RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications

Taegyu Kim
Purdue University
tgkim@purdue.edu

Chung Hwan Kim
NEC Laboratories America
chungkim@nec-labs.com

Hongjun Choi
Purdue University
choi293@cs.purdue.edu

Yonghwi Kwon
Purdue University
kwon58@cs.purdue.edu

Brendan Saltaformaggio
Georgia Institute of Technology
brendan@ece.gatech.edu

Xiangyu Zhang
Purdue University
xyzhang@cs.purdue.edu

Dongyan Xu
Purdue University
dxu@cs.purdue.edu

ABSTRACT

ARM is the leading processor architecture in the emerging mobile and embedded market. Unfortunately, there has been a myriad of security issues on both mobile and embedded systems. While many countermeasures of such security issues have been proposed in recent years, a majority of applications still cannot be patched or protected due to run-time and space overhead constraints and the unavailability of source code. More importantly, the rapidly evolving mobile and embedded market makes any platform-specific solution ineffective. In this paper, we propose RevARM, a binary rewriting technique capable of instrumenting ARM-based binaries without limitation on the target platform. Unlike many previous binary instrumentation tools that are designed to instrument binaries based on x86, RevARM must resolve a number of new, *ARM-specific* binary rewriting challenges. Moreover, RevARM is able to handle *stripped binaries*, requires no symbolic/semantic information, and supports *Mach-O binaries*, overcoming the limitations of existing approaches. Finally, we demonstrate the capabilities of RevARM in solving real-world security challenges. Our evaluation results across a variety of platforms, including popular mobile and embedded systems, show that RevARM is highly effective in instrumenting ARM binaries with an average of 3.2% run-time and 1.3% space overhead.

ACM Reference Format:
Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2017. RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications. In *Proceedings of ACSAC 2017*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3134600.3134627

1 INTRODUCTION

ARM is the de facto standard for a variety of mobile and embedded platforms — including smartphones and tablet computers, the "Internet of Things" (IoT) devices, unmanned aerial vehicles (UAVs), and other robotic vehicle systems. Unfortunately, as ARM-based systems gain popularity, security threats to these systems have also increased significantly. Many severe security vulnerabilities have been exposed recently in both mobile devices and embedded devices. For example, popular mobile platforms have experienced private information leaks through private API abuse [19, 63], many IoT devices have been exploited by malicious attackers for the invasion of victims' privacy [15, 17], and more importantly, compromised UAVs threaten privacy, financial loss, and even human lives [4]. These incidents happen mainly because many ARM-based devices remain unpatched and vulnerable to malicious attacks [28, 33].

Considering a large number of such applications on ARM-based platforms are not open-source, a precise ARM *binary* rewriting technique is highly desired. For example, most apps for iOS and Android smartphones and tablets are closed-source. App developers submit only the binary files of the apps to the marketplaces and even the distributors (e.g., Google and Apple) do not have access to the source code. While the submitted binary files go through the vendor's vetting process for security, privacy, and reliability, it has been shown that such vetting processes can be easily tricked by attackers [42]. Besides mobile applications, embedded systems also commonly deploy only closed-source binaries. A few examples include flight controller software for UAVs [11], IoT devices [21], and robotic vehicles [32]. Security challenges in such binary-only software can all be resolved with an accurate and practical binary rewriting technique. For example, during the existing app vetting process, one can instrument function calls in order to prevent the use of private APIs, which has been reported as a major threat in iOS platforms [42]. Such a binary rewriting technique is also much in demand by embedded systems. For instance, S.F. Express (one of China's leading logistics providers) has recently adopted UAVs for package delivery [10]. However, their UAVs run closed-source embedded software [11], and attackers have already found ways to hijack UAVs [9]. Consequently, it is highly desirable for service providers to be able to secure their software through binary rewriting techniques that are ready to be deployed in such cases.

Unfortunately, despite the pressing need for a highly effective ARM binary rewriting technique, existing techniques have several limitations. Dynamic analysis techniques incur large run-time and space overhead which often leads to high energy consumption. They are hardly deployed due to the nature of resource-scarce embedded system environments. Moreover, many existing dynamic analysis techniques [37, 40, 45, 50, 51] do not support non-rooted mobile devices, hence limiting their applicability. Many static binary instrumentation techniques cannot instrument stripped binaries which

ACSAC 2017, December 4–8, 2017, Orlando FL, USA

ACM ISBN 978-1-4503-5345-8/17/12…$15.00
https://doi.org/10.1145/3134600.3134627

Table 1: Comparison of static binary rewriting techniques for stripped binaries. I: Fine-grained instrumentation at arbitrary locations, MP: Multiple platform support, LSB: Large-scale binary support, RO: Run-time overhead, SO: Space overhead.

	Target	I	MP	LSB	RO	SO
BISTRO [43]	x86	✓	✗	✓	Medium	Medium
Uroboros [62]	x86	✓	✗	✓	Low	Low
SecondWrite (2011) [52]	LLVM	✓	✓	✗ [2, 61, 65]	Low	High
SecondWrite (2013) [34]	LLVM	✓	✓	✗ [2, 61]	Low	High
Dyninst [39]	x86	✗	✓	✓	High	High
Pebil [49]	x86	✗	✗	✓	Medium	High
REINS [65]	x86	✓	✗	✓	Low	High
PSI [68]	x86	✓	✗	✓	High	High
RevARM	ARM	✓	✓	✓	Low	Low

Table 2: Differences between the ARM and x86 architectures.

	ARM	x86
Branch instructions	`B`, `BL`, `BLX`, `IT`*, `TBB`*, `TBH`*, etc	`CALL`, `JMP`, `JE`, etc
Program counter	Readable and writable	Not directly accessible
Instruction modes	ARM and Thumb modes	-
Instruction lengths	Fixed (16 or 32 bits)	Variable lengths

* These instructions are unique in ARM.

do not have symbolic or relocation information [41, 44, 54, 55]. Further, most instrumentation techniques are incapable of supporting ARM binaries, instrumenting any instruction at arbitrary locations, or providing platform-agnostic instrumentation. We summarize the limitations of the existing techniques in Table 1.

To overcome the above issues, we propose RevARM, a platform-agnostic ARM-based binary rewriting technique for security applications. Unlike existing *trampoline*-based approaches which introduce additional control flow to the instrumented program (leading to a large overhead), RevARM leverages an insertion/replacement-based approach ("insertion-based" for short) which inserts and replaces ARM instructions with negligible run-time and space overhead. Further, our insertion-based approach enables fine-grained instrumentation at *arbitrary* binary locations. This capability enables powerful security applications that other approaches are not able to support, such as instruction-level code diversification and advanced software fault isolation (SFI) enforcement [56, 67] which we demonstrated in §4.

To enable insertion-based ARM binary instrumentation, RevARM addresses a number of challenges unique to the ARM architecture, which prior work in x86 binary rewriting did not encounter/handle. Based on our thorough analysis of ARM and x86, we found several fundamental differences between the two architectures, summarized in Table 2. These differences motivated our design of RevARM to overcome four key challenges unique to ARM binary rewriting: (1) the If-Then instruction, (2) branch table instructions, (3) direct access to the program counter (`PC`), and (4) run-time instruction mode switching. Further, RevARM supports both Mach-O and stripped ARM-based binaries, which cover a majority of ARM-based mobile and embedded platforms.

In summary, the contributions of this paper are as follows:

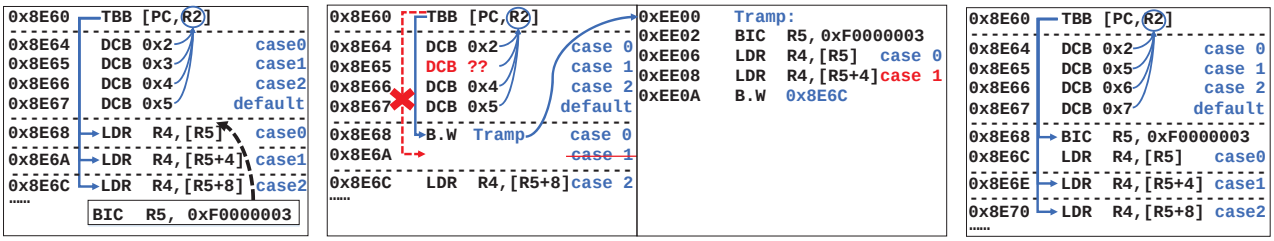
- We present the design and implementation of the RevARM technique. To the best of our knowledge, RevARM is the first fine-grained platform-agnostic ARM binary rewriting technique that is able to instrument instructions at arbitrary binary locations.
- We introduce a number of previously-unresolved, ARM-specific challenges that must be addressed to enable insertion-based ARM binary rewriting, and show how RevARM overcomes these challenges in detail.
- We demonstrate the effectiveness of RevARM in security applications through a number of case studies: inserting `NOP` instructions for code diversification, patching vulnerable functions using extracted function binary code, preventing private API abuses in iOS with SFI, and fine-grained run-time status monitoring of a UAV control system. Our evaluation results show that RevARM introduces only negligible run-time and space overhead while providing powerful ARM binary rewriting capabilities.

