DisARM: Mitigating Buffer Overflow Attacks on Embedded Devices

by Javid Habibi, Ajay Panicker, Aditi Gupta, and Elisa Bertino
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

DisARM: Mitigating Buffer Overflow Attacks on Embedded Devices

Javid Habibi, Ajay Panicker, Aditi Gupta, and Elisa Bertino
{jhabibi, apanicke, aditi, bertino}@purdue.edu

Purdue University, West Lafayette IN 47907

Abstract. Security of embedded devices today is a critical requirement for the Internet of Things (IoT) as these devices will access sensitive information such as social security numbers and health records. This makes these devices a lucrative target for attacks exploiting vulnerabilities to inject malicious code or reuse existing code to alter the execution of their software. Existing defense techniques have major drawbacks such as requiring source code or symbolic debugging information, and high overhead, limiting their applicability. In this paper we propose a novel defense technique, DisARM, that protects against both code-injection and code-reuse based buffer overflow attacks by breaking the ability for attackers to manipulate the return address of a function. Our approach operates on arbitrary executable binaries and thus does not require compiler support. In addition it does not require user interactions and can thus be automatically applied. Our experimental results show that our approach incurs low overhead and significantly increases the level of security against both code-injection and code-reuse based attacks.

Keywords: Internet of things, return oriented programming, control flow integrity, security, malware, embedded systems

1 Introduction

Recently, everything from refrigerators to sprinkler systems has evolved into smart devices that are pervasively connected to the Internet and powered by embedded processors. These devices are known collectively as the Internet of Things (IoT). Due to their positioning they have the potential to become more prominent in everyday lives than mobile phones. Cisco's Internet Business Solutions Group estimated 12.5 billion connected devices in existence globally as of 2010 with that number doubling to 25 billion by 2015 [15]. However, whereas on one side of the IoT will make possible many novel applications, such as in smart and connected health, on the other side IoT may increase the risk of data privacy breaches and cyber security attacks. A recent study by HP about the most popular devices in some of the most common IoT niches reveal an alarmingly high average number of vulnerabilities per device [14]. On average, 25 vulnerabilities were found per device. For example 80% of devices failed to require passwords of sufficient complexity and length, 70% did not encrypt communications to the Internet and local networks, and 60% contained vulnerable user interfaces and/or vulnerable firmware [14]. Several attacks have already been reported in the past on several other embedded systems such as the ones deployed in cars [47, 44, 10] and sensor networks [33]. We can expect similar attacks to be carried out against IoT embedded devices. Securing embedded devices is thus a critical fundamental step in securing the IoT.

In this paper, we focus on two forms of attacks that exploit buffer overflows on IoT embedded devices, namely code-injection and code reuse attacks. These attacks form a substantial portion of all security attacks due to the fact that buffer overflow vulnerabilities are so common and easy to exploit. Original buffer overflow exploits involved the injection of malicious code [2]. This allows the attacker to subvert the execution of the target program and take control. However, the wide adoption of the $W \oplus X$ protection technique by which all writable addresses are non-executable and vice-versa has rendered code injection attacks ineffective. By contrast, recent code-reuse attacks, such as return-oriented programming (ROP) [39], do not require code injection. These attacks allow an attacker to easily modify the execution path of the target program by reusing existing executable code that primarily exists in the application binary and shared libraries such as `libc`. In code-reuse, the attacker identifies small sequences of instructions, called `gadgets`, that end in a `ret` instruction. By carefully placing a sequence of return addresses on the stack, the attacker can use these gadgets to perform arbitrary computation. ROP attacks have continued to evolve to utilize gadgets that end in both `jmp` or `call` instructions [9].

Since code-reuse based attacks rely on detailed knowledge about the location of code in the executable and libraries, the intuitive solution is to randomize process memory images. Address obfuscation [4] and ASLR [36] are two well-known randomization techniques against such attacks. However, they suffer from the major drawback of small randomization spaces and have been shown to be vulnerable on 32-bit architectures [43, 41]. In considering a new protection technique, we start with two observations. First, the main shortcoming of earlier randomization-based approaches is insufficient entropy, thus making brute-force attacks feasible. Second, the critical step of a buffer overflow attack is to overwrite the return address in order to manipulate the value of the program counter (`PC`).

Our protection technique, referred to as DisARM, introduces a validation technique upon any interaction with the `PC`, (i.e. `ret, call`, or `jmp` instruction). Our validation technique consists of inserting a static check statement before any critical instruction to verify that the program is in the correct state. This invalidates the ability for an attacker to redirect the execution of the target program. By utilizing a hashmap of target addresses XORed with the correct `PC` values a constant time look up is performed upon our constructed hashmap to validate that the program is in the correct state. Upon a failed validation attempt we force the program to exit, thus stopping an attack. Our protection technique has several advantages. First it stops both forms of buffer overflow attacks with minimal overhead. Second it can be applied to any ELF binary without requiring the source code of an application. Finally it offers an alternative to approaches that dynamically monitor critical data sections such as return addresses.

We are not the only researchers to investigate binary modifications for buffer overflow attack mitigation. As discussed in section 2.3, the other approaches suffer from one or more of the following limitations. First, none of the proposed defense techniques is able to mitigate both code injection and ROP based attacks. Second, some of the existing defenses require source code access or other additional information that is generally not available. Third, the overhead of DisARM is constant when compared to the dynamic techniques that incur overhead throughout the execution of the target application. DisARM addresses these limitations and provides a strong and efficient defense techniques against buffer overflow attacks.

