linking. Examples of such non-breakable instructions are all the two-word instructions, or the CPSE instruction that compare two registers and jumps to PC+2 or PC+3 depending on the result.

[5] http://www.seg.inf.uc3m.es/~spastran/avrand/.

```
270: d9 f7      brne .-10; 0x268
272: 24 e0      ldi r18, 0x04 ; 4
```

```
270: 09 f4          brne .+2 ; 0x274
272: 02 c0          rjmp .+4 ; 0x278
274: 0c 94 34 01    jmp 0x268 ;  0x268
278: 24 e0          ldi r18, 0x04 ; 4
```

**Fig. 5.** Transformation of a relative conditional branch (left) to its absolute version (right).

Then, each instruction using relative offsets (i.e. RJMP and RCALL) is substituted by its corresponding absolute version (i.e., JMP and CALL). Changing relative by absolute versions adds 2 extra bytes. In case of conditional branches, we follow an approach similar to Oxymoron [4] to transform them into an absolute version, by adding a RJMP and a JMP instruction. This transformation is shown in Fig. 5. The whole BR/RJMP/JMP block is considered as an indivisible sequence in order to maintain its semantics. As it can be observed, each conditional branch modified adds 6 extra bytes to the binary code. Every time that the module inserts new code bytes, the offsets of the entire program are updated accordingly.

The next step is to link the pages using absolute JMP instructions, which are inserted in the bottom of each page, i.e., the last instruction of every page is a JMP to the first instruction of the next page. In this way, whenever a page changes its position during randomization, these linking pointers can be updated to point to the new address where the next page begins. The insertion of a JMP may occur between an indivisible sequence of instructions. If such situation is detected, the entire sequence is moved forward, to the beginning of the next page, by adding padding (i.e., NOP instructions).

Finally, the new HEX file is generated along with the *public metadata* and the *private metadata*. The *public metadata* provides the list of structures representing each control-flow statement. Concretely, each structure indicates the page where the statement is, the offset within the page, the type (i.e., CALL or JMP, prologue/epilogue function pointer, C++ vpointer or pointer to a global variable initialization routine), and the page pointed. Note that the offset within the page does not change in the randomization, and thus it is not necessary to store it since it can be obtained from the PC address, as explained in Sect. 2.1.

The binary code (HEX) and the *public metadata* are uploaded to the flash and data memories respectively, while the private metadata is encrypted (using an XOR-based encryption and a private key of 128 bytes) and uploaded to a memory region that is not directly observable by an adversary. During our experiments, we used the external EEPROM present in the Atmega32u4 chip of the Arduino Yun. In order to upload these contents to the device, we use the open source tool *avrdude* [11].

### 5.2 Runtime Module

The main purpose of the runtime module is to perform the randomization of the entire application after every device reset. Thus, it must be stored in the

booloader section of the flash memory. However, the bootloader contains critical functions from the standard library, such as those for reading and writing the *private metadata*. In a scenario where the adversary can reuse any code from the flash section, this *private data* would be accessible by just jumping to the proper function in the bootloader.

To protect the bootloader, we introduce a self-encryption and self-decryption routines that obfuscate its contents. Thus, these are the only two routines that could be potentially used in code reuse attacks. In our prototype they both occupy less than 2 pages (i.e., 256 bytes), which prevents the use of a practical ROP attack against our system. Moreover, these non-encrypted pages can also be shuffled by the randomization engine to prevent attackers from pinpointing them. Indeed, as the adversary is forced to perform the attack in one-shot, then if she is able to decrypt the bootloader, when trying to use it or read the *private metadata*, the device may be reset, which modify the *private metadata*.

The runtime module can be divided into 3 main parts: an initialization routine, the bootloader itself, and the encryption/decryption routine. The first one holds the Interrupt vectors and some required initialization instructions, and jumps to the decryption routine. The second part, which is encrypted, contains the main functionality to setup the hardware and randomize the binary code. Finally, the last part encrypts again the bootloader and the private data, and jumps to the beginning of the application code.

The decryption process reads the key (stored at a fixed position of the flash memory). This key has the same length than the page size (i.e., 128 bytes). Then, it reads the encrypted bootloader page by page, performing the XOR to obtain the clear-text of the code, and rewrites the output in the same position. Then, it jumps to the beginning of the decrypted bootloader.

The bootloader starts by setting up the required hardware (e.g., to initialize the USB or the clock of the device). It then performs the actual randomization of the application binary. To do so, it first reads the *private metadata* and loads it into a temporary buffer of the SRAM memory (which is deleted once finished). Second, it creates a random list of pairs of pages (i.e., the random swap), that must be exchanged, and updates the *private data* by exchanging the page positions. We use the *rand* function implementation from `libc`, which uses a LFSR based random number generator. However, in order to get the random seed, we rely on a timing jitter produced after the variance introduced between the internal timer of the AVR chip and the oscillator used by the watchdog timer [1]. In this way, AVRAND produces truly random numbers in each execution of the randomization engine.