2 BACKGROUND AND MOTIVATION

In general, binary instrumentation techniques can be categorized into two groups: *trampoline-based* and *insertion-based*. Trampoline-based instrumentation can be further divided into *detour-based* and *patch-based*. In this section, we discuss why we chose to use insertion-based instrumentation.

Detour-based Instrumentation: Detour-based instrumentation techniques, such as Dyninst [39], Etch [54], and Detours [45], overwrite original instructions at a target instrumentation point with a branch instruction. Whenever the branch instruction is executed, it passes control to a newly inserted instruction block, called a *trampoline*. Trampolines contain both the added instrumentation logic and the original instructions overwritten by the branch instruction. This approach introduces new control flows to and from the trampoline, which incur run-time overhead and space overhead (for the inserted trampoline and control transfer code).

Further, there are corner cases that detour-based instrumentation cannot handle, hence the correctness of the instrumented program cannot be guaranteed. Fig. 1 shows an example case which detour-based instrumentation cannot correctly handle. In this example, the binary code represents a switch statement in C with four switch cases (`case 0`–`2` and `default`). The value of `R2` determines which case will be selected. The goal of the instrumentation is to limit the range of memory addresses for the `LDR` instruction in `case 0`. As described in Fig. 1a, `BIC` is inserted to limit the range of `R5` — which the load instruction takes as an operand. A detour-based instrumentation creates a trampoline (`Tramp`) and overwrites two original instructions (the `LDR` instructions) with a branch instruction (`B.W Tramp`). The trampoline contains the added instruction (`BIC`) and the two original instructions. The branch instruction at the end (`B.W 0x8E6C`) of the trampoline ensures that the rest of the original program is executed after the trampoline. It is important that a long-range branch instruction (four bytes) is used here because the location of the trampoline can be far from the target instrumentation point. However, this violates the correctness of the program since the original instructions may span multiple basic blocks. As described in Fig. 1b, there is a control flow from the switch table to `case 1`. In this case, the switch table does not know that `case 1`



(a) Before instrumentation. (b) Detour-based approach. (c) Our approach.

Figure 1: Comparison between detour-based instrumentation and our approach.

was also moved to the trampoline, and thus the program will show unexpected behavior.

Patch-based Instrumentation: Patch-based approaches [49, 64] duplicate the target code to a new location. The instrumentation is then applied to the duplicated code but not the original code. The original code is modified to pass control to the duplicated code. The aforementioned problem of detour-based instrumentation is solved in this approach since new control flows are introduced while preserving the original instructions. However, introduced control flows leads to large run-time overhead. Further, this approach introduces large space overhead for the duplicated code.

Our Approach: RevARM leverages insertion-based instrumentation, which directly inserts new instructions into target instrumentation points without creating a trampoline or new control flow transitions. In comparison with the two other approaches, RevARM enables fine-grained instrumentation. In other words, RevARM neither introduces complex control flow transitions that can jeopardize program stability (like trampoline-based instrumentation) nor duplicates original code which introduces large space overhead (like patch-based instrumentation). Instead, RevARM *stretches* the target binary to create slots for the new instructions to be inserted while preserving the control flow. For example, inserting target instructions into the stretched switch case as described in Fig. 1c. This approach has two advantages. First, RevARM can achieve binary instrumentation with very low run-time and space overhead, which makes it a practical technique for mobile and embedded systems with limited resources. Further, RevARM is more versatile in enforcing a variety of security applications. For example, RevARM can enforce in-place, fine-grained code diversification by randomizing the code address space to prevent control flow hijacks. However, *trampoline*-based approaches cannot enable this since they cannot perform in-place instruction insertion at arbitrary locations. Finally, RevARM can enforce advanced SFI [56, 67] which requires changing the binary layout to guarantee that all indirect control flows pass through SFI instructions (see §4).

3 DESIGN

3.1 Overview

Fig. 2 illustrates the overall design of RevARM. Overall, our binary rewriting procedure goes through two stages: preprocessing and instrumentation. In the preprocessing stage, RevARM takes a binary file and an instrumentation specification as inputs. An *instrumentation specification* contains instrumentation information, such as

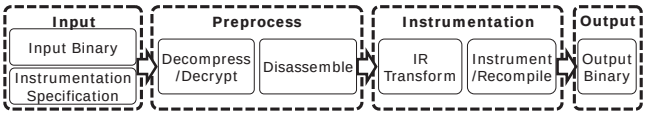


Figure 2: Overview of RevARM.

security policies to enforce and target embedding/extraction location for when RevARM instruments the input binary (i.e., to embed new logic or extract existing logic).

The input binary is first decompressed or decrypted if it is in a compressed or encrypted format. Then, RevARM disassembles the input binary and interprets the instrumentation specification. During the instrumentation stage, RevARM's IR transformer extracts all the information that it needs from the input binary — including instructions/data, instruction mode, type, size, reference — and creates a representation, called *internal representation (IR)*. Based on the instrumentation specification, the instrumentation algorithm inserts or replaces instructions to instrument the input binary. Finally, RevARM updates the metadata of the input binary with the new locations of code and data after instrumentation.

3.2 Preprocessing

Preprocessing consists of two steps: decompression/decryption and disassembling. Our target input binaries can be Mach-O binaries or stripped/unstripped binaries (e.g., UAV firmware binaries). Decompression/decryption should be performed by leveraging existing techniques [22, 23, 25]. Then, RevARM disassembles the input binary. During disassembly, it may be necessary to identify the memory location of a firmware image for proper disassembly. To resolve this problem, RevARM performs analysis of the binary's jump tables, indirect branch target values, and memory access patterns to locate the firmware image region [57].

3.3 Instrumentation

In this section, we explain our instrumentation algorithm to handle code sections. Other sections including data and metadata are described in §3.9. To instrument an input binary (disassembled in the previous stage), RevARM determines what instructions will be inserted or replaced based on the input instrumentation specification. Our algorithm takes a disassembled binary, P_{bin}, and a set of new instructions to insert or replace (INS) as inputs. RevARM follows

P:	Program (P := <C, D>, where C := IR | IR+C)		
C:	Code section	D:	Metadata section
IR:	List of IR	I:	Insertion Instruction
F:	Function Address List	FM:	Mapping Function Entry to IR
INSType	instype := INSERTION | REPLACE;		
INSPosition	inspos := BEFORE | AFTER;		
INS	ins := ins | ins · < IRnew, instype, IRpos, inspos>		

Figure 3: Definitions of variables used in Algorithms 1-3.

Algorithm 1 Pseudo-code to translate the input binary into IRs.

```
1: function TranslateToIR(P_bin)                    ▷ Transform instruction/data into IR in a code section
2:     P_IR := empty
3:     for each c_i ∈ P_bin.C do
4:         ir_i := GetCorrespondingIR(c_i)                    ▷ Basic and common translation
5:         ir_i.caddr := GetConcreteAddr(c_i)                 ▷ Actual address for an instruction/data
6:         ir_i.instmode := GetInstMode(c_i)                  ▷ Determine the current instruction mode
7:         ir_i.rang := GetReferRange(c_i)                    ▷ Get reference range
8:         ir_i.cref := GetReference(c_i)                     ▷ Get target reference instruction/data
9:         if ir_i.type is an IT instruction then             ▷ Get IT information
10:            ir_i.IT_cond := GetITCond(c_i)
11:            ir_i.IT_children := GetITChildren(c_i)         ▷ Get instructions influenced by the IT
12:        end if
13:        P_IR := P_IR · ir_i                                ▷ Insert initialized ir_i to P_IR
14:    end for
15:    for each ir_i ∈ P_IR where ir_i.type is Reference do   ▷ Get reference info on each ir
16:        ir_i.ref := GetReferenceIR(ir_i.cref, P_IR)
17:    end for
18:    return P_IR
19: end function
```

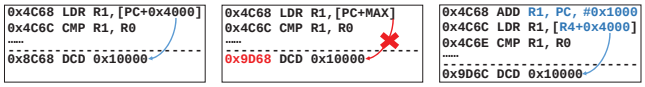
three steps: TranslateToIR, InstrumentCode, and AdjustBinaryLayout. Then, it generates instrumented IRs and exports the IRs into a new binary. We describe our algorithms using the definitions shown in Fig. 3.

