Received July 23, 2018, accepted August 24, 2018, date of publication September 3, 2018, date of current version September 28, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2868411

REPICA: Rewriting Position Independent Code of ARM

DONGSOO HA[1], (Student Member, IEEE), WENHUI JIN[1], (Student Member, IEEE), AND HEEKUCK OH[1], (Member, IEEE)
[1]Department of Computer Science and Engineering, Hanyang University, Ansan 15588, South Korea

Corresponding author: Heekuck Oh (hkoh@hanyang.ac.kr)

This work was supported in part by the Ministry of Science, ICT and Future Planning, Korea, through the Information Technology Research Center Support Program, Institute for Information and Communications Technology Promotion, under Grant IITP-2018-2014-0-00636 and in part by the National Research Foundation of Korea through the Korean Government, Ministry of Education, Science and Technology, under Grant NRF-2015R1D1A1A09058200.

ABSTRACT Binary rewriting techniques are widely used in program vulnerability fixing, obfuscation, security-oriented transforming, and other purposes, such as binary profiling and optimization. Over the past decade, most binary instrumentation techniques have been studied on ×86 architecture, specifically focusing on the challenges of instrumenting non-PIC. In contrast, ARM architecture has received little attention, and statically instrumenting PIC has not been studied in depth. In ARM, owing to its fixed-length instructions, addresses are frequently computed via multiple stages, making it difficult to handle all relative addresses, especially the relative address of *base-plus-offset* and *base-plus-index addressing*. In this paper, we present `REPICA`, a static binary instrumentation technique which can rewrite ARM binaries compiled in a position-independent fashion. `REPICA` can instrument at anywhere without symbolic information. With the aim of identifying and processing relative-addresses accurately, we designed a value-set analysis specialized for PIC of which the domain is in symbolic format. We also identified a new challenge for situations all relative addresses cannot be corrected in an optimized way and solved this problem efficiently by the stepwise correction of each relative address. We implemented a prototype of `REPICA` and experimented with approximately 1200 COTS binaries and SPECint2006 benchmarks. The experiment showed that all binaries rewritten by `REPICA` maintain relative addresses correctly with negligible execution and space overhead. Finally, we exhibit the effectiveness of `REPICA` by using it to implement a shadow stack.

INDEX TERMS Binary instrumentation, binary rewriting, PIC, position independent code, reassembly.

I. INTRODUCTION

In a few decades, binary instrumentation technique has become an important key technique in the security area. It may be necessary to fix a bug in a program of which the source code cannot be accessed, or an untrusted program may have to be run. In addition to this, even if the source code is available, there may have the type of work that cannot be carried out via only generic programming languages or compilers. In this situation, binary instrumentation technique can be used to solve these problems. Specifically, control-flow integrity (CFI) hardens binaries by inserting a verification routine at each indirect control flow transfer point [1]–[5], which can protect against most existing code and injected attacks such as jump-oriented programming (JOP) and return-oriented programming (ROP) attacks. Moreover, this technique can also be used for a variety of security purposes, including software-based fault isolation (SFI) [6]–[9], obfuscation [10]–[12], shadow stack [13]–[17], and bug fix [18]. Because of the ability to modify the program directly at low level, binary instrumentation technique is crucial.

To date, studies of binary instrumentation technique has been focusing on ×86 architecture and non-PIC [18]. Recently, the increase in the use of smart devices has caused ARM architecture to grow in importance. Due to the characteristics of smart device, it requires high security on limited resources. PIC can satisfy the requirements for both efficient memory management (e.g., physical memory sharing) and high security (e.g., address space layout randomization [19]), whereas non-PIC cannot satisfy both of these requirements together. For this reason, the significance of PIC[1] has also been increasing, and most files on smart devices are PIC despite the good performance of non-PIC. As a representative

[1]In this paper, PIC refers to code compiled in a position-independent fashion rather than a compiled option.

example, both Android (above 5.0) and iOS (above 4.3) have enhanced their policies to enforce the use of PIC in third-party applications [20], [21]. However, the many severe security vulnerabilities still encountered on PIC, and only the security feature of PIC cannot prevent it from all hacking attacks [22]–[24]. To cope with this situation, the binary instrumentation technique can be used as implementing binary protecting techniques such as CFI, SFI, and shadow stack. Unfortunately, it is difficult to apply existing techniques to the PIC of ARM architecture.

The main difficulty is from multiple address computation in PIC of ARM. All addresses have to be computed owing to characteristic of PIC, and ARM instruction is fixed-length. Thus, the relative-address may not be contained in a single instruction, thereby the computation for obtaining an address consist of multiple stages generally. In addition to this, in ARM, PC register is general-purpose register, thereby many instructions can read from and write to the PC register for the address computation. For these reasons, there are various address computation patterns. Existing ×86 techniques do not consider these difficulties [4], [25], [26]. Previous studies on ARM were only able to process relative-addresses that are directly computed with the PC register [18].

In addition to this problem, the PIC of ARM presents a new problem in that the correction process may not terminate. In some ARM instructions, the operands are scalable; hence, fewer instructions can express a larger relative-address. Because of this feature, in the process of correcting relative-addresses, when each relative-address is always corrected by optimized instructions, these corrections may occur infinitely in a cyclic manner. We refer to this problem as the *cyclic correction problem*. To the best of our knowledge, research attempting to address this problem has not yet been reported. This means that a technique capable of rewriting the PIC of the ARM architecture efficiently and correctly does not exist.

In this paper, we present `REPICA`, a static binary instrumentation technique that can flexibly modify text segments of a binary compiled in a position-independent fashion. `REPICA` has two methods to solve challenges for handling PIC. The first method is a value-set analysis specialized designed for PIC, which can find and correct the relative-address of the address computation in multiple stages. The second method is a correction method which adjust each relative-address stepwisely, which can solve the *cyclic correction problem* efficiently. In particular, we are the first to address the *cyclic correction problem*. We implemented a prototype of `REPICA` and tested it on approximately 1,200 binaries, including the entire PIC or PIE files of Android 8.1 (both the 32-bit and 64-bit versions). Our experiments showed that all binaries rewritten from the output produced by `REPICA` maintain relative-addresses correctly with negligible execution and space overhead. We summarize our contributions as follows:

- To the best of our knowledge, we are the first to propose a static technique with the ability to systematically rewrite ARM binaries that are compiled in the position-independent fashion. This technique allows the text segments of the target binary to be modified flexibly.
- We identified a new challenge (referred to as the *cyclic correction problem*), i.e., that relative-addresses cannot always be corrected optimally. In this paper, we present a detailed approach to overcome the *cyclic correction problem*.
- We implemented a prototype of `REPICA` and tested it on approximately 1,200 binaries, including the entire PIC or PIE files of Android 8.1 (both the 32-bit and 64-bit versions). Our experimental results confirmed that our tool is able to correctly and efficiently rewrite the binaries.