Once the random swap map is obtained, the bootloader processes one by one the pages from the bottom of the application section. Each page is temporarily stored in data memory, its control-flow statements are modified, and then it is stored again in the position indicated by the *private metadata*. Control-flow statements are updated by looking at the *public metadata* (i.e., where the pointer is, its type, and the page being pointed at) and the *private metadata* (i.e., the new position of the pointed page). In order to swap two pages, they are both

stored in SRAM memory and then re-written in each other's previous position of the flash memory. Thus, the size of SRAM required during the randomization is $page\_size * 2$. Finally, the new page positions (i.e. *private metadata*) is encrypted again, and written to the EEPROM memory. In order to prevent brute force attacks against the cryptosystem, the randomization engine generates a new XOR key each time. Figure 6 shows a schematic view of the memory layout of the application section before and after randomization.

Finally, when the randomization process is finished, the last step is to obfuscate the bootloader again using the XOR-based encryption routine and the newly generated key.

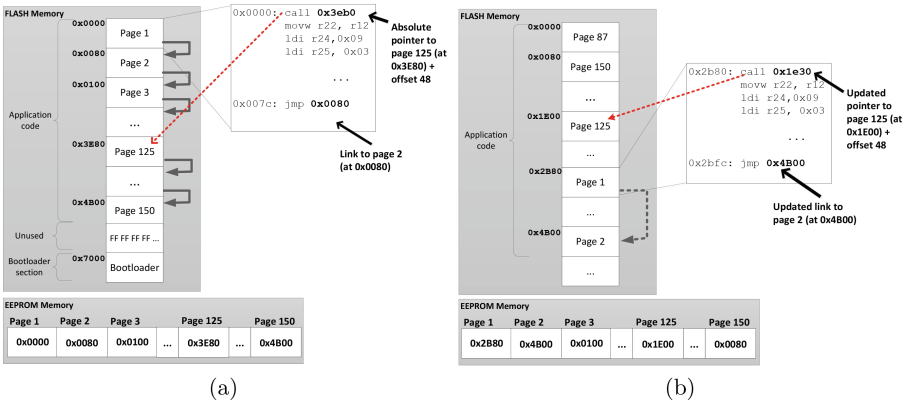



**Fig. 6.** Flash memory layout before (a) and after (b) randomization.

## 6 Discussion

AVRAND hampers code reuse attack by randomizing the application layout from the bootloader and by obfuscating the bootloader itself. We next provide a discussion of the suitability of our approach and the introduced overhead.

### 6.1 Suitability of AVRAND

AVRAND is designed specifically for AVR architectures. However, it could also be applied to other systems using a modified Harvard-based architecture, given that it is provided with a bootloader section that reprograms the flash memory. While the core idea of AVRAND (i.e., randomization of the memory layout) has been widely studied for other architectures such as x86 [5] or ARM [6], few works have addressed the problem in AVR. Moreover, our focus is on using a lightweight cryptographic routine, since AVR is designed for resource-constrained embedded

devices. In our prototype we have used an XOR-based encryption and a linear PRNG, which fit well in the space given for the bootloader section (4 KB). Nevertheless, our architecture is designed to accept stronger cryptographic functions if enough resources are available (e.g., using AES or 3DES and the more robust MersenneTwister PRNG). Nonetheless, in addition to a greater performance overhead, the use of complex encryption would have an extra drawback in AVRAND: since the code used to encrypt and decrypt the bootloader can be used in code-reuse attacks, using encryption and decryption routines with larger code size increases the available code for attackers. As explained in Sect. 5.2, currently the XOR-based encryption only occupies 2 pages.

External hardware can also be applied to palliate code reuse attacks [13,15]. We emphasize that our approach is complementary, but it benefits from a pure software-based solution. This perfectly suits scenarios where cost-minimization strategies play an important role in the device design. Francillon and Castellucia mentioned different protection mechanisms to prevent code injection attacks [12], such as preventing software vulnerabilities or using stack canaries. These mechanisms aim at avoiding the control-flow hijack and are complementary to the randomization provided by AVRAND. Our solution assumes that somehow the control flow may be hijacked, and thus it intends to hinder code reuse. Additionally, when the sensor is not physically accessible, then the chances for and adversary also decrease. While this may be subject for future research, we consider that AVRAND takes a step forward in the security of AVR devices.