Translating Instructions to IRs: In this step, RevARM converts each disassembled code section into IRs and Algorithm 1 presents the pseudo-code. Specifically, RevARM first stores basic information such as instruction address, instruction type, used registers and immediate values in each IR (line 4). Also, it stores ARM-specific information including the current instruction mode, reference, reference range, condition and related instruction/data addresses (lines 5-8). We note that related instruction/data address is any instructions or data used to perform addressing. For example, the entire 32-bit address space cannot be addressed with only one four-byte instruction because these instructions cannot include the entire addresses and the opcode. Furthermore, RevARM stores If-Then (IT) instructions' information. IT instructions make following instructions to be executed conditionally. We will describe more details in §3.4. RevARM should store IT conditions and all child instructions controlled by the current IT (lines 9-12). We note that child instructions are sequential, and the number of IT_children and IT_cond will be identical. Lastly, RevARM stores the converted IRs in P_{IR} (line 13). After generating the IRs, RevARM creates reference pointers to any code segments referenced by each IR (lines 15-17). Using such pointers, we can find the target reference instruction or data even after binary layout modification. Then, the algorithm returns a set of IRs (line 18).

Instrumentation: Algorithm 2 presents the pseudo-code for the instrumentation step. It inserts and replaces instrumentation instructions and data (lines 4-26). We note that there may be many replacement instruction/data objects (lines 7-10). In addition, RevARM records any changes to instruction/data addresses and sizes. This is necessary to later adjust reference targets in the stretched binary.

Algorithm 2 Pseudo-code for code instrumentation.

```
1: function InstrumentCode(P_IR, INS)
2:     P_IR' := empty
3:     stretchedSize := 0                          ▷ Stretched size by the instrumentation
4:     for each ir_cur ∈ P_IR do                   ▷ Instrument all codes
5:         ir_instr := empty
6:         if < ir_new, instype, ir_p, inspos > ∈ INS where ir_cur = ir_p then
7:             if instype is REPLACE then          ▷ Instruction replacement
8:                 stretchedSize += sizeof(ir_cur) - sizeof(ir_instr)
9:                 ir_cur.stretchedSize := stretchedSize
10:                ir_instr := ir_instr · ir_new
11:            else if instype is INSERT then      ▷ New instruction insertion
12:                if inspos is BEFORE then        ▷ Insertion before the current instruction
13:                    ir_cur.stretchedSize := stretchedSize + sizeof(ir_instr)
14:                    ir_instr := ir_instr · ir_new · ir_cur
15:                else                            ▷ Insertion after the current instruction
16:                    ir_cur.stretchedSize := stretchedSize
17:                    ir_instr := ir_instr · ir_cur · ir_new
18:                end if
19:                stretchedSize += sizeof(ir_instr)   ▷ Update stretched size after adjustment
20:            end if
21:        else
22:            ir_cur.stretchedSize := stretchedSize   ▷ Adjust the concrete address
23:            ir_instr := ir_cur
24:        end if
25:        P_IR' := P_IR'·ir_instr                 ▷ Store instrumented ir_instr
26:    end for each
27:    for each ir_cur ∈ P_IR' do                  ▷ Fix all erroneous references
28:        ir_cur := ExtendReferenceInst(ir_cur)   ▷ Make an instruction reachable to its target
29:        ir_cur := AdjustDerefTargets(ir_cur)    ▷ Fix all call targets
30:        ir_cur := AdjustBranchStructure(ir_cur) ▷ e.g., IT instruction
31:    end for each
32:    return P_IR'
33: end function
```



(a) Before instrumentation. **(b) Unreachable data after stretching the binary layout.** **(c) Correctly extending the short reference range by RevARM.**

Figure 4: An example where RevARM makes unreachable data reachable by supplementing the short reference range.

RevARM then extends the code segments to make branch/data loading instructions reachable (lines 27-31). Note that the insertion of instructions/data usually causes stretched distances between referencing/referenced instructions/data. As mentioned before, this may push a referenced instruction beyond the reach of a referencing four-byte instruction. To resolve this problem, it is necessary to replace unreachable instructions with one or multiple reachable instructions. For example, a two-byte instruction can be replaced with a four-byte instruction by substituting a four-byte LDR for a two-byte LDR. However, there are cases where an LDR or VLDR instruction's offset must also be updated to reach a target address. In this case, RevARM prepends an ADD instruction. Lastly, RevARM adjusts all reference instructions/data to ensure that they point to their original targets. We describe such a case in Fig. 4. Note that RevARM does not consume an additional register in Fig. 4 as RevARM uses a destination register which will store the result of the LDR. However, there are cases where such registers are unavailable, such as VLDR which does not change any general register. To complement this case, RevARM checks any following instructions and searches for any overwritten register without a subsequent read in the possible paths. Otherwise, RevARM adds a PUSH and POP for a used register to reference the target data. Note that RevARM

Algorithm 3 Pseudo-code to adjust the layout of the instrumented binary.

```
1:  function AdjustBinaryLayout(P_IR)
2:      stretchedSize := 0
3:      for each ir_cur ∈ P_IR do                          ▷ Adjust instruction/data address
4:          if ir_i.caddr is not aligned then
5:              stretchedSize += GetAlignment(ir_cur)      ▷ Keep the alignment correct
6:          end if
7:          ir_cur.caddr := AdjustInstAddress(ir_cur, stretchedSize)   ▷ Adjust each address
8:          for each ir_deref that dereferences ir_cur do
9:              ir_deref := AdjustDerefTargets(ir_deref)   ▷ Fix all call targets
10:         end for
11:     end for
12:     return P_IR
13: end function
```

also considers conditional execution. In particular, RevARM passes a condition field if any replaced instruction has it. Lastly, RevARM must take special care of every If-Then instruction (IT) in the instrumented program, which is a unique type of a conditional instruction in ARM. We describe how RevARM handles this instruction in Section 3.4.

Adjusting Binary Layout: In this step, RevARM adjusts the binary layout as described in Algorithm 3. Specifically, RevARM modifies each instruction/data address based on the size of the stretched code from Algorithm 2 and the necessary alignment. Stretching the input binary inevitably leads to violating the alignment which was originally assigned to the original binary. Therefore, we preserve the original alignment by adding or removing NOP instructions appropriately, similar to modern compilers (lines 3-11). Then, RevARM adjusts or replaces instructions/data to ensure the original control flows remain intact (lines 7-9).

3.4 If-Then Instruction

IT is a unique instruction of ARM that allows multiple instructions following the IT instruction to become conditional. For example, an LDR instruction that follows an IT instruction may or may not be executed depending on the condition specified in the IT instruction. We found that it is critical to correctly handle all of IT instructions as an incorrect handling of IT instruction may cause an erroneous control flow at run-time. However, handling IT is challenging due to its dynamic and complex nature, and, unfortunately, existing works do not address the problem. In this section, we describe how to handle IT instructions.

IT consists of an opcode and two fields: firstcond and mask. firstcond indicates which conditions will be enforced on the following instructions. mask determines how many instructions that follow the IT instruction will become conditional, namely an IT block. According to mask values, conditions of executed instructions can be reversed if these instructions belong to the IT block.

To handle IT, we set the IT condition flag in the IR representing each instruction and insert an IT at every fourth instruction in the IT block. However, there are two more complications that RevARM needs to consider. First, each IT can maximally cover only the following four instructions. Second, the first condition cannot be reversed. This situation is described in Fig. 5. In Steps 1 and 2 (the target of the two LDR instructions are unreachable after instruction insertion. In this case, one more IT must be inserted because the IT instruction cannot cover five instructions due to the length limitation. Therefore, SUB must belong to a new IT block at Step

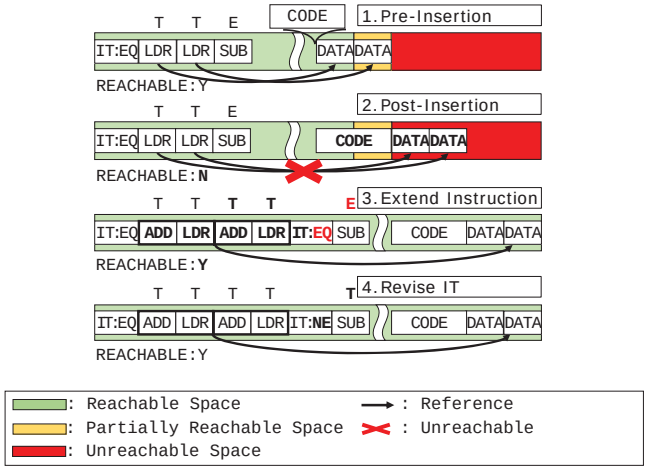


Figure 5: Handling IT instructions.