The remainder of this paper is organized as follows: Section II contains the background to the problem, Section III presents the challenges that need to be addressed, Section IV provides our approach to overcome these challenges, Section V describes our design and the implementation of `REPICA`, Section VI presents the experimental results and case study, Section VII discusses the limitations of `REPICA`, Section VIII describes related work, and Section IX summarizes our conclusions.

II. BACKGROUND

In this section, we first define some terms for convenience of explanation as it is not easy to describe our technique in existing terms. Next, we describe the binary instrumentation technique, PIC, and ARM architecture. Finally, we discuss the limitation of existing approach on PIC.

A. TERM DEFINITION

The target instructions we are interested in are those that use the relative-address via an immediate value or an index register, and we divide them into two classes depending on whether their base register is the PC register. Among the target instructions, an instruction of which the base register is the PC register is termed a *PC-relative instruction*, whereas an instruction of which the base register is not the PC register is termed a *base-relative instruction*. Examples of target instructions include general-purpose arithmetic instructions such as `ADD` and `SUB`, data transfer instructions such as `LDR` and `STR`, and branch instructions such as `B`, `BL`, and `CBZ`. Note that instructions involving *register indirect addressing*, such as `LDR rm, [rn]`, are not target instructions because we only consider instructions that use a nonzero relative-address.

The addresses of interest are the `PC` value, the address computed through arithmetic instructions, and the *effective-address* of the target instruction. The `PC` value itself or the address obtained through arithmetic operation between the `PC` value and the relative-address is termed the *PC-based address*, and the others is termed the *non-PC-based address*. In addition, we also divide the all addresses into two types according to the area within which it falls. An address that falls within the instruction area is termed a *code-address*, whereas an address in another area, such as the literal pool of the text segment and the data segment is termed a *data-address*. As a special case, when considering the relative-address range, we regard the PC value itself as the address of the instruction regardless of the prefetch value.

B. BINARY INSTRUMENTATION TECHNIQUES

Depending on the placement of the instrumentation code, the binary instrumentation technique can be classified into *trampoline-based* and *reassembly-based* approaches. In the *trampoline-based* approach, a new code area, named *trampoline*, is typically created at the lower end of the program, and the instrumentation code is placed in that area. More specifically, *detour-based* techniques such as Dyninst [27], Etch [28], and Detour [29] replace the original instruction with a jump instruction that passes control to the *trampoline* area in which the corresponding instrumentation code run. When the instrumentation has completed its task in the *trampoline* area, program control returns to the original area. Slightly differently, *patch-based* techniques such as STIR [30], PEBIL [31], and Multiverse [32] keep the original almost intact and create a duplicated code area, which is patched for instrumentation. The program is executed on the duplicated area, where program control is passed to the *trampoline* area when instrumentation is needed. This approach generates high runtime and space overhead due to the frequent jumps to the *trampoline* area and the duplicated code.

The *reassembly-based* approach identifies absolute-addresses in the literal pool[2] and data segments by performing work known as *symbolization*. The first related study led to the development of Uroboros [25], which uses a heuristic approach under strong assumptions to recognize pointer-like data. A follow-up study proposed Ramblr [26], which uses localized data flow analysis and value-set analysis to remove the heuristic introduced by Uroboros as much as possible. In this technique, because it identifies all absolute-addresses in a program, in contrast to the *trampoline-based* approach, this approach can insert the instrumentation code at any point in the original code. In other words, this approach does not need to create a separate *trampoline* area and can perform more flexible instrumentation with negligible execution and space overhead.

Our REPICA is a static binary instrumentation technique that targets PIC composed of an ARM instruction set and does not require any symbolic information. REPICA can perform instrumentation at levels similar to those of the *reassembly-based* approach. Specifically, it can insert instrumentation code before or after any instruction in the text segments, and requires neither a *trampoline* area nor duplication of the original. The output generated by REPICA is also a standalone binary which does not require a special environment.

[2]The literal pool is an area of memory in the text segment, and is used to store constants such as plain numerical constants, strings, and addresses of variables.

C. POSITION INDEPENDENT CODE AND ARM ARCHITECTURE

PIC has a characteristic that can be loaded on any location in the virtual memory. Thus, unlike non-PIC, binaries compiled in a position-independent fashion do not contain absolute-addresses directly. All addresses used in the code are obtained by computing between the PC value and a relative-address except for addresses obtained by the stack register or dynamic allocation, and the computed address can be recomputed with other relative-addresses. Although the data segments can contain absolute-addresses at runtime, relocation information is required as well. This feature of PIC ensures that the distinction between addresses and constants is clear.

The ×86 instruction set has a variable length, whereas the length of the ARM instruction set is fixed. Depending on the size of the relative-address, a single instruction in ARM architecture may not fully include a relative-address. Therefore, in ARM architecture, address computation is mostly multi-staged. In addition, because of the general-purpose nature of the PC register, relative-addresses can be used in various instructions. These two characteristics enable the use of various address computational patterns in the PIC of ARM. Considering the PIC patterns generated by most modern compilers, a single pattern is generated in ×86 architecture, whereas various address computational patterns are generated in ARM architecture. The following are examples of the address computation with relative-address 0×1281180 in each of the architectures:

```
<x86 architecture>
CALL __i686.get_pc_thunk.bx          __i686.get_pc_thunk.bx:
ADD  EBX,0x1281180                   MOV  EBX,[ESP]
                                     RET

<ARM architecture>
LDR R2,[PC,#0x100]    MOVW R2,#0x1180    ADR R2,#0x1280000
ADD R2,PC,R2          MOVT R2,#0x0128    ADD R2,#0x1180
...(0x01281180)...    ADD  R2, PC, R2
```

D. LIMITATION OF EXISTING APPROACH ON PIC

Existing static binary instrumentation techniques [18], [25], [26], [33] process the PIC using a simple approach. In the case of ×86 architecture, they identify a function call such as __i686.get_pc_thunk.bx, and trace the return value which is generated from the call instruction via a simple analysis. After finding the instruction responsible for computing an address with the return value (e.g., ADD in the example above), they correct its relative-address. Contrary to this, for the ARM architecture, existing techniques find instructions that have the PC register as operand (e.g., the first and second ADD in the example above), and then track the instructions that generate the relative-address (e.g., LDR, MOVW, MOVT) via backward slicing [34]. In summary, to the best of our knowledge, existing techniques only process relative-addresses that are computed directly with the PC value (referred as to *PC-relative instructions*), and do not consider other relative-addresses computed with the base register (referred as to *base-relative instructions*).