### 6.2 Limitations

During the design of AVRAND, we have assumed that the exploit size is restricted by the size of the SRAM memory. For example, as explained in Sect. 3.2, the stack size may not be large enough to store a complex payload, thus limiting stack-based exploitation and requiring the adversary to reset the device when injecting large payloads in memory. Additionally, some devices based on the TinyOS restrict the packet size to 28 bytes. However, this is not the case of other chips like the Atmega32u4. Accordingly, other exploitation techniques such as heap or integer overflows may provide the adversary with the ability to inject larger payloads.

In this work, we have considered that the memory should be re-randomized with every device reset. Indeed, it is reasonable that a reset may be produced because the device is under attack or some other abnormal activity. However, we are aware that a smart adversary may find techniques to attack the sensor without causing resets or a system crash (e.g., by cleaning the stack after the payload execution [14,15]). In any case, AVRAND could be configured to reset the device periodically, or only under certain conditions. Due to the limited number of write/erase cycles of the flash memory (e.g., 10,000 in the Atmega32u4 chip), this feature should be carefully adjusted to meet the security requirements while maximizing the lifetime of the chip, which in turn depends on the application scenario. For example, by periodically randomizing the device every 5 min, a device using the Atmega32u4 chip would last approximately 35 days.

Finally, it is important to understand that AVRAND is a countermeasure to code reuse attacks in AVR based chips. However, these chips may be directly connected to other sensors (e.g., wireless antennas or thermometers) or chips (e.g., the Atheros chip in the Arduino Yun). In this last case, the Atheros chip in the Arduino Yun has far more resources than the AVR to secure the device. Indeed, the installed `OpenWrt` OS has support for ASLR, DEP, and other security measures such as authenticating and encrypting communications (e.g., through SSH). If the adversary could gain access to the MIPS-based chip (for example, by performing a brute force attack against the SSH or exploiting a vulnerability in the Linux kernel), then the security gained by AVRAND would be useless. However, no matter how strong the security measures taken in the Atheros chip are, the exploitation of AVR opens a security hole, since both chips are connected through the Bridge library. This is where AVRAND is particularly helpful.

### 6.3 Overhead Incurred by AVRAND

We have tested the prototype of AVRAND in the Atmel atmega32u4 chip integrated within the Arduino Yun device, equipped with a 32 KB flash Memory (from which 4 KB corresponds with the bootloader section). Our evaluation indicates a noteworthy increase in the code size due to changes introduced by the preprocessing module. We have tested our prototype on the entire set of examples included in the Arduino IDE software. While all the tested programs fit in the flash memory, we have observed an average of 20 % of extra code on the modified binary. However, this overhead is related to a binary which has been compiled turning on the optimization flags of avr-gcc [19], that prioritizes the use of relative versions of control flow instructions. However, if these optimization flags were turned off, as done in MAVR [15], then the difference between initial code size and modified code size would be considerably smaller. Re-compiling all the libraries without an optimization requires having the source code of every library (which is certainly not possible in case of proprietary code), so we decided to transform the binary directly in our preprocessing module, thus providing a more general solution.

As for the time spent by the runtime module in the bootloader, results show that it requires an average of 1.7 s to randomize the code of our proof-of-concept program, which takes 18 KB of the flash memory. For example, a bootloader using the AVR 109 protocol [2] (that allows self-programming without external programmers) takes a minimum of 750 ms. Given the security provided by AVRAND, we consider that an overhead of 1 s is acceptable, especially since the bootloader is only executed under certain circumstances.

## 7 Conclusions

In this paper, we have presented a software-based defense against code reuse attacks for AVR systems—a modified Harvard architecture. These type of architectures are popular among embedded devices used in different contexts.

We focus on providing an inexpensive solution tailored for resourced constrained devices. Our system perfectly balances the trade-off between the attack surface exposed in this class of devices and the level of protection required to defeat code reuse attacks. Thus, we design an architecture based on a fine-grained randomization defense with self encryption that does not require additional hardware support. We have implemented a proof-of-concept for the Arduino Yun, an emerging open-source platform widely used in the IoT arena. Our prototype introduces a negligible overhead with respect to the normal operation of the Arduino. We evaluated the proposed scheme against a code reuse attack based on Return Oriented Programming that first exploits a buffer overflow to execute code from the Arduino libraries. Finally, to foster research in this area, we provide functional prototypes of the attack and the proposed defense.

**Acknowledgments.** We would like to thank our shepherd, Andrea Lanzi, for his assistance and the feedback provided during the reviewing process. This work was supported by the MINECO Grant TIN2013-46469-R (SPINY), the CAM Grant S2013/ICE-3095 (CIBERDINE) and the UK EPSRC Grant EP/L022710/1.