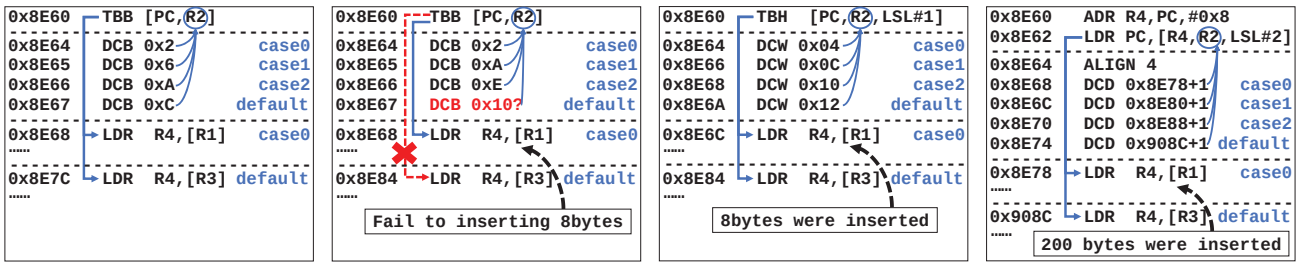
3. Then, firstcond should be reversed because the first condition must always be true and IT condition of SUB is a reversed value.

There are two cases which lead to the above case. First, instructions whose reference range is short belonging to an IT block. Even with small code insertions within their reference ranges, their reference ranges may be violated. Second, some instructions are inserted into an existing IT block. In this case, inserted instructions are enforced by the IT condition at the inserted address. However, we should consider two IT restrictions before insertions. The first restriction is that any instructions within IT block cannot set the condition flags except CMP, CMN, and TST. These behave differently from traditional branch instructions. Therefore, we must not replace IT with branch instructions such as B, BL, BLX, BX. In addition, some instructions such as B cannot be inserted into an IT block. In this case, RevARM rejects the insertion because such trials are invalid, resulting in unpredictable results when they are executed [6]. Finally, some branch instructions such as BX can be inserted with location limitation to the end of an IT block.

3.5 Branch Table Instructions

In x86, a switch statement in C/C++ is often implemented using conditional jump instructions (e.g., JE). In contrast, ARM provides special branch instructions for switch statements: Table Branch Byte (TBB) and Table Branch Halfword (TBH). These branch instructions dereference the jump table location first and then select the target address for the chosen case. Unfortunately, the branch instructions have limited reference distances and this limits the size of inserted instruction/data in ARM binary rewriting. To overcome this limitation, we handle the branch table instructions differently based on the reference distances that the instrumentation requires: short switch with TBB, medium switch with TBH, and long switch with LDR.

In Fig. 6a, we show an example of the short switch with TBB. The jump table is referenced by PC because the current PC value plus four which indicate the next PC value is the jump table location. Then, the value of R2 determines which case will be selected. TBB



(a) Before insertion. (b) Insertion failure due to the limited reachable distance. (c) 8-byte instruction insertion. (d) More than 200-byte instruction insertion.

Figure 6: An example of making unreachable switch statement reachable via instruction/data replacement.

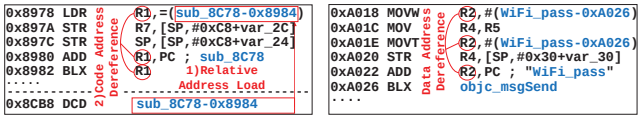
basically reaches only `0x22` relative address because only one byte is assigned to it. In other words, it can maximally reach `0x8E82`. Therefore, the `default` case located at `0x8E84` is unreachable if 8 bytes of instructions are inserted as described in Fig. 6b. In order to resolve this problem, RevARM replaces the `short switch` with `medium switch` consisting of `TBH` and two-byte relative address values as illustrated in Fig. 6c.

However, when more than 200 bytes of instructions are inserted, even `medium switch` cannot reach the `default` case since `TBH`'s maximum reachable distance is `0x202`. In this case, RevARM replaces `medium switch` with `long switch` as described in Fig. 6d. Unlike both `TBB` and `TBH`, `long switch` relies on `LDR` and absolute addresses which cover the whole address space. In order to jump to the right location, `ADR` resolves the jump table location by adding the next `PC` value and relative jump table address via `ADR`. Then, `LDR` dereferences the jump table address stored in `R4`, selects one of absolute addresses using `R2` and then updates the `PC` value with the selected jump address. We note that the last bit of each absolute address should be set if referenced instructions are thumb-mode instructions as described in §3.7. Also, each absolute address should be word aligned as illustrated in §3.8.

3.6 Direct Access to the Program Counter

The ARM architecture handles the `PC` register as a general register. This allows many instructions (e.g., `LDR`, `MOV`, `ADD` and etc) to directly read from and write to `PC`. Such access to `PC` involves dereferencing the target address before control transitions or referencing data, which occurs frequently in *position independent code* (PIC). In order to properly instrument the binary, RevARM dereferences the target address. Such dereference requires backward slicing [66] to figure out which instructions (e.g., LDR to load pointer values and `ADD` to modify a referenced address value) involve reading or writing `PC`. This is because most `PC` register accesses are performed via multi-staged address dereferences. After finding sliced instructions, RevARM dereferences the target address and modifies the correct address value on instructions or code pointer values to keep code and data dereference correct.

Fig. 7a presents an example of code address dereference. Dereference is done by executing two instructions: `LDR` and `ADD`. In this case, the target function (PC-relative address at `0x8CB8`) is loaded by `LDR`. Then, `ADD` adds the `PC` value to resolve the target address.



(a) Relative code address dereference. (b) Relative data address dereference.

Figure 7: Examples of the address dereference using the `PC`.

After that, `BLX` will be executed to jump to `0x8CB8`. In the case of the data address dereference, both `MOVW` and `MOVT` puts the relative address of data in `R2`. Then, `PC` is added to `R2` via `ADD`. This dereferenced data address is used as a parameter of `objc_msgSend` which is a typical iOS function call.

3.7 Run-time Instruction Mode Switching

32-bit ARM architecture provides two interchangeable instruction mode (ARM and Thumb). Unlike conventional architectures, ARM allows a program to switch the instruction mode even at run-time. This adds new challenges to ARM binary rewriting because many real-world ARM binaries are written in both ARM and Thumb instructions to reduce the binary sizes. In order to properly handle the instruction mode switching while keeping control flows correct, RevARM must abide by the branch rule of run-time instruction mode switching. There are three branch instructions capable of changing the instruction mode: `BLX`, `BXJ`, and `BX`. When they triggers branch operations, they have both target address and instruction mode bit. Such instruction mode bit is always located at the lowest bit of the address value. If the lowest value is set to 1 then the jump target executes in the Thumb mode. Otherwise, it will be executed in the ARM mode. Designed to be aware of the mode switching, RevARM separately handles direct and indirect branch instructions. In terms of the direct branch, RevARM simply set or clear the instruction mode bit according to the instruction mode of the referenced instruction. On the other hand, RevARM should perform backward slicing to dereference code pointer values. Then, it modifies pointer values in instructions or data. Further, when inserting instructions into the binary, RevARM makes sure that instrumented program uses the correct instruction mode before executing the inserted instructions. For example, RevARM ensures that the CPU switches to the Thumb mode whenever the inserted

Thumb instructions are executed while the instrumented program is running in the ARM mode.

3.8 Alignment

In this section, we described the three alignment cases which RevARM must correct: code, data, and reference address alignment.
Code Alignment: Since the instruction mode can be switched dynamically in ARM, all ARM mode functions (i.e., functions written in ARM instructions) and Thumb mode functions (i.e., functions written in Thumb instructions) must be aligned to a word (4 bytes) and a half-word (2 bytes), respectively. However, after the instrumentation, the stretched binary may not have the correct alignment, especially when the binary contains both ARM and Thumb mode functions. In such a case, RevARM re-align the code by inserting or removing `NOP` instructions before each function affected by the instrumentation, similar to how a compiler handles code alignment while generating the machine code.
Data Alignment: ARM allows instructions to load data in a code section in alignments of byte, half-word, word and double-word. From our observation, such data are aligned to word or double-word. Even if there are byte and half-word data, they will be word aligned by compilers. Therefore, RevARM aligns word data in 4 bytes and double word in 8 by inserting or removing `NOP`.
Reference Address Alignment: The Thumb mode supports half-word aligned addressing. However, there are several instructions that cannot access half-word aligned addresses. For example, `VLDR`, half-word-sized `LDR` and `ADR` — because their two lowest addressing bits are ignored. These mostly reference data in a text section due to their short reference ranges. Except data referenced by `TBB` and `TBH`, they are aligned to word or double-word. Therefore, it is not necessary to add additional logic to reference data in a code section. However, `ADR` sometimes needs to reference half-word aligned addresses. RevARM handles these by reordering independent instructions or replacing them with half-word-addressable instructions while such cases rarely happen.