As with any defense technique, there are always costs that must be considered. In our proposed defense technique, there is a one time overhead when applying DisARM to the target binary and a runtime overhead during the process execution. We have evaluated the time to apply DisARM to compiled binaries on a selection of commonly used applications and Linux `coreutils`, showing that the performance penalty for DisARM is reasonable in the average case. Our work demonstrates that, although DisARM imposes certain performance costs, its success in thwarting buffer overflow attacks makes DisARM a feasible approach for embedded systems that prioritize execution integrity over optimal performance.

The remainder of this paper is structured as follows. We start by surveying buffer overflow attack and proposed defenses in Section 2. In Section 3, we introduce the target platform for DisARM and describe our approach in more detail. Section 4 discusses the implementation details of DisARM. Section 5 shows the results from various experiments performed to evaluate our approach. Finally we conclude in Section 7.

2 Background and Related Work

In this section we start with a brief summary of attacks based on buffer overflows and existing defense techniques, and then introduce our target platform.

2.1 Code Injection

Code injection attacks are one of the first publicized exploits utilizing a vulnerable buffer. This form of exploit allows the execution of arbitrary code under the attacker's control, potentially allowing the attacker to seize control of an entire program or even an entire system (through exploitation of vulnerable targets with elevated privileges). In order to accomplish this, an attacker injects malicious code into a vulnerable target and then redirects the execution to the injected code [2]. In order to perform such form of attack, several prerequisites must be met. First, the targeted program must have a memory corruption vulnerability. Second, there must be a writable and executable region of memory. Third there must be a way to redirect the processor to execute the injected code. The first and third requirements are generally met through a buffer overflow that allows the attacker to push arbitrary code onto the stack and then overwrite the stack return address to redirect to control the attack payload. The second requirement requires finding an area of memory that can both be written to and executed. The processor then begins to execute the attack payload, granting the attacker control of the current thread.

2.2 Code Reuse

Return oriented programming (ROP) is a technique that evolved from buffer overflow attacks. As discussed in Section 2.1 previous attacks depended on the presence of an executable stack. However the adoption of $W \oplus X$ (also known as Data Execution Prevention – DEP) under which a memory page is either writable or executable, but not both at the same time, has made such attacks ineffective. Code reuse attacks [39] bypass DEP protection. Instead of executing injected code, attackers identify small sequences of instructions, called gadgets, that end in a `ret` instruction. By carefully constructing a sequence of addresses on the software stack, an attacker can manipulate the `ret` instruction to jump to any gadget to perform arbitrary computations. Code reuse techniques work in both word-aligned architectures like RISC [8] and unaligned CISC architectures [39]. These techniques have been shown to be able to perform privilege escalation in Android [17], create rootkits [29], and even inject code into Harvard architectures [23]. Additionally the same technique has been used to manipulate other instructions, such as `jmp`, and their variants [9, 13, 6].

2.3 Defense Techniques

Several defense techniques for mitigating buffer overflow attacks have been proposed. As mentioned before, DEP is the most widely used. However there are a lot of ARM based microcontrollers that do not support DEP as this protection technique was only introduced in ARMv6 and newer architectures [3].

Address obfuscation [4] and ASLR [36] are two well-known defense techniques against ROP attacks. However, they suffer from small randomization and have been shown to be vulnerable on 32-bit architectures [41, 40]. Instruction set randomization (ISR) [31], another well known defense technique, has also been shown to have similar limitations [43]. Several fine grained randomization techniques have been proposed as a defense against code-reuse attacks such as ILR [27], In-place randomization [35], STIR [45], Marlin [26], XIFER [20], Librando [28], Code Shredding [42], ASR [25], Genesis [46], nop-insertion [24] and Bhatkar et al. [5]. Though these defenses have low overhead, they are considerably more invasive in that they require extensive program restructuring which often lead to instability in larger binaries. Also those techniques are not able to account for different optimization levels of binaries and are unable to protect against code-injection based attacks.

Compiler based solutions that create code without return instructions have also been proposed [34, 32]. However those solutions are unable to handle ROP variants such as jump oriented programming [6] attacks. Another mitigation tactic for code reuse attacks is to detect and terminate the attack as it occurs. Examples of these include DROP [11], DynIMA [18], CCFIR [49], CFL [7], ROPdefender [19], [12] and [50]. The dynamic monitoring approach used by these techniques make them unsuitable for our target platform where the processor is limited in comparison to its x86 variants.

Lastly there have been techniques proposed to reinforce the control flow on ARM. Two most notable utilities are MoCFI [16] and control-flow restrictor [37]. However these techniques are both unsuitable for our application. While they are able to reinforce the control flow integrity of a target application, the overhead incurred by the verification is far too great. Within MoCFI, the CPU overhead of the verification grows in relation to the number of jumps as it must traverse the binary graph that is included within the binary after MoCFI has been applied [16]. Pewny takes a different approach by integrating itself within the compiler eliminating the need for disassembly and construction of a control flow graph but the verification process is very long [37]. For each valid target of `1` to `n`, a comparison is made at the end of the function before the final `jump` instruction. This incurs a large CPU overhead within recursive functions or loops making at worst up to `n` function calls. As discussed later DisARM, addresses these issues through the usage of a hashmap.

2.4 Challenges in securing embedded devices

In most x86 based defenses it is acceptable to introduce performance overhead of a factor of 2x [19, 11, 18]. This however is not the case with embedded devices since these devices have low power and very limited resources available. These limited resources include CPU cycles, memory and code size. These were the factors considered in the design of DisARM.