## References

1. Anderson, W.: Entropy library documentation. Google Code Projects (2012)
2. Atmel, C.: Avr109: Self programming (2004). atmel.com/images/doc1644.pdf
3. Atmel, C.: Automotive compilation (2012). http://www.atmel.com/Images/atmel_autocompilation_vol9_oct2012.pdf
4. Backes, M., Nürnberger, S.: Oxymoron: making fine-grained memory randomization practical by allowing code sharing. In: USENIX Security Symposium (2014)
5. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: USENIX Security (2003)
6. Braden, K., Crane, S., Davi, L., Franz, M., Larsen, P., Liebchen, C., Sadeghi, A.R.: Leakage-resilient layout randomization for mobile devices. In: Network and Distributed Systems Security Symposium (NDSS) (2016)
7. Carlini, N., Wagner, D.: Rop is still dangerous: breaking modern defenses. In: USENIX Security Symposium (2014)
8. Cowan, C., Wagle, P., Pu, C., Beattie, S., Walpole, J.: Buffer overflows: attacks and defenses for the vulnerability of the decade. In: DARPA Information Survivability Conference and Exposition, 2000, DISCEX 2000, vol. 2, pp. 119–129. IEEE (2000)
9. Crane, S., Liebchen, C., Homescu, A., Davi, L., Larsen, P., Sadeghi, A.R., Brunthaler, S., Franz, M.: Readactor: practical code randomization resilient to memory disclosure. In: IEEE Symposium on Security and Privacy, S&P, vol. 15 (2015)
10. Davi, L., Liebchen, C., Sadeghi, A.R., Snow, K.Z., Monrose, F.: Isomeron: code randomization resilient to (just-in-time) return-oriented programming. In: Proceedings of the 22nd Network and Distributed Systems Security Symposium (NDSS) (2015)
11. Dean, B.S.: Avr downloader/uploader (2003). http://www.nongnu.org/avrdude/. Accessed Jan 2016
12. Francillon, A., Castelluccia, C.: Code injection attacks on harvard-architecture devices. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 15–26. ACM (2008)

13. Francillon, A., Perito, D., Castelluccia, C.: Defending embedded systems against control flow attacks. In: Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code, pp. 19–26. ACM (2009)
14. Gu, Q., Noorani, R.: Towards self-propagate mal-packets in sensor networks. In: Proceedings of the ACM Conference on Wireless Network Security, pp. 172–182. ACM (2008)
15. Habibi, J., Gupta, A., Carlsony, S., Panicker, A., Bertino, E.: MAVR: code reuse stealthy attacks and mitigation on unmanned aerial vehicles. In: Distributed Computing Systems (ICDCS), pp. 642–652. IEEE (2015)
16. Intel, C.: Hexadecimal object file format specification (1988)
17. Mohan, V., Hamlen, K.W.: Frankenstein: stitching malware from benign binaries. In: 6th USENIX Workshop on Offensive Technologies. USENIX (2012)
18. Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K., Franz, M.: Opaque control-flow integrity. In: Network and Distributed Systems Security Symposium (NDSS) (2015)
19. GNU Project: Avr libc home page (1999). http://www.nongnu.org/avr-libc/. Accessed Jan 2016
20. Roemer, R., Buchanan, E., Shacham, H., Savage, S.: Return-oriented programming: systems, languages, and applications. ACM Trans. Inf. Syst. Secur. (TISSEC) **15**(1), 2 (2012)
21. Sadeghi, A.R., Wachsmann, C., Waidner, M.: Security and privacy challenges in industrial internet of things. In: Annual Design Automation Conference. ACM (2015)
22. Schuster, F., Tendyck, T., Pewny, J., Maaß, A., Steegmanns, M., Contag, M., Holz, T.: Evaluating the effectiveness of current Anti-ROP defenses. In: Stavrou, A., Bos, H., Portokalidis, G. (eds.) RAID 2014. LNCS, vol. 8688, pp. 88–108. Springer, Heidelberg (2014)
23. 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: Security and Privacy (SP), pp. 574–588 (2013)
24. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: eternal war in memory. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 48–62. IEEE (2013)
25. Tang, A., Sethumadhavan, S., Stolfo, S.: Heisenbyte: thwarting memory disclosure attacks using destructive code reads. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 256–267. ACM (2015)
26. Trevennor, A.: Practical AVR Microcontrollers: Games, Gadgets, and Home Automation with the Microcontroller Used in the Arduino. Apress, USA (2012)
27. Wojtczuk, R.: The advanced return-into-lib (c) exploits: Pax case study. Phrack Magazine, vol. 0x0b, Issue 0x3a, Phile# 0x04 of 0x0e (2001)