Existing techniques do not consider the range between the base address and the target address of *base-relative instruction*, thereby they cannot handle the case instrumentation code is inserted within the range. According to our observations, the *base-relative instruction* is often used in various situations. For example, in the case where the start address of an array placed on top of a subroutine is computed, the start address of the subroutine is first computed, after which the start address of the array is computed based on the computed address. In another case, the address of a specific instruction is first computed, and then several relative-addresses are added to the computed address depending on a condition. In addition, in ARM architecture, owing to the characteristics of fixed-length instructions, *base-relative instructions*, which involve either *base-plus-offset addressing* or *base-plus-index addressing*, are frequently used. Therefore, the *base-relative instruction* should be considered to handle all the relative-addresses in PIC on ARM.

III. CHALLENGES

Compared to ×86 architecture, ARM architecture has three unique features. First, many instructions (e.g., `LDR`, `MOV`, and `ADD`) can directly read from and write to the PC register, because the PC register is general-purpose register. Second, because of the limitation introduced by the instruction length,[3] the relative-address may be obtained via more than one instruction. Third, in some instructions, an operand, such as immediate and register, can be scaled (e.g., `LSL` and `ROR`). These features complicate address computations in ARM architecture more than in ×86 architecture, making it difficult to correct relative-addresses. In this section, we discuss these difficulties in more detail.

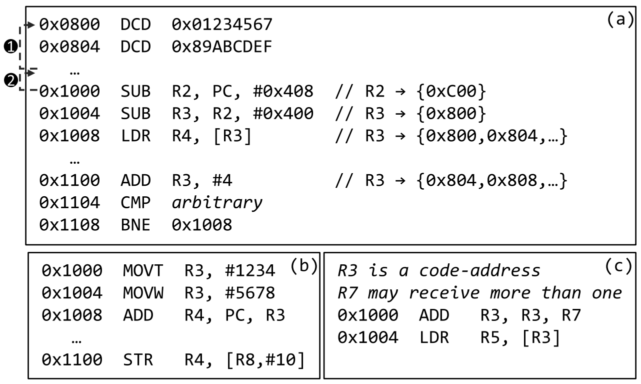
A. DIFFICULTY IN RELATIVE-ADDRESS CORRECTION IN ALL ADDRESS COMPUTATIONS

Base-relative instructions as well as *PC-relative instructions* need to be considered to ensure that all address computations in the PIC are processed. In this subsection, we demonstrate by example the difficulty associated with processing *base-relative instructions* correctly using existing techniques.

The most difficult situation in the relative-address correction is when the address computation is continuous, as shown in the example (a) on Figure 1. In this example, the start address `0×800` of the array in the literal pool is computed at `0×1004`, and the address is repeatedly added with 4 bytes at `0×1100`.[4] Suppose some data are inserted into `0×900` and `0×E00`, in which case the relative-address of `SUB` at `0×1004`(❶) and that of `SUB` at `0×1000`(❷) are affected, respectively. Because existing techniques only process *PC-relative instructions*, it only considers relative-addresses that are computed with the `PC` value. Thus, only the `SUB` at `0×1000`(❷) is considered despite the `SUB` at `0×1004`(❶)

[3]In ARM architecture, instructions are fixed-length (`ARM mode`: 4 bytes, `Thumb mode`: 2 or 4 bytes)

[4]The instructions at `0×1004` and `0×1100` compute the result of *base-plus-offset addressing* and write the address to register `R3`.



```
0x0800  DCD  0x01234567
0x0804  DCD  0x89ABCDEF
   …
0x1000  SUB  R2, PC, #0x408  // R2 → {0xC00}
0x1004  SUB  R3, R2, #0x400  // R3 → {0x800}
0x1008  LDR  R4, [R3]        // R3 → {0x800,0x804,…}
   …
0x1100  ADD  R3, #4          // R3 → {0x804,0x808,…}
0x1104  CMP  arbitrary
0x1108  BNE  0x1008
```

```
0x1000  MOVT  R3, #1234
0x1004  MOVW  R3, #5678
0x1008  ADD   R4, PC, R3
   …
0x1100  STR   R4, [R8,#10]
```

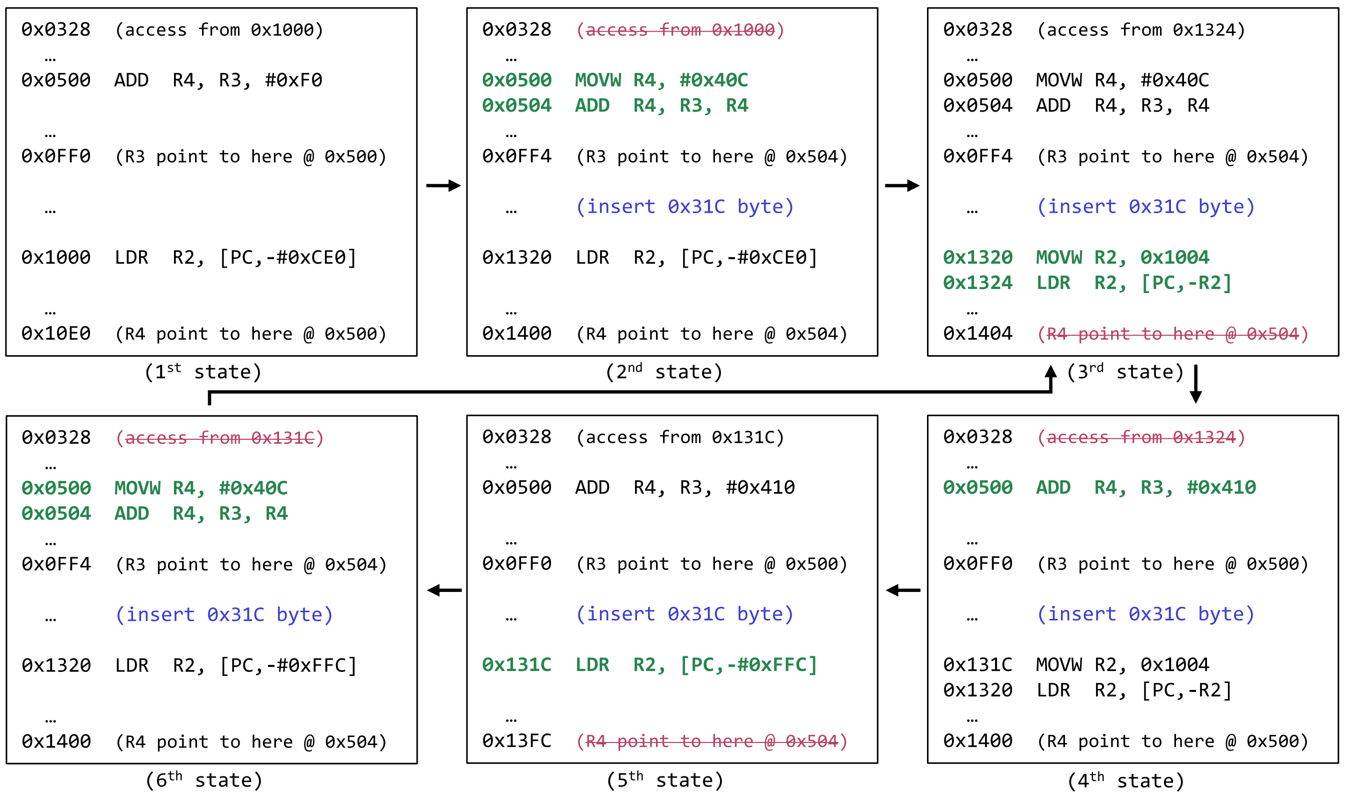
```
R3 is a code-address
R7 may receive more than one
0x1000  ADD   R3, R3, R7
0x1004  LDR   R5, [R3]
```