With respect to the limited cycles available, the modifications done to the target binary cannot require too much computation. The reason is that different embedded systems have strict deadlines that must be met and typically operate at very high CPU and memory usage already. Therefore any defense implementation cannot have a large performance impact due to possible interrupts or deadlines that must be met in the protected applications.

x86 vs ARM The x86 architecture's calling convention is set up to mainly use two instructions, one to call a function and one to return from it. The `call` and `ret` assembly instructions are the instructions that control the flow for an application. In addition, there are jump instructions that allow the execution to jump to an address stored in a register. The ARM architecture has many differences in the way in which the flow is controlled. The ARM architecture does not have `call` or `ret` instructions but it has something similar. The ARM assembly includes the use of a linking register, `lr`, that is updated when a function is called with the branch and link instruction `bl`. This works exactly like a `call` instruction, with the difference that the return address is stored into the `lr` register, instead of being pushed onto the stack as in x86. It is thus up to the programmer or compiler to make sure that the value is not lost.

Within the strategy that the ARM architecture utilizes there is a special case to highlight. If the function being utilized does not have any further function calls, it will not ever have an `lr` register update. In order then for these functions to 'return', they branch on the value stored in `lr` by executing the instruction `bx lr`. This implies that the value in `lr` has not changed since the beginning of the function. Due to this, even if there were a vulnerability within such a function, the attacker would not be able to redirect the control since the `lr` register is never pushed onto the stack. However this is not the case for extended nested function calling for which the compiler has to push `lr` onto the stack in order to preserve the return address. Through the combination of pushing `lr` onto the stack and branching, we get the same effect as a `call` in x86.

In order to return from a function, the ARM processor pops the value of the `lr` register that is on the stack into either the `lr` register or the `PC`. Once such action is executed, the program will start to execute the instruction at the address referenced by the old value of the `lr` register which is the address from where the function was called from. By contrast in x86 the `ret` instruction both pops off the stack the address to which the execution has to jump and jumps to such address. Such characteristic of ARM simplifies our defense techniques. Within DisARM we only need to look for and verify any instruction that pops values into the `lr` register or the `PC` as these are the entry points into the execution flow of the program.

3 DisARM Defense Technique

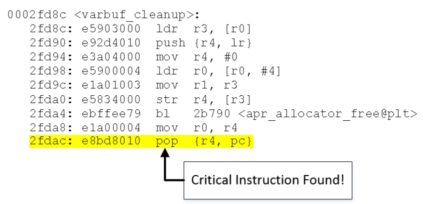
We now describe our DisARM defense technique to mitigate buffer overflow based attacks targeting the Raspberry Pi platform. DisARM uses a fine-grained analysis of the binary to find all critical interactions that manipulate the hardware PC and verifies any change to the PC before the change is applied. For each such critical instruction, we insert a verification block immediately before the critical instruction in order to evaluate whether the target address is valid with respect to the current instruction the program is executing. If the target address fails the verification, the program is forced to exit (see Figure 2). Our technique prevents the attacker from successfully utilizing an overwritten return address to begin a buffer overflow attack. We now introduce our basic assumptions for a buffer overflow attack scenario and then present the details of DisARM.

3.1 Enabling Factors and Attack Assumptions

Based on our survey of buffer overflow based attacks and defenses, we have identified distinct characteristics and requirements for a successful exploit. The fundamental assumption and enabling factor such attacks is as follows:

The attacker is able to modify the return address of the exploited function. That is, if an attacker overflows a buffer, the attacker is able to force the execution to return to a different address than intended and either inject or reuse existing code.

Given this we assume that the vulnerable application must have a buffer overflow or heap overflow vulnerability that can be leveraged by the attacker to inject an exploit payload. Such payload may either contain native machine code to be executed or a string of gadget addresses previously identified by the attacker. The attacker is assumed to have access to the target binary that has undergone DisARM processing. The attacker is also assumed to be aware of the functionality of DisARM. Our approach protects against both remote and local exploits as long as the attacker is not able to modify the target binary while being executed.



```
0002fd8c <varbuf_cleanup>:
   2fd8c: e5903000  ldr  r3, [r0]
   2fd90: e92d4010  push {r4, lr}
   2fd94: e3a04000  mov  r4, #0
   2fd98: e5900004  ldr  r0, [r0, #4]
   2fd9c: e1a01003  mov  r1, r3
   2fda0: e5834000  str  r4, [r3]
   2fda4: ebffee79  bl   2b790 <apr_allocator_free@plt>
   2fda8: e1a00004  mov  r0, r4
   2fdac: e8bd8010  pop  {r4, pc}
```

Fig. 1: Example of a critical instruction

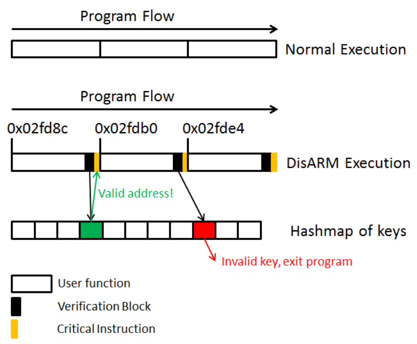


Fig. 2: Example execution of a DisARM'd binary

3.2 Target Platform

The platform targeted in these attacks is the single board computer Raspberry Pi. This is a popular example of an embedded platform and a prime example of the hardware used in IoT devices. Raspberry Pi is based on an ARM11 32-bit RISC processor clocked at 700 MHz which implements the ARMv6k architecture. The ARM11 microprocessor is a Von Neumann Architecture processor. In this architecture, a processor has one physical signal and storage for instructions and data. This allows the processor to load instructions or read/write values from the same section of memory. This is a critical factor in allowing code injection based attacks. In addition, the program space in ARM is 32-bit word aligned with fixed instruction length. Thus, within DisARM we do not have to handle unintended instruction sets from jumping amidst of a variable length instruction. For the purpose of DisARM we look at the ARM instruction set and not the Thumb or Thumb2 instruction set that ARM processors are also compatible with.