3.9 Code Pointers

Data and metadata sections contain code pointers which must be adjusted with respect to the new address space layout modified by our instrumentation. However, it is challenging to identify code pointers in binaries because semantic information for those code pointers has been removed by the compiler. In order to solve this problem, RevARM adopts similar approaches proposed in previous work [62, 69] to recognizing pointer-like data. Specifically, RevARM checks whether pointers reference an instruction start address in a certain section. Note that we must also consider the branch rule of the Thumb instruction mode. As illustrated in §3.7, the lowest bit of the Thumb instruction address have to be set. Therefore, we can filter out code pointer-like data if it refers to Thumb instructions without setting the instruction mode bit.

3.10 Mach-O Metadata

Metadata handling is critical for Mach-O binaries since the validity of the metadata is checked by Apple's vetting process in order for binaries to appear in the App Store. We note that RevARM is designed to handle other popular binary formats (e.g., executable and linkable format (ELF)). Although the ELF format is well-documented [1] and supported by other binary rewriters, the official document which describes the Mach-O format only covers a few parts, leaving many other parts uncertain. Below we describe these uncertain parts, which can often lead to an incorrect binary transformation.

To handle the Mach-O binaries, RevARM should revise load commands and metadata sections. Load commands let the loader identify how to load an input binary in specified addresses. On the other hand, metadata sections contain symbols and other information that the linker uses.

For load commands, RevARM modifies file offsets, virtual addresses and sizes of the file and virtual address space for modified segments, sections and function start addresses to adjust the new binary layout. For metadata sections, RevARM also modifies identical types of information described for the load commands.

More importantly, variable-length addresses are also stored in the metadata sections. Such addresses are encoded with `uleb128` as the DWARF format stores them. For example, the dynamic loader's information consists of a set of nodes which respectively encoded symbols and other nodes absolute or relative addresses. Function start addresses are encoded in the same way. Hence, RevARM must increase the sizes of encoded addresses because of the stretched values of addresses between functions and symbols, which lead to larger encoded address values. As a result, RevARM stores the original data and reconstructs it after instrumentation.

4 EVALUATION

We tested RevARM on two commodity target systems: a mobile system and an embedded system. Our target mobile system is iOS version 10.0.2. For our target embedded system, we selected a 3DR IRIS+ [27], a popular quad-copter UAV based on the 3DR Pixhawk micro-controller [20]. Pixhawk contains an ARM Cortex-M4 processor with 256KB SRAM, and a 2MB flash memory, which is quite representative of a resource-constrained environment. The UAV is controlled by ArduPilot [12], a robotic vehicle controller program. ArduPilot supports many different types of unmanned vehicles (UxVs), such as copters and planes, as well as ground and underwater vehicles. For any UxV based on the Pixhawk micro-controller, ArduPilot runs the NuttX [29] real-time operating system (RTOS) along with many other operational components, including device drivers, libraries, and applications. In addition, ArduPilot relies on MAVLink [18] which is responsible for communicating with a ground control station (GCS) for the UAV. The GCS sends control commands abiding by the MAVLink protocol. We tested RevARM on such large-scale firmware to show that our approach is effective on complex ARM binary programs. In addition, we instrumented Mach-O binaries and stripped/non-stripped firmware binaries using RevARM to test the effectiveness of the binary rewriting on our iOS and UAV target devices, respectively.

In order to verify the correctness of RevARM's instrumented binaries, we conduct an experiment that inserts `NOP` instructions in between every instruction in the binaries. Then, we run the instrumented binaries with various workloads. Specifically, for binaries on ArduPilot, we run all flight missions provided by the vendor (127 missions) on the instrumented binaries. Moreover, we

Table 3: Instrumentation APIs of RevARM

Prototype	Description
`preproc`	Responsible for all preprocessing procedures
`instrument`	Main instrumentation function
`rearrange`	Control flow rearrangement to keep control flows intact
`flush`	Write an instrumented binary to a binary form

manually trigger all possible operations supported by ArduPilot (77 operations). For the instrumented iOS apps such as Twitter, Gmail, Amazon, and PerformanceTest, we exercise all functionalities displayed on the screen (e.g., click all buttons, explore all menu items).

RevARM is built upon an IDA-Pro 6.8 Plug-in which is responsible for disassembling the ARM binaries and identifying functions [16]. However, the RevARM technique is generic enough to be directly ported to other disassembly libraries (e.g., Capstone [24] or Radare2 [31]). In addition, we thoroughly evaluate the compatibility of RevARM on binaries generated by different compilers. Specifically, we analyzed binaries generated by popular compilers on three different platforms: GCC 4.9.3 for embedded systems (e.g., ArduPilot), GCC 4.8.2 on top of Linux, and Clang on iOS. Our result shows that RevARM is able to instrument binaries from all these compilers/platforms, without breaking any of the binaries. Finally, RevARM provides a set of APIs in order to facilitate instrumentation and functional extension on ARM binaries. We summarized these APIs in Table 3. In this section, we show that RevARM can be used in various security applications: fine-grained code diversification, vulnerable function patching, private API call prevention via SFI [56, 59, 67], and control system status monitoring. Then, we show run-time and space overhead.

4.1 Case Study I: Fine-grained Code Diversification

Code diversification is a defense technique that probabilistically limits the impact of an attack to a known target [48]. For example, randomizing the instructions in a binary program can prevent a wide range of code-reuse attacks which rely on prior knowledge of the layout of the victim binary code. Moreover, ROP attacks often chain together gadgets in binaries to construct attack payloads which are also sensitive to the layout of the binaries. Code diversification, which randomizes the layout of a binary, can significantly reduce the success rate of these attacks by randomizing the locations of code (and ROP gadgets) in the binary [46].

In this case study, we show the application of RevARM on fine-grained code diversification. Specifically, we used RevARM to perform instruction-level diversification on the ArduPilot firmware and four different iOS apps, including Gmail, Twitter, Amazon, and PerformanceTest Mobile benchmarking app [30]. Specifically, we inserted an increasing number of `NOP` instructions at arbitrary locations in the binaries and verified that all of the programs run correctly. To test the code diversification with the finest granularity, we inserted a `NOP` instruction before every instruction in the target binary. The details of the experiment is summarized in Table 4.

Table 4: Fine-grained code diversification with `NOP` insertion.

Program	Code size	Diversified code size	# of inserted `NOP`s	Space overhead
Twitter	7KB	12KB	2,448	1.39%
Gmail	2,367KB	3,973KB	786,909	1.78%
PerformanceTest	117KB	191KB	35,688	2.1%
Amazon	9,950KB	16,995KB	3,372,547	2.77%
Firmware	596KB	1,024KB	201,678	3.26%

Table 5: List of real-world bugs in ArduPilot. We patch these bugs in the stripped firmware binary using RevARM without using any symbol information.

Target	Bug ID	Module	Description
Memory	B1 [13]	ArduCopter	String null-terminated bug
Memory	B2 [8]	PX4 driver	Double free bug — Heap corruption
Memory	B3 [7]	PX4 driver	Potential integer overflow
Memory	B4 [14]	PX4 driver	Parsing bug — Buffer overflow
File system	B5 [5]	ArduCopter	Duplicated directory creation
File system	B6 [3]	NuttX	File system clustering

4.2 Case Study II: Vulnerable Function Patching

In this section, we show that RevARM can patch vulnerable functions in a stripped ArduPilot firmware binary. Vulnerable iOS apps are removed by Apple from App Store without an official announcement, but ArduPilot vulnerabilities can lead to property damage or even personal injury. Thus, we choose to demonstrate patching the ArduPilot firmware.