FIGURE 1. Examples, simplified for explanation, of PIC in COTS binaries (The annotations in the example on the (a) indicate the program state immediately after the execution of each instruction, and the result of the value-set analysis at `0×1008` and `0×1100` may be over-approximated.)

also need considering. As a result, the start address of the array is computed incorrectly.

To process *base-relative instructions* together, suppose we use backward slicing at all `ADD` and `SUB` instructions. However, it is unclear which result should be used to correct the relative-address. Even if we were to use the result at `0×1100`, which is the superset of the others, it is unclear which instruction to fix or how to fix it. That is, we need to know the base address and the target address (or *effective address*) of each instruction in order to identify a change within the range and correct the variation. To complement the lack of information, suppose that we use generic value-set analysis [35] to collect addresses and constants at each program point. Because of the limitation of the analysis, the result may be over-approximated [36], [37]. In this example, as the results at `0×1008` and `0×1100` are over-approximated, the values of `R3` at `0×1100` may include `0×900` and `0×E00` although these two addresses are outside the range of the array. When performing a correction based on this result, the `ADD` at 0×1100 also be considered as well as the two `SUB` at 0×1000 and 0×1004, which cause the correction to be incorrect.

As an alternative, suppose we attempt to use dynamic techniques to correct relative-addresses immediately before instructions that use computed addresses. For example, an indirect jump address at the `BLX` instruction, a dereferenced address at the `LDR` instruction, and an address value passed to a function parameter at a call instruction may be corrected immediately before each instruction. As illustrated (a) on Figure 1, the correction code can be placed before `0×1008`; however, it is unclear in which way to handle the entry point of the loop. In (b) on the Figure, the computed address is stored directly in memory (e.g., a field of an object); hence, the method cannot process these cases. As shown in (c) on the Figure, if more than one computed address is transferred at a point, it is difficult to correct every address individually. In conclusion, it is difficult to handle both *pc-relative* and *base-relative* instructions appropriately with existing techniques and heuristics alone.



```
0x0328  (access from 0x1000)
  …
0x0500  ADD  R4, R3, #0xF0
  …
0x0FF0  (R3 point to here @ 0x500)
  …
0x1000  LDR  R2, [PC,-#0xCE0]
  …
0x10E0  (R4 point to here @ 0x500)
```
(1st state)

```
0x0328  ~~(access from 0x1000)~~
  …
0x0500  MOVW R4, #0x40C
0x0504  ADD  R4, R3, R4
  …
0x0FF4  (R3 point to here @ 0x504)
  …     (insert 0x31C byte)
0x1320  LDR  R2, [PC,-#0xCE0]
  …
0x1400  (R4 point to here @ 0x504)
```
(2nd state)

```
0x0328  (access from 0x1324)
  …
0x0500  MOVW R4, #0x40C
0x0504  ADD  R4, R3, R4
  …
0x0FF4  (R3 point to here @ 0x504)
  …     (insert 0x31C byte)
0x1320  MOVW R2, 0x1004
0x1324  LDR  R2, [PC,-R2]
  …
0x1404  ~~(R4 point to here @ 0x504)~~
```
(3rd state)

```
0x0328  ~~(access from 0x1324)~~
  …
0x0500  ADD  R4, R3, #0x410
  …
0x0FF0  (R3 point to here @ 0x500)
  …     (insert 0x31C byte)
0x131C  MOVW R2, 0x1004
0x1320  LDR  R2, [PC,-R2]
  …
0x1400  (R4 point to here @ 0x500)
```
(4th state)

```
0x0328  (access from 0x131C)
  …
0x0500  ADD  R4, R3, #0x410
  …
0x0FF0  (R3 point to here @ 0x500)
  …     (insert 0x31C byte)
0x131C  LDR  R2, [PC,-#0xFFC]
  …
0x13FC  ~~(R4 point to here @ 0x504)~~
```
(5th state)

```
0x0328  ~~(access from 0x131C)~~
  …
0x0500  MOVW R4, #0x40C
0x0504  ADD  R4, R3, R4
  …
0x0FF4  (R3 point to here @ 0x504)
  …     (insert 0x31C byte)
0x1320  LDR  R2, [PC,-#0xFFC]
  …
0x1400  (R4 point to here @ 0x504)
```
(6th state)

FIGURE 2. **Example of *cyclic correction problem* (The instruction set for that code is ARM mode; thus, the value of the PC register is +8. The R3 of the ADD is always assumed to point to the correct address.)**

B. DIFFICULTY IN EFFICIENT RELATIVE-ADDRESS CORRECTION

In some ARM instructions an operand can be scaled, and this feature enables a larger offset to be expressed by fewer instructions. For example, in the case of the ADD instruction of ARM mode, an 8-bit immediate value can be scaled by the ROR operation (0, 2, 4, . . . , 30), and its register can be scaled by the RRX, LSL, and LSR operations (0, 1, 2, . . . , 31). Because of this feature, the range of offsets an instruction can express is discontinuous. In ×86 architecture, ADD has immediate values of 8 bits, 16 bits, and 32 bits. Unlike ARM architecture, as the size of the immediate value increases, the size of the instruction becomes larger. In addition to a single instruction, there is also the discontinuity of offset expressed by more than one instruction.