3.3 Critical Instructions

Since DisARM reinforces the execution path of a given binary by inserting a verification block before a critical instruction, we must first define what is a critical instruction. A critical instruction is one that takes input from the stack and leads to an update to the `PC`. In the ARM architecture critical instructions are all the instructions of the form `pop {...,pc}` or `pop {...,lr}`, which are instructions that remove the next X values off the top of the stack and immediately set the `PC` to the last of these values (in the pop pc case) or later set the PC to the value popped `bx lr`. No other instructions need to be monitored as discussed in Section 2.4 because of the fact that the ARM architecture is a RISC architecture. An example of a piece of code containing a critical instruction is shown in Figure 3. The code in figure 3 is a code snippet from Apache with a `pop` instruction that updates the values of both registers `r4` and `pc` before the conclusion of the `varbuf_cleanup` function.

3.4 Preprocessing Phase

As mentioned above, DisARM operates at the instruction granularity to reinforce the control flow of the target application. This requires us to identify the critical instructions, within the user defined functions, that require verification. In the preprocessing phase, the ELF binary is thus parsed to extract the critical instructions and their location.

3.5 Hashmap Construction Phase

Once the instructions and their locations have been identified, a key is calculated from the `target address` $\oplus$ `PC` value. Since the number of keys per binary is static, the most efficient solution is to use a hash function that minimizes the number of collisions. The straight-forward solution would be to use a cryptographic hash function but as discussed in section 2.4, CPU cycles are limited. This would not allow for the calculation of a cryptographic hash within every verification block. Instead, due to the nature of the data set, we utilize a minimum perfect hash function (MPHF). The algorithm we use in the generation of a minimum perfect hash is as follows:

1. Given key `K` and square array `S` of dimension `t`, place each key in `S` at location (x,y), where `x = K / t`, `y = K mod t`.
2. Sort each row in `S` in descending order according to the number of elements it contains.
3. Slide each row within `S` of an amount `A` such that no column has more than one entry. Record the shift amount in an array `R`.
4. Collapse `S` into a linear array `C`.

Thus the hash function uses `t` and the displacement `A` calculated in step 3 to locate `K` such that `index = R[x] + y` and `H(K) = C[index]`. For example, if we were to validate the key 15, x = 2 (15 / 6 by integer division) and y = 3 (15 mod 6). Then the index of the key would be at location index = 8 + 3 = 11 (R[2] + 3). As it stands C[11] does in fact equal the key value of 15.

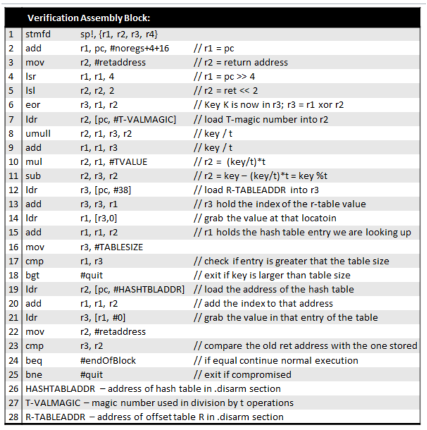
By using the MPHF we can construct a hashmap resulting in the minimum number of collisions over the set of keys storing the target addresses to be validated within the verification blocks. Upon completion of the hashmap generation, the offset array `R` and hashmap array `C` are then appended to the binary in the `.disarm` section.

3.6 PC Lock Phase

After constructing the hashmap, we must lock down each instruction that updates the PC from the stack. To do this we insert our verification block before each critical instruction. By doing this, the relative offsets between instructions are changed and this may affect branch instructions. The reason is that the original destination address for branch instructions is a relative address. Thus, when verification blocks are inserted during the PC lock phase, the targets of these branch instructions are no longer valid and must be corrected to point to the desired location. This is achieved by performing offset patching as discussed next.

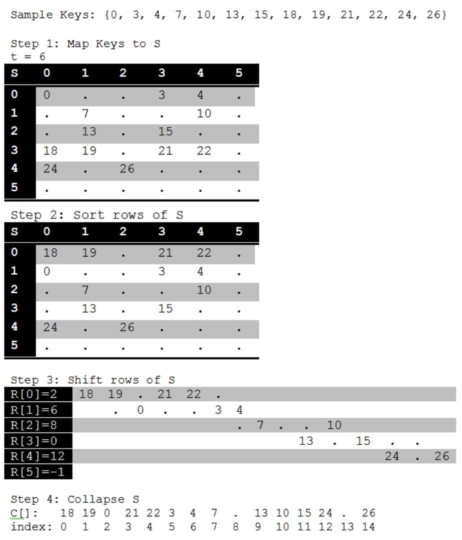
3.7 Patching Phase

Upon completion of the PC lock phase, the target binary needs to be patched in multiple areas. Due to the additions made in the `.text` section for the



```
Verification Assembly Block:
stmfd   sp!, {r1, r2, r3, r4}
add     r1, pc, #noregs+4+16      // r1 = pc
mov     r2, #retaddress           // r2 = return address
lsr     r1, r1, 4                 // r1 = pc >> 4
lsl     r2, r2, 2                 // r2 = ret << 2
eor     r3, r1, r2                // Key K is now in r3; r3 = r1 xor r2
ldr     r2, [pc, #T-VALMAGIC]     // load T-magic number into r2
umull   r2, r1, r3, r2            // key / t
add     r1, r1, r3                // key / t
mul     r2, r1, #TVALUE           // r2 =  (key/t)*t
sub     r2, r3, r2                // r2 = key – (key/t)*t = key %t
ldr     r3, [pc, #38]             // load R-TABLEADDR into r3
add     r3, r3, r1                // r3 hold the index of the r-table value
ldr     r1, [r3,0]                // grab the value at that locatoin
add     r1, r1, r2                // r1 holds the hash table entry we are looking up
mov     r3, #TABLESIZE
cmp     r1, r3                    // check if entry is greater that the table size
bgt     #quit                     // exit if key is larger than table size
ldr     r2, [pc, #HASHTBLADDR]    // load the address of the hash table
add     r1, r1, r2                // add the index to that address
ldr     r3, [r1, #0]              // grab the value in that entry of the table
mov     r2, #retaddress
cmp     r3, r2                    // compare the old ret address with the one stored
beq     #endOfBlock               // if equal continue normal execution
bne     #quit                     // exit if compromised
HASHTABLADDR – address of hash table in .disarm section
T-VALMAGIC – magic number used in division by t operations
R-TABLEADDR – address of offset table R in .disarm section
```