The target bugs we patched using RevARM are listed in Table 5. B1 is a null-terminated string bug in the `GCS_MAVLink` module in the ArduPilot firmware. The buggy function incorrectly uses the communication protocol between the UAV and the GCS. Specifically, it can send a wrong status report to the GCS and cause the GCS to send back an invalid command. B2 is a double free bug in a device driver, called `i2C`. A device pointer is not assigned null after heap deallocation by mistake. This leads to a heap corruption when later code tries to illegally deallocate the same heap memory again, which was already deallocated previously. The B3 case is an integer overflow bug in an I/O device driver. The driver computes a float actuator state value and an unregulated computation of the value causes an integer overflow, which leads to an unexpected flight control state. For B4, the bug is a stack buffer overflow in `i2C`. This target function calls the `sscanf` function to copy an input string to a fixed-size local buffer. Before our patch was deployed, there was no string length checking. Therefore, it can lead to a stack overflow which allows an attacker to hijack the control flow via manipulation of a return address. On the other hand, B5 is a file system bug in ArduPilot. The module `DataFlash_File` is responsible for logging UAV operation information on an SD card. This bug leads to SD card corruption by creating two directories for logging because this function did not check whether a logging directory was previously created. B6 is a second file system bug in the NuttX RTOS. Two vulnerable functions are responsible for file read and write respectively. However, they do not check file system cluster boundaries when reading and writing file system, which leads to file content corruption.

Patching the above cases require two steps, extract and embedding functions. In the extraction step, we used a patched binary and let RevARM extract the functions' instructions, data, and reference information. However, there are two types of data we should handle: RevARM first must store all data referred by any instruction that belongs to a function, and second, if this data is a pointer value, RevARM must also preserve its reference information. Then, RevARM stores extracted function information in an instrumentation specification file. In the embedding step, RevARM interprets the instrumentation specification file and replaces/inserts the code and data. RevARM basically replaces each instruction with a new instruction. Any remaining instructions are inserted between the end of the code and the start of the data. Then, RevARM rebuilds reference relationships based on the instrumentation specification. However, there are a few cases to handle when patching data: If new data belongs to a function, then RevARM inserts it into the code section. However, the original location of some data does not belong within a function. We can see this example in B4. B4 regulates stack overflow by changing the destination string of `sscanf`. Although we can replace such unpatched string with a new string, there is a possibility that the string is reused by other functions. Therefore, we insert such data at the end of its original section.

Other than the above cases, there are a substantial number of patched bugs in the ArduPilot repository. RevARM assumes that there is no open source repository for its target (as with many commercial UAVs), and we only leverage this information to identify which functions are vulnerable and their locations. This technique is also applicable to library patches since libraries for UAVs are statically compiled in their firmware. If other UAVs are patched and the patched vulnerabilities are in commonly used libraries, we also can utilize them for patches across other vulnerable systems.

4.3 Case Study III: Preventing Private API Abuses via SFI

Abusing private APIs has been shown to be an iOS specific attack vector [19, 63]. Unlike public APIs, which Apple allows developers to freely use, private APIs are undocumented functions that third-party developers are not allowed to directly use. This is because many of the private APIs are security-critical, which could allow an attacker to leak the user's private information and maliciously control the device.

Several works have attempted to prevent the abuse of private APIs, such as MoCFI [40] and XiOS [38]. However, these have technical limitations. For example, MoCFI [40] requires jailbreaking the target iOS platform for instrumentation. Similarly, XiOS [38] provides limited protection on private APIs [42] by preventing attackers from inferring the addresses of the private APIs.

To resolve the above issues, RevARM applies SFI to prevent the code section (`__text`) from accessing private APIs and symbol address modifications. To enable this, RevARM modifies the code sections to block write access to symbol pointers and indirect branch to library codes directly. We note that RevARM allows the stub code section to access both sections to preserve the correctness of the program because the stub code section is responsible for accessing symbol pointers and library functions.

In order to enforce SFI, RevARM also needs to modify the binary layout and inserts SFI instructions. To modify a binary layout, RevARM modifies sizes of segments and sections in the input binary. For SFI instruction insertions, we insert SFI instructions (described in NaCl for ARM [56]). However, SFI does not work well simply with SFI instruction insertions. Control flows must execute SFI instructions before potentially vulnerable instructions. Consequently, it requires aligning each SFI instruction and function address, since some attacks can directly jump to target instructions without executing SFI instructions. In order to adjust this alignment, we inserted padding instructions to forcibly fit the alignment.

Lastly, in order to evaluate the efficiency of our added SFI protection, we implemented two attack cases. Private API calls using inferred and actual API addresses. The first attack we implemented was described earlier in XiOS [38]: The private API addresses can be inferred by a statically loaded symbol, (`dyld_stub_binder`) before app execution. Attackers can abuse this symbol with some offset to call private APIs. We implemented this attack case by calling a dynamic library loading function, which is a private API and can load arbitrary libraries including private libraries. However, our SFI enforcement does not allow to access such symbol section directly from code sections. Therefore, our SFI enforced app is protected against attacks which try to infer private API addresses. The second attack is calling private APIs using direct addresses via signature-based scanning to find target private APIs [42]. XiOS cannot prevent such attack because XiOS only hides symbol values — not preventing the call to a private API. On the other hand, our added app protections do prevent such attacks because SFI prevents malicious accesses to private APIs.

4.4 Case Study IV: Control System Status Monitoring

Control systems heavily depend on parameters, commands of the ground control station (GCS), sensor inputs, control status and control model. Out of such factors, such a control system determines a new system status based on dynamic variables, including sensor input, command, current control status and control model. In order to prevent and observe unexpected behavior within such a system, it is necessary to observe such dynamic variables. As a case study of monitoring these variables, we insert our monitoring instrumentation into a UAV's firmware.

An adversary can cause an anomalous system status by changing dynamic control variables into exceptional values, which leads to system disruption or crash. Such system disruption has been achieved by sensor attacks via intentional ultrasonic signals [58] and control hijacks [9] via wireless communication modules. In order to prevent such attacks, we inserted control variable status monitoring as the first step of anomaly system status regulation. Our instrumentation lets UAVs trace status variables and sends these traces to a remote UAV managing system.

We summarize our target functions in Table 6. All of them are related directly to the above attack scenarios. For example, in the 'intentional ultrasonic signals' case, our target UAV utilizes multiple sensors some of which are vulnerable to intentional ultrasonic signals [58]. Furthermore, a communication protocol vulnerability can be abused by malicious attackers. One demonstrative example

Table 6: List of ArduPilot functions instrumented for the run-time status monitoring of the flight controller.

Class	Function	Purpose of instrumentation
AC_PID	get_p, get_i, get_d	Get PID control values
AP_Mount_Servo	move_servo	Get the desired motor speed
AC_AttitudeControl	rate_bf_roll_pitch_yaw, etc	Get 3-axis attitude for balancing
AC_PosControl	set_pos_target	Get the target position of the UAV
AP_InertialNav_NavEKF	get_altitude, get_latitude, etc	Get estimated geometric information
AP_GPS	velocity, location, etc	Get estimated velocity and location
AP_InertialSensor	get_gyro, get_accel	Get estimated inertial sensor values
None (OS C Function)	nsh_parse	Get shell commands
GCS_MAVLINK	handleMessage	Handle MAVLink messages from GCS

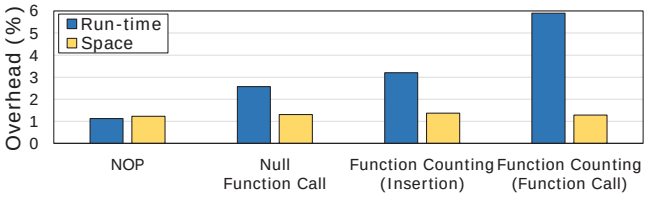


Figure 8: Run-time and space overhead with CoreMark.

is the SynerComm A-Team's work [9] which shows how to hijack a UAV using MAVLink. This communication protocol is widely used in UAVs including ours. With our status variable monitoring, security investigators are able to determine which functions are vulnerable or involved in an anomalous system status. Then, they can not only test but also limit anomaly values by adding robust control functions based on the status variable monitoring.

4.5 Run-time and Space Overhead

In this section, we evaluate any run-time and space overhead induced by RevARM on our two target platforms: iOS and UAV. In order to measure the instrumented ArduPilot firmware overhead, we ported the CoreMark [26] benchmark to the firmware as an application run on NuttX. Then, we performed four general instrumentation cases: NOP, null function call, insertion-based function counting and function call-based function counting. In the `NOP` case, we inserted a `NOP` instruction to the entry of every function. A null function call involves executing two branch instructions. There are two function counting scenarios in our experiments: First is function counting on the function start addresses directly. The other is performed by counting at the call site of each function.

Fig. 8 presents both run-time and space overhead measurements for these four cases. In terms of run-time overhead, NOP, null function call, insertion-based function counting and function call-based function counting respectively show 1.13%, 2.57%, 3.2% and 5.9% run-time overhead and 1.23%, 1.31%, 1.37% and 1.28% space overhead. Among the above tests, the two function counting experiments best show the trade-off between run-time and space overhead. Insertion-based function counting shows smaller run-time overhead then function call-based function counting because function call-based function counting is similar to trampoline-based instrumentation. Such approach introduces additional expensive control transitions unlike our insertion-based approach.