Because of this characteristic, the correction process may not terminate when all relative-addresses are corrected using optimized instructions. We refer to this problem as the *cyclic correction problem*, and Figure 2 shows a related example. The first state is the original state in which there are two relative-addresses on the LDR[5] and ADD[6] instructions. Assume that 0×31C bytes of data are inserted between 0×FF0 and 0×1000. In this case, the relative-address (0×FF0↦0×10E0) of ADD and the relative-address (0×1000↦0×28) of LDR are affected. In the second state, new MOVW and ADD instructions are used to correct the relative-address of ADD, and the relative-address of LDR remains incorrect. In the third state, new MOVW and LDR instructions are used to correct the relative-address of the LDR. In this state, because an additional instruction is used to correct the relative-address of the LDR, the previously corrected relative-address of the ADD becomes incorrect. In the fourth state, the relative-address of ADD is corrected again. Although the relative-address 0×410 is larger than the relative-address immediately preceded (0×40C), it can be expressed by a single ADD instruction, which uses the ROR operation. Therefore, the relative-address of the LDR, which becomes incorrect again, is corrected again in the fifth state. However, because the reduced relative-address of the LDR can be corrected by a single instruction, it causes the relative-address of the ADD to be incorrect again. The above discussion shows that the correction process is unable to terminate during cyclic correction. ($3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 3 \rightarrow \cdots$).

The *cyclic correction problem* makes it difficult to rewrite a binary efficiently by using a simple correction algorithm. Intuitively, this problem can be avoided if the correction algorithm is considered to handle all relative-addresses to be corrected with a size of 4 bytes. If so, regardless of the instruction mode, an additional 8 bytes (e.g., the combination of MOVW and MOVT for a 4-byte relative-address) is required

[5]The range of the offset is ROR(imm, $2 \times n$) where $0 \leq imm < 0\times100$ and $0 \leq n < 15$. The base and target addresses are 0×101C and 0×28, respectively.

[6]The range of the offset is $-0\times1000 < imm < 0\times1000$. The base and target addresses are 0×FF4 and 0×1100, respectively.

at each point where correction of the relative-address occurs. In some cases, a larger size of bytes may be required. This heuristic method can solve the *cyclic correction problem*, but generates large space overhead and runtime overhead because PIC consists of many address computations related to the relative-address. However, our binary instrumentation technique is aimed at minimizing the space overhead and runtime overhead by inserting an instrumentation code inside the original code; thus, this heuristic method is not suitable to implement our technique.

IV. OVERVIEW OF APPROACH

Although our technique is intended for both AArch32 and AArch64, the following explanation is based on AArch32. Compared to AArch64, rewriting AArch32 binaries is more challenging because additional features need to be taken into account such as 16-bit and 32-bit instructions, two instruction modes, dynamic instruction mode switching, and 2-byte and 4-byte instruction alignment. Moreover, as AArch32 continues to be used in a wide variety of devices. Therefore, AArch32 is more suitable for describing our technique.

A. TARGET BINARY

In this research, the type of target binary we are concerned with is as follows. First, the binaries are stripped binaries without any relocation information or symbols, except those necessary for dynamic linking. Second, the binaries are compiled in a position-independent fashion, including all the binaries generated by the compiler options `-fPIC` and `-fPIE`. Third, the binaries do not contain self-modifying code. Fourth, the binaries are in the executable and linkable format (ELF) of which the target architecture is either AArch32 or AArch64.

B. BINARY MODIFICATION CAPABILITY

The modification capability of `REPICA` can be divided into three categories. First, some code can be inserted before or after any instruction, and any instruction can be modified and removed. This capability can be used for instrumenting binaries. Second, in the literal pool, relative-addresses and some data such as a 4-byte constant can be modified or removed. This feature can be applied for the optimization of the corrected instructions (see §V-E). Third, additional data can be inserted before or after each literal pool and data segment. This feature enables additional data to be inserted without creating a new data segment. `REPICA` retains the rewritten binary as original as possible using these three capabilities.

C. ASSUMPTION ON ADDRESS COMPUTATION

Our aim is to identify and correct relative-addresses that are affected by instrumentations in the instruction area. This requires accurate analysis results and clear correction method, thereby we make two assumptions. The first assumption pertains to the characteristic of the value we need, and can be used to narrow the scope of our value-set analysis, thereby preventing over-approximation of the analysis result. The second assumption relates to the characteristic of the *code-address*, and can reduce the type of address computations we need to consider, thereby improving the uniformity of the correction method. The two assumptions are as follows:

A1. The relative-addresses used to compute a particular address are obtained in the subroutine in which the computation is performed. The particular address here refers to the start address of the subroutine, the start address of the data, and other base addresses; however, it does not include the addresses computed to access the interior of data chunks such as arrays or recursive data structures. To verify this assumption, we checked all values computed with *PC-based addresses* in approximately 1,200 COTS binaries[7] using our value-set analysis (see §V-C). In our observations, we were unable to find a case that violates this assumption except for the following case:

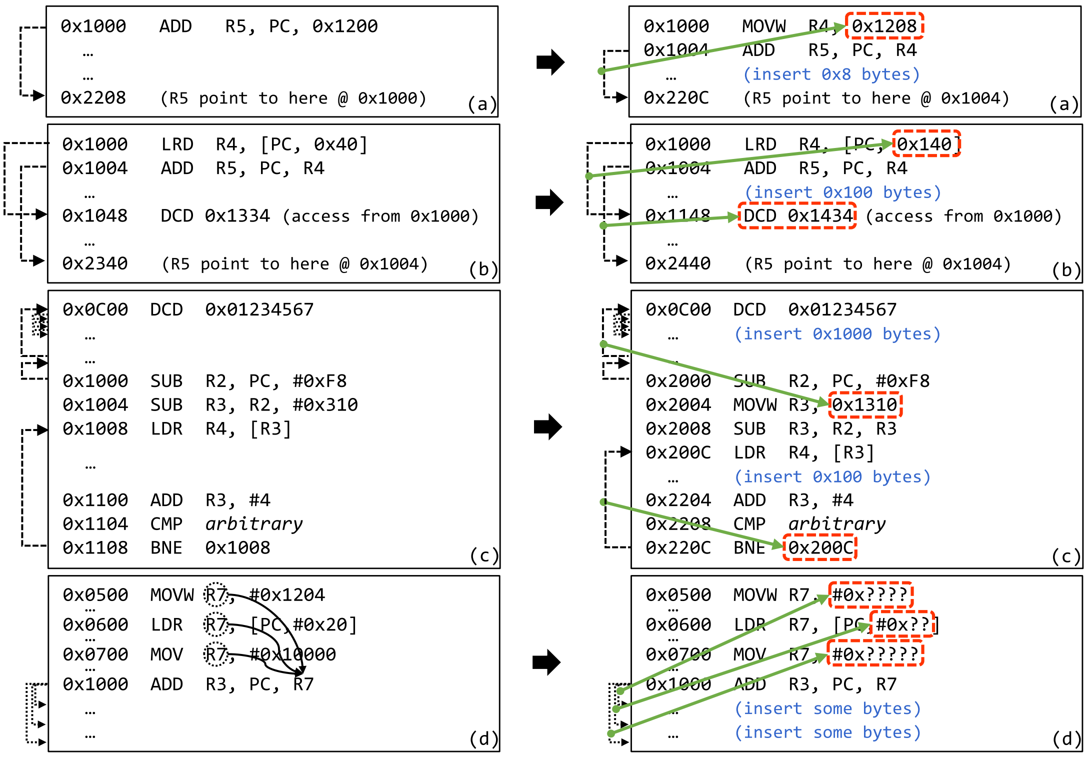
```
0x1000   CMP     R3, #0xC
0x1004   ADDLS   PC, PC, R3, LSL#2
0x1008   B       label  // default
0x100C   B       label  // case 0
                   ...
0x103C   B       label  // case 12
```

`R3`[8] in this example is actually used as the index of the jump table (which is the list of `B` from `0x1008` to `0x103C`), and this case is handled separately. The relative-addresses used in computations for the particular address consist of the immediate values of instructions in the subroutine and constants in the literal pool. Based on this assumption, we perform an intraprocedural value-set analysis to collect relative-addresses.