Fig. 3: Verification Block Assembly



Sample Keys: {0, 3, 4, 7, 10, 13, 15, 18, 19, 21, 22, 24, 26}

Step 1: Map Keys to S
t = 6

S	0	1	2	3	4	5
0	0	.	.	3	4	.
1	.	7	.	.	10	.
2	.	13	.	15	.	.
3	18	19	.	21	22	.
4	24	.	26	.	.	.
5	.	.	.	.	.	.

Step 2: Sort rows of S

S	0	1	2	3	4	5
0	18	19	.	21	22	.
1	0	.	.	3	4	.
2	.	7	.	.	10	.
3	.	13	.	15	.	.
4	24	.	26	.	.	.
5	.	.	.	.	.	.

Step 3: Shift rows of S

```
R[0]=2   18 19 .  21 22 .
R[1]=6         .  0  .  .  3  4
R[2]=8                     .  7  .  .  10
R[3]=0                              13 .  15 .  .
R[4]=12                                      24 .  26
R[5]=-1
```

Step 4: Collapse S

```
C[]:   18 19 0  21 22 3  4  7  .  13 10 15 24 .  26
index: 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
```

Fig. 4: Example of MPHF construction process over a sample set of keys

verification blocks and the addition of data for the hashmap in the `.disarm` section, the ELF header needs to patched in addition to all branch and load instructions within the `.text` section that have been shifted during the PC Lock Phase. Within the ELF header, both the program header and the section header table need to be patched to account for their new locations.

After completing the ELF patching, each relative load and branch instruction offset by the modifications needs to be patched to point to the proper location. This is achieved by performing patching as the DisARMed binary is generated. During this process, we utilize the information gathered earlier during the Preprocessing Phase such as the original target location of load and branch. With this information and the number of verification blocks installed, we can calculate the new relative offsets for each branch/load instruction to properly patch the binary. This is discussed in more detail in section 4.4

4 Implementation Details

We have implemented a DisARM prototype that can operate on any C based ELF binary without requiring its source code. The implementation was done for 32-bit ARMv6 architecture on a system running the Raspbian operating system [22]. The implementation of DisARM involved two major components. The first component consists of preprocessing and constructing the hashmap. The second component deals with applying the verification blocks to the PC and patching the binary. We discuss the details of DisARM implementation below.

4.1 Preprocessing

Before we can reinforce the binary, we need to identify the critical instructions and every instruction that will be affected by the installation of the verification blocks. This includes locating all load, relative jump branches, function calls, returns and relative load instructions as these will all needed to be patched. In addition, while parsing the binary, to find these instructions we track the number and location of each critical instruction. To accomplish this we utilized a java based elf parsing library [48]. This library allows us to read in the entire ELF binary and represent it objectually, thus allowing us to easily update the right sections of the file. However this library does not have all the functionality that we were looking for. In its stock form it treated the `.text` section as a block of bytes. We extended this library to also parse each individual instruction byte by byte in order to identify all the information needed for the patching phase.

4.2 Hashmap Construction Phase

In this stage, the hashmap to be utilized in the verification blocks is constructed. The first step is to take each PC value and target address of a critical instruction found, XOR their values to generate a key for that pair, and add them to the key list. After generating all keys that will be used in the hashmap, we can construct the MPHF as described by the algorithm in section 3.5. Once both the MPHF and the hashmap are generated, the two arrays (offset array R and flattened hashmap C) are appended to the binary within their own section.

4.3 PC Lock phase

Upon completion of the hashmap construction phase, we must install the verification blocks at the critical instructions to reinforce the program flow of the target binary. Each critical instruction is read in ascending order and wrapped in the verification block, as shown in figure 5. Each verification block utilizes multiple constants that are used during its execution. These include the MPHF `t` value, the location of the offset table, and the location of the hashmap. During this phase we also track how many verification blocks there were previous to the currently installed block as this information is utilized in the patching phase.

The challenge in this phase was designing and constructing a verification block that does not depend on any external resources aside from the global hashmap and offset table in order to perform that validation. To accomplish this, we followed the same methodology that GCC compiler uses during compilation. Each constant used within the verification block is appended to the end of the verification block after the final branch. This way they are available as immediate values that can be loaded into registers during the execution of a verification block. In addition since each verification block is going to be the primary source of runtime overhead, it had to be minimized. To accomplish this we replaced the expensive division operations with multiplications using magic number constants that provided the same functionality while also hand-optimizing the entire verification process. This reduced the verification block size down to 25 instructions for a combined total of 34 cycles of execution.

4.4 Patching phase

In this stage all patching is performed in the same pass when the DisARMed binary is written. This is done by using the information available from the preprocessing stage. As we process the new binary, whenever a load, relative jump branches, function calls, return or relative load instruction is encountered, we compare the instruction's location within the new binary to the previous binary. If there is a difference due to the installation of a verification block, we generate a patch that will then point the specified instruction to the correct location. However there is a special case with relative load as a relative load instruction and an instruction loading a constant into a register have the exact same signature. To distinguish between these instructions, we analyze the value being 'loaded' into the specified register. If the value is within the program space then the instruction is a relative load and it needs to be patched.