To measure instrumented iOS app run-time overhead, we used the PerformanceTest Mobile benchmarking app [30] on an iPhone 5s. Unlike the above experiments, here we applied our SFI to this app

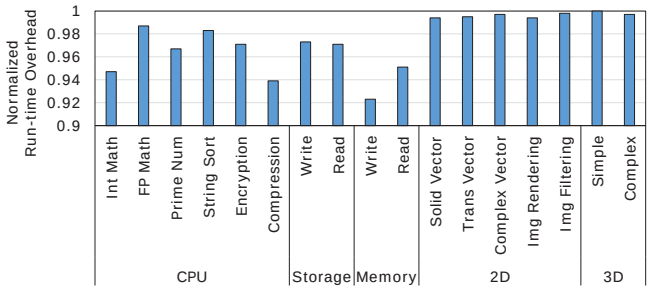


Figure 9: Run-time overhead of RevARM-based SFI on iOS.

and ran it to measure the overhead. The PerformanceTest Mobile performs benchmarks into five different areas: CPU, Disk, Memory, 2D and 3D graphics consisting of 17 sub-experiments. Then, it shows scores for each experiment area. Using this benchmark, we tested SFI on this app to measure their overhead. This result is shown in Fig. 9. CPU, disk, storage, and memory showed smaller run-time overhead: 3.4%, 2.8%, and 6.3%, respectively. The highest overhead was the memory test due to more frequent memory accesses. On the other hand, 2D and 3D show negligible run-time overhead (< 0.5%) since their performance depends largely on graphic libraries. We also experimented with enforcing SFI on real applications (such as Amazon, Gmail, Twitter) and found an increase in 31.3% on average. Make sure that both run-time and space SFI overhead includes both SFI codes and our tool's overhead in this case. We show our tool's own overhead in Fig. 8.

5 RELATED WORK

Static Binary Instrumentation: There are many existing static instrumentation techniques developed for x86. A number of techniques leverage symbolic information to accurately locate and rewrite binary code [41, 44, 54, 55]. Detour-based approaches [45, 54] are incapable of instrumenting instructions at arbitrary locations but the inserted code for trampoline and control transfers incur additional run-time and spatial overhead. Unlike detour-based approaches, patch-based approaches [49, 64, 68] duplicate the original code and patching the duplicated code. However, such code duplication causes large space overhead. Dyninst [39] utilizes both approaches depending on the specific instrumentation task.

More similar to RevARM, insertion-based approaches directly insert into or replace instructions and data in a target binary without adding a control transition. BISTRO [43] inserts code into a stretched binary. Also, it supports function patching as our work does. REINS [65] is a machine-verifiable binary rewriting technique that protects a target program from unsafe branch targets by inserting code into the binary. Uroboros [62] is based on an advanced disassembling technique [61] to convert a binary into their own internal representations and perform instrumentation on those.

All of the above techniques are designed for the x86 architecture. Due to the unique challenges in handling ARM-specific instructions (§3), their approaches are limited to supporting x86 only and can hardly be applied to ARM binary instrumentation. Dyninst is going to support for rewriting only 64-bit ARM binaries as an experimental feature, but not 32-bit, due to the specific challenges

related to rewriting 32-bit ARM binaries. Rewriting 32-bit ARM binaries is much more challenging than 64-bit ARM binaries since it requires handling: (1) both 16-bit and 32-bit instructions, (2) much more diverse branch instructions, (3) instructions with a limited address dereference distance, and (4) dynamically switching ARM and Thumb instruction modes. RevARM enables 32-bit ARM binary rewriting by addressing these fundamental challenges that existing works (including Dyninst) do not solve. Furthermore, RevARM is capable of rewriting Mach-O binaries while all existing approaches do not support these.

LLVM IR Instrumentation: SecondWrite [34, 52] leverages Low Level Virtual Machine (LLVM) lifting to convert a binary into LLVM IR for instrumentation. Yet, this LLVM lifting feature was officially removed since LLVM 3.1 because it is unable to lift any non-trivial program binaries into LLVM IR [2]. Consequently, SecondWrite is not mature enough to rewrite full-scale stripped applications [61, 65]. Moreover, SecondWrite relies on patch-based instrumentation which leads to larger space overhead than RevARM. Finally, there exists no experimental result to show that this approach can rewrite ARM-based binaries. On the contrary, the experimental results in §4 show that RevARM can instrument large-scale ARM binaries and, by applying our technique, can improve the security of real-world mobile and embedded systems.

Dynamic Binary Instrumentation: Dynamic binary instrumentation techniques, such as PIN [50], DynamicRio [37], Valgrind [51], Detours [45] and QEMU [36], instrument binaries loaded in the memory at run-time. However, there are two reasons why we chose static instrumentation when designing RevARM. First, dynamic instrumentation techniques incur large run-time and space overhead which is a critical problem on embedded and mobile systems which not have high processing power and large memory. In addition, these tools only support a small number of commodity operating systems since their techniques are OS-dependent. In contrast, RevARM is designed to provide platform-agnostic binary rewriting capabilities.

6 LIMITATIONS

We point out that, despite its new capability, RevARM has a number of limitations — some of which are intrinsic to the current-generation binary rewriting methodology and call for innovative advances through future research.

Dynamically Generated or Obfuscated Code: Like all other static binary instrumentation techniques [43, 62, 64, 68, 69], dynamically generated code (e.g., self-modifying code) cannot be targeted by RevARM since the code can only be seen while the program is running. In addition, the correct disassembly of obfuscated binaries is an orthogonal problem to all binary rewriting techniques, including RevARM. Existing obfuscation-resilient disassembly techniques [47, 57] can be used to complement RevARM's rewriting capabilities. Further, we observed that such obfuscation is not common in our target binaries because obfuscated iOS apps may be rejected by App Review and the computing power of embedded systems is limited [57].

Limitations Inherited from Disassembly: Since RevARM is built upon IDA Pro for the disassembly of ARM binaries, it inherits the current limitations of the disassembler. It is well-known that disassemblers are imperfect [35, 60, 61], and this has remained a restrictive problem for binary rewriting. The main reason for this is that compilation removes semantic information (e.g., pointer types) [65] from the resulting binary program. Inheriting this absence of information can lead RevARM to produce incorrect instrumentation for target binaries. Possible effects of this problem include: (1) misidentification of pointers and virtual tables in C++ programs, and (2) incorrect disassembly of data as code and code as data. If the disassembler fails to identify pointers and virtual tables correctly, the program instrumented by RevARM may take ill-formed execution paths or access data at incorrect locations. In addition, erroneously disassembling data as code or vice versa may also lead to incorrect control or data flow during the execution of the instrumented program. While this remains an open research challenge, there have been some advances in improving the accuracy of disassemblers, which may be leveraged to mitigate these negative effects in the future. BinCFI [69] improves disassembler accuracy by combining two existing disassembly algorithms. Marx [53] restores C++ class hierarchy information with high accuracy from stripped binaries, which enhance the detection of virtual tables and pointers.

Pointer Arithmetic: While it is theoretically possible for a binary program to have pointer arithmetics on indirect branch and data reference targets, RevARM does not handle complex arithmetic operations, based on the observation that a vast majority of modern binaries do not include such pointer arithmetics in practice [69]. In our experiments, we did not find any pointer arithmetic operation except one simple bitwise arithmetic that flips a bit to switch the instruction mode between the ARM and Thumb modes. We handle this case by setting the mode bit based on the current instruction mode.

7 CONCLUSION

Despite the popularity of ARM-based systems, ARM binary instrumentation techniques are still immature due to the challenges associated with accurately rewriting ARM binaries without source code across a variety of platforms. To the address the challenges, we proposed RevARM, a platform-agnostic ARM binary rewriting technique capable of instrumenting binaries without symbolic/semantic information. Due to its insertion-based instrumentation technique, RevARM is able to support powerful security applications with a fine-grained ARM binary rewriting capability while introducing very low run-time and space overhead. Furthermore, RevARM addresses a number of unique challenges in ARM binary rewriting, which previous work did not solve. Our experimental results demonstrated the usefulness and practicality of RevARM in various security applications on real-world ARM-based devices.

8 ACKNOWLEDGMENT

We thank our shepherd, Maverick Woo and the anonymous reviewers for their valuable comments and suggestions. This work was supported, in part, by ONR grants N00014-17-1-2045, N00014-17-1-2513, and N00014-17-1-2947. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

REFERENCES

[1] 1995. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2.* http://refspecs.linuxbase.org/elf/elf.pdf.