A2. In the *base-relative instructions*, if the type of the values transferred to the base register is a *code-address*, the number of values transferred is always one. For various reasons, one or more values can be transferred to the base register. For example, when an array is accessed in a loop or the field of an arbitrary object is accessed, more than one of the *data-addresses* may be transferred to the base register of an instruction. However, to the best of our knowledge, cases in which multiple *code-addresses* are transferred to a base register for computing new addresses, except for self-modifying code, rarely occur. To verify this assumption, we checked all base registers in approximately 1,200 COTS binaries using our value-set analysis. We performed intraprocedural analysis, which is based on the underlying assumption that *code-addresses* delivered from the outside are not recomputed. As our observations did not reveal a case that violates this assumption, it is not necessary to consider situations in which multiple base addresses and one or more relative-addresses are computed at one point. Thus, all target

[7]This experimentation is performed with the binaries that are disassembled with full coverage among the first dataset in §VI.

[8]The value of `R3` may be generated indirectly with external values. In general, the index values of a jump table are not directly in the subroutine.



```
0x1000   ADD   R5, PC, 0x1200
  …
  …
0x2208   (R5 point to here @ 0x1000)          (a)
```

```
0x1000   MOVW  R4, 0x1208
0x1004   ADD   R5, PC, R4
  …      (insert 0x8 bytes)
0x220C   (R5 point to here @ 0x1004)          (a)
```

```
0x1000   LRD  R4, [PC, 0x40]
0x1004   ADD  R5, PC, R4
  …
0x1048   DCD 0x1334 (access from 0x1000)
  …
0x2340   (R5 point to here @ 0x1004)          (b)
```

```
0x1000   LRD  R4, [PC, 0x140]
0x1004   ADD  R5, PC, R4
  …      (insert 0x100 bytes)
0x1148   DCD 0x1434 (access from 0x1000)
  …
0x2440   (R5 point to here @ 0x1004)          (b)
```

```
0x0C00  DCD   0x01234567
  …
  …
0x1000  SUB   R2, PC, #0xF8
0x1004  SUB   R3, R2, #0x310
0x1008  LDR   R4, [R3]
  …
0x1100  ADD   R3, #4
0x1104  CMP   arbitrary
0x1108  BNE   0x1008                          (c)
```

```
0x0C00  DCD   0x01234567
  …     (insert 0x1000 bytes)

0x2000  SUB   R2, PC, #0xF8
0x2004  MOVW  R3, 0x1310
0x2008  SUB   R3, R2, R3
0x200C  LDR   R4, [R3]
  …     (insert 0x100 bytes)
0x2204  ADD   R3, #4
0x2208  CMP   arbitrary
0x220C  BNE   0x200C                          (c)
```

```
0x0500  MOVW  R7, #0x1204
0x0600  LDR   R7, [PC, #0x20]
0x0700  MOV   R7, #0x10000
0x1000  ADD   R3, PC, R7
  …
  …                                           (d)
```

```
0x0500  MOVW  R7, #0x????
0x0600  LDR   R7, [PC, #0x??]
0x0700  MOV   R7, #0x?????
0x1000  ADD   R3, PC, R7
  …     (insert some bytes)
  …     (insert some bytes)                   (d)
```

FIGURE 3. Examples of our approach to correct relative-addresses (The original code and corrected instrumented code are shown on the left and right, respectively. Dashed arrows denote the *relative range*.)

addresses can be corrected accurately only by adjusting the relative-addresses.[9]

D. INDIVIDUAL RELATIVE-ADDRESS CORRECTION

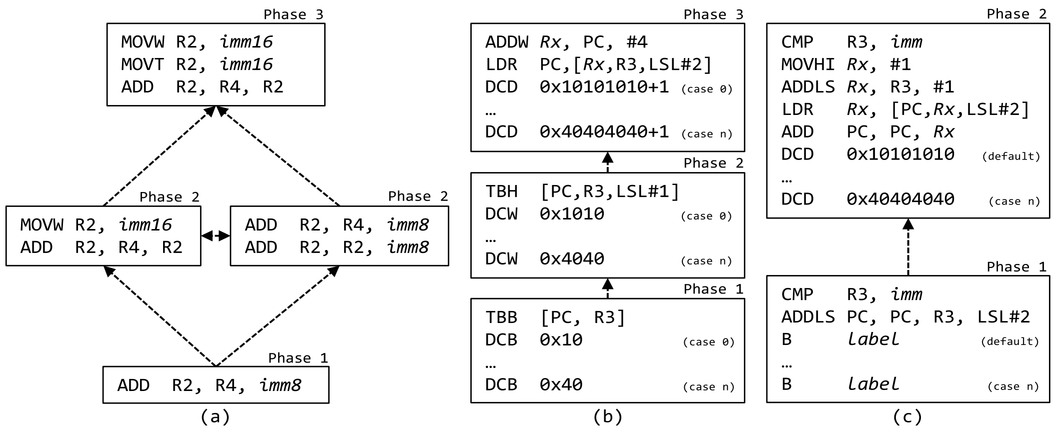
In this study, we define the range between the base address and the target address (or *effective-address*) on an instruction as the *relative range*. On the left side of the figure 3, each instruction at 0×1000 has a *relative range* pointing from the current address to the target address. An instruction can have one or more *relative ranges*, as shown the fourth example (d) in the Figure 3 with ADD at 0×1000. The starting point of the *relative range* may not be the address of the instruction, and the SUB at 0×1004 in the third example (c) shows this case. In this example, the ADD at 0×1100 also has one or more *relative ranges* which are fallen within data area. We do not consider these *relative ranges* practically because we do not modify the data area.

In order to correct all relative-addresses properly, the most important challenge is to find all instructions with *relative ranges* accurately. Specifically, the difficulty lies in accurately collecting its base address and offset values in the *base-relative instruction*. Once this problem could be solved, in the example (a) on Figure 1, we could confirm that the SUB at 0×1004 and 0×1100 are related to the address computation, and this enables us to clearly identify the instruction which is need to correct. Because the base address of the ADD at 0×1100 is a *data-address* and this *relative range* is related to the data area, it can be clearly seen that the instruction is irrelevant to the correction.