Upon completing the patching for all instructions identified by the preprocessing stage, we must patch the ELF header. The reason is that due to the installation of the verification blocks, the `.text` section has grown which offsets all subsequent sections (`.fini`, `.data`, `.rodata` etc etc...). In addition, we need to insert an additional entry in the program header of the ELF file to recognize the new `.disarm` section we include. This section contains the hashmap and offset table, generated by the hashmap construction phase (see section 4.2). Finally, a patch is applied within the ELF header to the location of `_start` as its location may have shifted during the installation of verification blocks in the `.text` section.

5 Evaluation

We now describe various experiments that we performed to evaluate the DisARM technique. These experiments test the effectiveness of DisARM and also the performance overhead incurred due to the PC verification. These experiments were performed on a Raspberry Pi Model B+ with 512MB of RAM. This Raspberry Pi Model had $W \oplus X$ and ASLR enabled during our experiments. We used `coreutils` binaries, some commonly used application binaries (see figure 6) and byte-unixbench [1] benchmarks to conduct various experiments. In addition to measuring the overhead of instrumenting binaries to apply the DisARM technique, we utilized a Windows 8.1 Machine with an i7 processor and 8GB of RAM. To launch attacks against DisARM-protected binaries, we use ROPgadget (v5.3) [30], an attack tool that automatically creates exploit payload for ROP attacks by searching for gadgets in an application's executable section.

5.1 Effectiveness

First, we tested the effectiveness of DisARM using a test application that has a buffer overflow vulnerability. The application, `elf-ARM-ls` is a test binary as part of the ROPgagdet test binaries. We used ROPgadget on this target application and found 1392 unique gadgets. These were sufficient to craft a code exploit payload. When this exploit payload was provided as an input to the unprotected binary, and we were able to redirect the execution. Next, we applied DisARM to this application to lock down the PC and ran the ROPgadget again to find the new locations of the gadgets. We then executed the DisARM-binary with the new payload and it failed. This highlights the crucial factor of buffer overflow attacks that require them to successfully manipulate the PC by overwriting return addresses.

Application	Version
Apache	2.4.10
Bash	4.3
Coreutils	8.23
Git	2.2.0
Gzip	1.6
Make	4.1
Openssh	6.4
Nano	2.0.6
Tar	1.2.8
Vim	7.4
MySQL	5.6.22
Lighttpd	1.4.35
Perl	5.20.1
Nginx	1.7.8
Python	3.4.2
Monkey	1.5.4

Fig. 5: Target Applications used in the evaluation of DisARM

5.2 DisARM utility overhead

When an application is being deployed to an IoT device, DisARM identifies all critical instructions for verification blocks to be installed, and every instruction that will be affected by the installation of the verification blocks that is used later in the patching phase. This computation is unique to each binary. The next phase involves generating the hashmap from all the critical instructions found and the installation of the verification blocks. The last phase involves patching all instructions that were affected by this instrumentation. DisARM processing cost is the combined overhead of all four phases. We measure DisARM processing overhead on the same set of binaries (see figure 6) used throughout the evaluation.

Our first evaluation of DisARM was to measure the execution overhead of applying it to each test application. We noticed that there was a direct correlation to the number of verification blocks being installed within the application and the total runtime of DisARM. Our average runtime was 193.5 seconds over our sample applications. This is due to the fact that we tested on multiple large binaries that had 3000+ verification blocks installed such as Perl, and Python. However the median execution time 5.23 seconds. This is quite reasonable as the larger the target application, the more processing is required for DisARM to reinforce the control flow. Notice that this overhead is incurred only once. Since DisARM modifies the binary so that each critical instruction is secured, it does not need to be run before every execution of the application.

5.3 Efficiency

Due to the hashmap that DisARM utilizes in the verification process, as discussed in section 3, DisARMed binaries incur a runtime memory overhead. This overhead is constant per binary due to the fact that each hashmap is unique to the target application. We evaluated the memory efficiency of DisARM by taking 14 common applications (see figure 6), we expect to be widely used in IoT

Binary	Original Memory Usage	# of Verification Blocks	Additional Memory Usage (KB)	% increase
Apache	3.6 MB	1287	19.11	0.518
Bash	2.7 MB	2766	63.88	2.310
sftp	332 KB	215	7.95	2.395
Git	524 KB	3983	87.04	16.611
Gzip	220 KB	156	7.34	3.336
Make	110 KB	346	10.37	9.427
Nano	852 KB	817	19.05	2.236
Tar	206 KB	891	17.64	8.563
Vim	1.2 MB	5836	148.03	12.047
Lighttpd	1.2 MB	369	17.32	1.410
Perl	2.8 MB	3509	470.41	16.407
Nginx	1.1 MB	1845	46.79	4.154
Python	3.2 MB	6270	368.87	11.257
Monkey	152 KB	650	18.58	12.224

Table 1: DisARM memory overhead

devices, and measured their idle memory usage. Unfortunately for some of the target applications it was not possible to measure the idle memory usage since the memory usage of the application directly correlated to size of its input. Our results, reported in Table 2 show a maximum of 17%, median of 4.1% and an average of 7% increase in memory at runtime. The reason is that, the size of the hashmap directly correlates to how many functions there are within the application and how often they are utilized. We found this overhead to be acceptable as in most tested applications there was less then an average of 10% increase in memory usage.

5.4 Code Size

In addition to the memory overhead, we measured the increase in code size of these applications. The increases is due to the modifications we had made to the applications in order to insert the verification blocks and hashmap. We thus wanted to see if there were significant changes in code size. Our results, reported in table 3 show that the code sizes do not significantly increase. As shown in table 3 there is a maximum of 0.21% increase in code size with a median of 0.1% and an average of 0.13%. The reason is that DisARM does not require any special compiler flags to be enabled on target binaries and can operate at any optimization level thus taking advantage of all existing code reduction techniques.