[2] 2012. *LLVM 3.1 release note.* http://releases.llvm.org/3.1/docs/ReleaseNotes.html.

[3] 2013. *Ardupilot File System Corruption Bug - DataFlash: don't try to create a directory that exists.* https://github.com/ArduPilot/ardupilot/commit/4ce2555a6563939f95991621facc7ff4b9f27d1d.

[4] 2013. *Hacking Drones - Overview of the Main Threats.* http://resources.infosecinstitute.com/hacking-drones-overview-of-the-main-threats.

[5] 2014. *Nuttx File System Corruption Bug - FAT: move cluster expansion checks to start of IO loops.* https://github.com/PX4/NuttX/commit/ed45e813aff84f5646ea7ad1d7ab50f597bdebb9.

[6] 2014. *Processors | ARMv7-M.* https://silver.arm.com/download/ARM_and_AMBA_Architecture/AR580-DA-70000-r0p0-05rel0/DDI0403E_B_armv7m_arm.pdf.

[7] 2015. *Firmware Control Output Handling Bug - IO driver: Ensure comms protocol cannot get into integer overflow.* https://github.com/PX4/Firmware/commit/e09f5d2871f0c23cf8eb8154a2fa8831d9b96062.

[8] 2015. *Firmware Memory Vulnerability - i2c: prevent double free of _dev pointer.* https://github.com/PX4/Firmware/commit/1b8a830a38caf393cb308ad206d3c23329d58a48.

[9] 2015. *Hijacking drones with a MAVLink exploit on DIY Drones.* http://diydrones.com/profiles/blogs/hijacking-quadcopters-with-a-mavlink-exploit.

[10] 2015. *S.F. Express Launches First Drone Delivery Service in China.* http://english.cri.cn/12394/2015/03/24/1261s871432.htm.

[11] 2015. *Xaircraft - Drone Vendor.* http://www.xaircraft.cn.

[12] 2016. *ArduPilot Autopilot Suite.* http://ardupilot.org/ardupilot/index.html.

[13] 2016. *Ardupilot Memory Vulnerability - GCS_MAVLink: fixed null termination bug.* https://github.com/ArduPilot/ardupilot/commit/197e72acc0efa094c48070b6409d605b00b36ba6.

[14] 2016. *Firmware Memory Vulnerability - Prevents the possibility of buffer overflow in mixer parsing.* https://github.com/PX4/Firmware/commit/db44129ec099a05debf9187da2fd09035c9a67d7.

[15] 2016. *Hackers take over security camera; live stream girls' bedroom on Internet.* https://www.hackread.com/hackers-live-stream-hacked-security-camera.

[16] 2016. *Hex-Rays, IDA Pro disassembler.* http://www.hex-rays.com/products/ida.

[17] 2016. *How Hackers Violate Privacy and Security of the Smart Home.* http://resources.infosecinstitute.com/how-hackers-violate-privacy-and-security-of-the-smart-home.

[18] 2016. *MAVLink - Micro Air Vehicle Communication Protocol.* http://qgroundcontrol.org/mavlink/start.

[19] 2016. *Technical Analysis of Pegasus Spyware.* https://info.lookout.com/rs/051-ESQ-475/images/lookout-pegasus-technical-analysis.pdf.

[20] 2017. 3DR Pixhawk | 3DR - Drone & UAV Technology. (2017). http://3dr.com/support/articles/207358096/3dr_pixhawk.

[21] 2017. *Amazon Echo - Smart Speaker.* https://www.amazon.com/Amazon-Echo-Bluetooth-Speaker-with-WiFi-Alexa/dp/B00X4WHP5E.

[22] 2017. *BAT - Binary Analysis Tool.* http://www.binaryanalysis.org/en/home.

[23] 2017. *Binwalk - Firmware Analysis Tool.* http://binwalk.org.

[24] 2017. *Capstone.* http://www.capstone-engine.org.

[25] 2017. *Clutch - Fast iOS executable dumper.* https://github.com/KJCracks/Clutch.

[26] 2017. *CoreMark - Industry-Standard Benchmarks for Embedded Systems.* http://www.eembc.org/coremark.

[27] 2017. *IRIS+ | 3DR - Drone & UAV Technology.* http://3dr.com/support/articles/207358106/iris.

[28] 2017. *McAfee Labs 2017 Threats Predictions, Nov. 2016.* https://www.mcafee.com/kr/resources/reports/rp-threats-predictions-2017.pdf.

[29] 2017. *NuttX Real-Time Operating System.* http://nuttx.org.

[30] 2017. *PerformanceTest Mobile.* http://www.passmark.com/products/pt_mobile.htm.

[31] 2017. *radare.* https://www.radare.org.

[32] 2017. *Roomba - Robot Vacuum Cleaner.* http://www.irobot.com/For-the-Home/Vacuuming/Roomba.aspx.

[33] 2017. *Symantec Internet Security Threat Report 2017, Volume 22.* https://www.symantec.com/content/dam/symantec/docs/reports/istr-22-2017-en.pdf.

[34] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A compiler-level intermediate representation based binary analysis and rewriting system. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13).*

[35] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security '16).*

[36] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track (ATC '05).*

[37] Derek L Bruening. 2004. *Efficient, transparent, and comprehensive runtime code manipulation.* Ph.D. Dissertation. MIT.

[38] Mihai Bucicoiu, Lucas Davi, Razvan Deaconescu, and Ahmad-Reza Sadeghi. 2015. XiOS: Extended application sandboxing on iOS. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (ASIACCS '15).*

[39] Bryan Buck and Jeffrey K Hollingsworth. 2000. An API for runtime code patching. *International Journal of High Performance Computing Applications* (2000).

[40] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. 2012. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security (NDSS '12).*

[41] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. 2005. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems* (2005).

[42] Zhui Deng, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2015. iris: Vetting private api abuse in ios applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15).*

[43] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. 2013. BISTRO: Binary Component Extraction and Embedding for Software Security Applications. In *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS '13).*

[44] Andrew Edwards, Hoi Vo, and Amitabh Srivastava. 2001. Vulcan binary transformation in a distributed environment. (2001).

[45] Galen Hunt and Doug Brubacher. 1999. Detours: Binary Interception of Win32 Functions. In *Proceedings of the 3rd USENIX Windows NT Symposium.*

[46] Todd Jackson, Andrei Homescu, Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Diversifying the software stack using randomized NOP insertion. In *Moving Target Defense II.* Springer.

[47] M Ammar Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. 2016. Speculative disassembly of binary code. In *Proceedings of the 2016 IEEE International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES '16).*

[48] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P '14).*

[49] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. 2010. Pebil: Efficient static binary instrumentation for linux. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS '10).*

[50] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05).*

[51] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07).*

[52] Pádraig O'Sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D Keromytis. 2011. Retrofitting security in cots software with binary rewriting. In *Proceedings of the IFIP International Information Security Conference (SEC '11).*

[53] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. 2017. MARX: Uncovering class hierarchies in C++ programs. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS '17).*

[54] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. 1997. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop.*

[55] Benjamin Schwarz, Saumya Debray, Gregory Andrews, and Matthew Legendre. 2001. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proceedings of the Workshop on Binary Translation (WBT '01).*

[56] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the 19th USENIX Security Symposium (USENIX Security '10).*

[57] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2015. Firmalice-Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the 22nd Annual Symposium on Network and Distributed System Security (NDSS '15).*

[58] Yunmok Son, Hocheol Shin, Dongkwan Kim, Youngseok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, and Yongdae Kim. 2015. Rocking Drones with Intentional Sound Noise on Gyroscopic Sensors. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15).*

[59] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1994. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles (SOSP '93).*

[60] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS '17).*

[61] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable disassembling. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*.
[62] Shuai Wang, Pei Wang, and Dinghao Wu. 2016. UROBOROS: Instrumenting Stripped Binaries with Static Reassembling. In *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER '16)*.
[63] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. 2013. Jekyll on ios: When benign apps become evil. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security '13)*.
[64] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. 2012. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security (CCS '12)*.
[65] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. 2012. Securing untrusted code via compiler-agnostic binary rewriting. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*.
[66] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software engineering (ICSE '81)*.
[67] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (IEEE S&P '09)*.
[68] Mingwei Zhang, Rui Qiao, Niranjan Hasabnis, and R Sekar. 2014. A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual Execution Environments (VEE '14)*.
[69] Mingwei Zhang and R Sekar. 2013. Control flow integrity for COTS binaries. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security '13)*.