For this purpose, we designed a value-set analysis of which the domain is in the form of a symbol, which collects *PC-based addresses*. The main reason of using symbol form is to track the generation location of the relative-address and to distinguish addresses from constants. In addition to this, the address information contained in the symbol can be used to avoid over-approximation of the collected values. Because the values we aim to collect are localized and limited, we can collect them without incurring over-approximation. Thus, we designed a value-set analysis specialized for PIC (see §V-C); because generic value-set analysis does not suit for our purpose.

On the basis of the precisely collected *relative ranges*, we individually correct the *relative ranges* that were affected by instrumentation. Figure 3, on the right, shows the instrumented code which contains the corrected instructions, and shows the main idea in our approach. (a) shows that one *relative range* is corrected by the added 4-byte instruction to express a larger relative-address. (b) shows that the LDR and the ADD are corrected individually. Note that the relationship between two instructions is not taken into account when performing a correction. (c) contains more than three *relative*

[9]In general address computations, if the base address is a *data-address*, the computed address is also a *data-address*, so we do not have to consider this case.



```
(a)
Phase 3
MOVW R2, imm16
MOVT R2, imm16
ADD  R2, R4, R2

Phase 2
MOVW R2, imm16
ADD  R2, R4, R2

Phase 2
ADD  R2, R4, imm8
ADD  R2, R2, imm8

Phase 1
ADD  R2, R4, imm8
```

```
(b)
Phase 3
ADDW Rx, PC, #4
LDR  PC,[Rx,R3,LSL#2]
DCD  0x10101010+1  (case 0)
…
DCD  0x40404040+1  (case n)

Phase 2
TBH  [PC,R3,LSL#1]
DCW  0x1010        (case 0)
…
DCW  0x4040        (case n)

Phase 1
TBB  [PC, R3]
DCB  0x10          (case 0)
…
DCB  0x40          (case n)
```

```
(c)
Phase 2
CMP    R3, imm
MOVHI  Rx, #1
ADDLS  Rx, R3, #1
LDR    Rx, [PC,Rx,LSL#2]
ADD    PC, PC, Rx
DCD    0x10101010    (default)
…
DCD    0x40404040    (case n)

Phase 1
CMP    R3, imm
ADDLS  PC, PC, R3, LSL#2
B      Label         (default)
…
B      Label         (case n)
```

FIGURE 4. Examples of instruction extension design (*Rx* is a register that is additionally required for instruction extension, and is obtained through liveness analysis.)

ranges, of which two are affected by instrumentation. Note that we correct the *relative range* of the SUB at 0×1004, but the *relative range* of SUB at 0×1000 is not involved. In (d), each *relative range* is corrected at each generation location of the transferred relative-addresses.

E. STEPWISE INSTRUCTION EXTENSION

Both the *cyclic correction problem* and optimization problem can be addressed by extending the instruction irreversibly step by step. By dividing the instruction extension into several phases, it is possible to correct the relative-address by using as few instructions as possible. Furthermore, by increasing the extension phase in one direction, the cyclic correction can be avoided. With these two methods, we solve the *cyclic correction problem* and the optimization problem simultaneously.

Figure 4 shows the instruction extended by phase to fit the size of the relative-address perfectly following our instruction extension design. In this Figure, (a) is an example that applying our extension design of ADD instruction, showing stepwise extension in sizes of 4, 8, and 12 bytes. If it is extended to an 8-byte expression (phase 2) in the correction process, the 4-byte expression (phase 1) is no longer used in the subsequent process. That is, the subsequent process uses the expression in the same or higher phase. In phase 2, there are two expressions, but it can be further divided to cover a wider range of relative-addresses using the same bytes. In REPICA, we designed extension phases for each instruction which use relative-address, as shown in this example.

In some cases, a relative-address should not be extended individually, shown as (b) in Figure 4. The example contains a jump table for indirect jump, which consists of relative-addresses of the same size. In general, the offset calculation of the jump table involves multiplying or shifting the index value to take into account the size of the relative-address (or the instruction length). Therefore, if only some relative-addresses are changed to become larger, the offset calculation of the jump table is incorrect. For this reason, we modify the entire table when the size of a relative-address needs to be larger, as in the method proposed by RevARM [18].

There exist jump tables that do not consist of relative-addresses, which need to be considered in a special way. (c) in Figure 4 shows a jump table constructed with jump instructions. In this example, the index R3 is used to move to the address of the B instructions that performs the actual jump. Therefore, in this case, only some B instructions should not be extended; thus, we replace the list of 4-byte instructions with a jump table consisting of the 4-byte relative-addresses.

V. DESIGN AND IMPLEMENTATION

This section covers the design and implementation of REPICA. First, an overview of REPICA is provided, and then each component is discussed in detail in each subsection following the execution flow.

A. OVERVIEW

We implemented REPICA's prototype based on Python. We used Capstone 3.05 [38] for instruction decoding and implemented all the other parts ourselves. Figure 5 shows the REPICA architecture, which consists of the disassembly and instrumentation modules. The input of the disassembly module is a target binary, and that of the instrumentation module is the instrumentation specification (see Figure 8) and the results of the disassembly.

First, the disassembly module disassembles the text segments of the input binary (§V-B) and performs the value-set analysis based on the disassembled code (§V-C). Based on the result of the analysis, it extracts the indirect jump addresses and new subroutine addresses, and disassembles at the extracted addresses again. This process is repeated until no new address is found.

Next, in the instrumentation module, it instruments the disassembled code along the instrumentation specification. Then, it corrects the relative-addresses that are affected by applying the instrumentation (§V-D) and optimizes the corrected relative-addresses (§V-E). Lastly, after rewriting the instrumented code corrected to the new binary, the entire process is complete (§V-F).

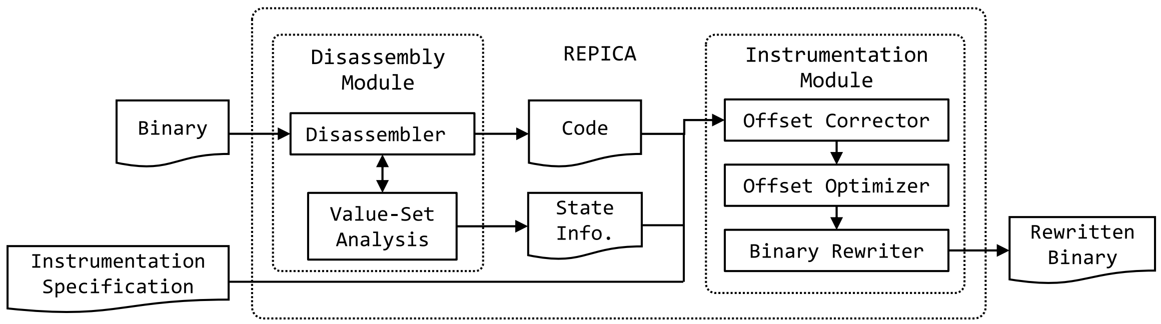


FIGURE 5. Architecture of `REPICA`.