5.5 DisARM Runtime overhead

We measured the runtime overhead of DisARMed binaries to see if the application of DisARM greatly affects the execution time of a binary. For this purpose we use the byte-unixbench binaries. We used the benchmark scores of the stock

Binary	Original Size (MB)	DisARM Size (MB)	% increase
Apache	1.5	1.66	0.10
Bash	2.5	2.86	0.14
sftp	0.30	0.34	0.10
Git	5.6	6.11	0.09
Gzip	0.261	0.28	0.09
Make	0.51	0.56	0.09
Nano	0.499	0.60	0.21
Tar	1.2	1.31	0.09
Vim	5.7	6.47	0.13
Lighttpd	0.631	0.67	0.09
Perl	1.5	1.87	0.56
Nginx	2.8	3.00	0.09
Python	6.7	7.37	0.15
Monkey	3.9	3.97	0.02

Table 2: DisARM Code Size

binaries as a baseline to compare with DisARMed benchmarks. We applied DisARM to each benchmark and took the average of scores from each run. We observed that the benchmark scores were not greatly affected by the application of DisARM to the binaries. The average score generated from the stock Unixbench is 78.4 with a median of 78.8 as opposed to the DisARMed Unixbench which has an average of 76.2 and a median of 76.4. The reason is that each verification block installed consists of only 25 instructions which equates to an additional 34 cycles. This equates to an additional 48 nanoseconds of execution time per function with the processor clocked at 700 MHz. These results support our initial claim of minimal runtime overhead per DisARMed binary.

6 Conclusions and Future Work

In this paper, we proposed a verification technique to defend against buffer overflow attacks. This approach installs verification blocks at critical instructions preventing the attacker from manipulating return addresses. We have implemented a prototype of our approach and demonstrated that it is successful in defeating buffer overflow attacks crafted using automated attack tools. We have also evaluated the effectiveness of our approach and showed that the effort to exploit DisARM is significantly high. Based on the results of our analysis and implementation we argue that fine-grained verification is both feasible and practical as a defense against these persistent buffer overflow based attack techniques. Future work will evaluate the effectiveness of this strategy on CISC architectures, such as x86, in addition to exploring techniques that would allow us to apply DisARM to a binary that has been stripped of all symbol information making the deployment of DisARM even more seamless.

Acknowledgement. The work reported in this paper has been partially supported by the Purdue Cyber Center and the National Science Foundation under grant CNS-1111512.

References

1. byte-unixbench: A Unix benchmark suite. `http://code.google.com/p/byte-unixbench/`.
2. Aleph One. Smashing the stack for fun and profit. *Phrack Magazine 49*, 14 (November 1996).
3. ARM Holdings plc. ARM Architecture Reference Manual.
4. Bhatkar, E., Duvarney, D. C., and Sekar, R. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *In Proc. of the 12th USENIX Security Symposium* (2003), pp. 105–120.
5. Bhatkar, S., Sekar, R., and DuVarney, D. C. Efficient techniques for comprehensive protection from memory error exploits. In *Proc. of the 14th conference on USENIX Security Symposium - Volume 14* (2005), SSYM'05, pp. 17–17.
6. Bletsch, T., Jiang, X., and Freeh, V. Jump-oriented programming: A new class of code-reuse attack. Tech. Rep. TR-2010-8, North Carolina State University, 2010.
7. Bletsch, T., Jiang, X., and Freeh, V. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference* (New York, NY, USA, 2011), ACSAC '11, ACM, pp. 353–362.
8. Buchanan, E., Roemer, R., Shacham, H., and Savage, S. When good instructions go bad: generalizing return-oriented programming to risc. In *Proc. of the 15th ACM conference on Computer and communications security* (2008), pp. 27–38.
9. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. Return-oriented programming without returns. In *Proc. of the 17th ACM conference on Computer and communications security* (2010), pp. 559–572.
10. Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., and Kohno, T. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX Conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 6–6.
11. Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L. DROP: Detecting return-oriented programming malicious code. In *Proc. of the 5th International Conference on Information Systems Security* (2009), pp. 163–177.
12. Chen, P., Xing, X., Han, H., Mao, B., and Xie, L. Efficient detection of the return-oriented programming malicious code. In *Proc. of the 6th international conference on Information systems security* (2010), pp. 140–155.
13. Chen, P., Xing, X., Mao, B., and Xie, L. Return-oriented rootkit without returns (on the x86). In *Proc. of the 12th international conference on Information and communications security* (2010), pp. 340–354.
14. Daniel Miessler. HP Study Reveals 70 Percent of Internet of Things Devices Vulnerable to Attack. `http://h30499.www3.hp.com/t5/Fortify-Application-Security/HP-Study-Reveals-70-Percent-of//-Internet-of-Things-Devices/ba-p/6556284#.VH4faTHF9Zg`, July 2014.
15. Dave Evans. The Internet of Things How the Next Evolution of the Internet is Changing Everything. `http://www.cisco.com/web/about/ac79/docs/innov/IoT_IBSG_0411FINAL.pdf`, April 2011.

16. Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nürnberger, S., and Sadeghi, A.-R. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *NDSS* (2012).
17. Davi, L., Dmitrienko, A., Sadeghi, A.-R., and Winandy, M. Privilege escalation attacks on android. In *Proc. of the 13th international conference on Information security* (2011), pp. 346–360.
18. Davi, L., Sadeghi, A.-R., and Winandy, M. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proc. of the 2009 ACM workshop on Scalable trusted computing* (2009), pp. 49–54.
19. Davi, L., Sadeghi, A.-R., and Winandy, M. ROPdefender: a detection tool to defend against return-oriented programming attacks. In *Proc. of the 6th ACM Symposium on Information, Computer and Communications Security* (2011), pp. 40–51.
20. Davi, L. V., Dmitrienko, A., Nürnberger, S., and Sadeghi, A.-R. Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security* (New York, NY, USA, 2013), ASIA CCS '13, ACM, pp. 299–310.
21. Debian. ElfUtils. `https://packages.debian.org/sid/elfutils`, 2014.
22. Debian Foundation. Raspbian. `http://www.raspbian.org/`.
23. Francillon, A., and Castelluccia, C. Code injection attacks on harvard-architecture devices. In *Proc. of the 15th ACM conference on Computer and communications security* (2008), pp. 15–26.
24. Franz, M., Brunthaler, S., Larsen, P., Homescu, A., and Neisius, S. Profile-guided automated software diversity. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)* (Washington, DC, USA, 2013), CGO '13, IEEE Computer Society, pp. 1–11.
25. Giuffrida, C., Kuijsten, A., and Tanenbaum, A. S. Enhanced operating system security through efficient and fine-grained address space randomization. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 40–40.
26. Gupta, A., Habibi, J., Kirkpatrick, M., and Bertino, E. Marlin: Mitigating code reuse attacks using code randomization. *Dependable and Secure Computing, IEEE Transactions on PP*, 99 (2014), 1–1.
27. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., and Davidson, J. W. Ilr: Where'd my gadgets go? In *Proc. of the 2012 IEEE Symposium on Security and Privacy* (2012), pp. 571–585.
28. Homescu, A., Brunthaler, S., Larsen, P., and Franz, M. Librando: transparent code randomization for just-in-time compilers. In *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 993–1004.
29. Hund, R., Holz, T., and Freiling, F. C. Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In *Proc. of the 18th conference on USENIX security symposium* (2009), SSYM'09, pp. 383–398.
30. Jonathan Salwan. ROPgadget tool. `http://shell-storm.org/project/ROPgadget/`.
31. Kc, G. S., Keromytis, A. D., and Prevelakis, V. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2003), CCS '03, ACM, pp. 272–280.

32. Li, J., Wang, Z., Jiang, X., Grace, M., and Bahram, S. Defeating return-oriented rootkits with "return-less" kernels. In *Proc. of the 5th European conference on Computer systems* (2010), pp. 195–208.
33. Newsome, J., Shi, E., Song, D., and Perrig, A. The sybil attack in sensor networks: Analysis & defenses. In *Proceedings of the 3rd International Symposium on Information Processing in Sensor Networks* (New York, NY, USA, 2004), IPSN '04, ACM, pp. 259–268.
34. Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., and Kirda, E. G-free: defeating return-oriented programming through gadget-less binaries. In *Proc. of the 26th Annual Computer Security Applications Conference* (2010), pp. 49–58.
35. Pappas, V., Polychronakis, M., and Keromytis, A. D. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (Washington, DC, USA, 2012), SP '12, IEEE Computer Society, pp. 601–615.
36. PaX Team. PaX. `http://pax.grsecurity.net/`.
37. Pewny, J., and Holz, T. Control-flow restrictor: Compiler-based cfi for ios. In *Proceedings of the 29th Annual Computer Security Applications Conference* (2013), ACM, pp. 309–318.
38. Raspberry Pi Foundation. Raspberry Pi Hardware.
39. Roemer, R., Buchanan, E., Shacham, H., and Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur. 15*, 1 (Mar. 2012), 2:1–2:34.
40. Roglia, G., Martignoni, L., Paleari, R., and Bruschi, D. Surgically returning to randomized lib(c). In *Computer Security Applications Conference, 2009. ACSAC '09. Annual* (dec. 2009), pp. 60 –69.
41. Shacham, H., Page, M., Pfaff, B., Goh, E.-J., Modadugu, N., and Boneh, D. On the effectiveness of address-space randomization. In *Proc. of the 11th ACM conference on Computer and communications security* (2004), pp. 298–307.
42. Shioji, E., Kawakoya, Y., Iwamura, M., and Hariu, T. Code shredding: Byte-granular randomization of program layout for detecting code-reuse attacks. In *Proceedings of the 28th Annual Computer Security Applications Conference* (New York, NY, USA, 2012), ACSAC '12, ACM, pp. 309–318.
43. Sovarel, A. N., Evans, D., and Paul, N. Where's the feeb? the effectiveness of instruction set randomization. In *Proc. of the 14th conference on USENIX Security Symposium - Volume 14* (2005), pp. 10–10.
44. Verdult, R., Garcia, F. D., and Balasch, J. Gone in 360 seconds: Hijacking with hitag2. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2012), Security'12, USENIX Association, pp. 37–37.
45. Wartell, R., Mohan, V., Hamlen, K. W., and Lin, Z. Binary stirring: self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 157–168.
46. Williams, D., Hu, W., Davidson, J., Hiser, J., Knight, J., and Nguyen-Tuong, A. Security through diversity: Leveraging virtual machine technology. *Security Privacy, IEEE 7*, 1 (Jan 2009), 26–33.
47. Wright, A. Hacking cars. *Commun. ACM 54*, 11 (Nov. 2011), 18–19.
48. Xiao-Feng Li. ELF Parser. `http://people.apache.org/~xli/`.
49. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., and Zou, W. Practical control flow integrity and randomization for binary executables. In *IEEE Symposium on Security and Privacy* (2013), IEEE Computer Society, pp. 559–573.

50. Zhang, M., and Sekar, R. Control flow integrity for cots binaries. In *Proceedings of the 22Nd USENIX Conference on Security* (Berkeley, CA, USA, 2013), SEC'13, USENIX Association, pp. 337–352.