B. DISASSEMBLER

Because the correctness of the rewriting technique depends on the disassembly accuracy, it is important to clearly separate the literal pool and code in the text segments. Therefore, we use only the recursive traversal approach (a.k.a. recursive disassembly) to guarantee that the instructions that are disassembled are real instructions. For this approach to be successful, in general, it is necessary to overcome three challenges: (1) identifying absolute-addresses in the code, (2) identifying the types of addresses, and (3) identifying indirect control flow. Fortunately, in PIC, absolute-addresses cannot be placed directly on code, which solves the first challenge. This subsection contains details of how the two remaining challenges are addressed.

1) INDIRECT CONTROL FLOW IDENTIFICATION

In PIC, absolute-addresses placed in the data segments, such as addresses of the virtual method table, contain relocation information. Thus, we only have to consider the indirect jump via jump table among the indirect control flow transfer. The following explains how to identify the type of indirect jump via jump table and how to identify the size of the table:

a: INDIRECT JUMP TYPE IDENTIFICATION

In ARM, because the PC register can be read to and written from, an indirect jump can occur according to various patterns. We use an existing technique [4] and some patterns to cover all possible patterns. First, to identify the indirect jump via jump table, we search for an indirect jump instruction or an instruction that writes a value to the PC register within the disassembled code. Subsequently, we derive the expression of the target address by using backward slicing. If the expression has the form of `*(TBL_BASE+IDX)+JMP_BASE`, we obtain the jump base address and the jump table base address through the result of our value-set analysis. If not, we attempt to identify the ARM-specific indirect jump,[10] such as in (b) and (c) in Figure 4, using some patterns.

[10]Special functions can be used to generate switch patterns: Thumb mode toolchain helpers for compact switch (https://chromium.googlesource.com/chromiumos/platform/ec/+/refs/heads/master/core/cortex-m0/thumb_case.S)

b: JUMP TABLE SIZE INFERENCE

The information of the jump table disappears at compile time, and an index value may come from outside the program. Therefore, it is difficult to collect all values of the index through an analysis based on an abstract interpretation [36]. According to our observation, in the code generated by the modern compiler, the index value is validated before calculating the offset of the jump table. The validation is performed by comparing with a certain constant value. With the constant value and the condition field of the followed conditional jump instruction, we can infer the size of the jump table. The example in §IV-C indicates that the size of the jump table can be deduced from `CMP` at `0×1000` and `ADDLS` at `0×1004`. In most cases, the size can be inferred by using this method, but if this is not possible, the next method is used.

c: BRUTE-FORCE JUMP TABLE SEARCH

Similar to the method in `BinCFI` [4], our method is sequentially to read and verify each target address in the jump table until the address is invalid. The validity of the target address is determined by the location where the address points to, which is based on the underlying assumption that jump table targets are intraprocedural. If it is lower than the start address of the subroutine or higher than the start address of the next subroutine that is found, it is considered to be invalid. If the target address is not invalid, we will try to disassemble at the target address and at all subroutine addresses obtained during this disassembly. If there are no errors in the entire disassembly, the target address is judged to be valid.

2) ADDRESS TYPE IDENTIFICATION

We examine each arithmetic instruction to determine whether it computes a new address, and if the result is a new *PC-based address*, we will check whether the address is a *code-address*. It is usually difficult to determine the type of a given address. In order to solve this challenge, we first perform a simple pre-identification process as follows: (1) determine whether the address falls within data segments, (2) determine whether the address falls within the disassembled instruction area, (3) determine whether the address falls within the area dereferenced by `LDR` instructions. If the type of the address cannot be determined in the pre-identification process, we will attempt to disassemble at the address to check whether the address is a *code-address* by using the same way as the verification in *Brute-force jump table search*.

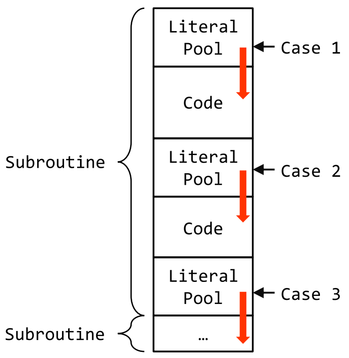


FIGURE 6. Cases in which identification through disassembly produces incorrect results.

Unfortunately, this verification method may often produce incorrect results. Figure 6 shows cases where disassembly starts at an address that falls within the literal pool, and proceeds to the code area without error. Based on our observations, these erroneous outcomes often occur. In order to estimate the failure probability of this verification method, we disassembled all values in the 4-byte range to find out how many values were disassembled without error. The following table presents the result[11]:

Instruction set	Success	Failure	Succ. rate
AArch32 (ARM)	0xDCDC6637	0x232399C9	86.27%
AArch32 (THUMB)	0xF33012C1	0xCCFED3F	94.99%
AArch64	0x4CEE8832	0xB31177CE	30.05%

In particular, the resultant values for the `ARM mode` and the `Thumb mode` of AArch32 are very high at 86.27% and 94.99%, respectively. In addition, the end point of the literal pool is usually aligned by `NOP` instructions. For these reasons, the probability of failure is high when trying to identify the *data-address* which point to nearby the end of the literal pool.

We leverage the placement property of the literal pool to compensate for the weakness of this verification method. The literal pool includes immutable local data, such as floating point, integer, and string, and relative-addresses for address computation. For example, in C language, local variables of the `const` type are placed in the literal pool, but variables that can be seen with other subroutines or mutable variables are not placed in this pool. Thus, each literal pool is placed adjacent to the subroutine which uses it, as shown in Figure 6. This means that the code of the subroutine is placed in one bundle together with the data that are locally accessed by this code. This characteristic is obvious in the code generated by modern compilers. According to this characteristic, if a subroutine computes a *data-address*, the address can be expected to fall within the data segments or the literal pool adjacent to the subroutine.

[11]In the experiment, we used `Capstone 3.05`, an open source.

This characteristic enables us to compensate for the weakness of the address type identification method. The type of an address computed in a subroutine is identified after the subroutine is completely disassembled. Therefore, the first and second cases in Figure 6 can be easily identified. We use two criteria to identify the third case. First, for a case in which the computed address is higher than the address of last instruction and lower than the dereferenced addresses in the subroutine, the address is regarded as a *data-address*. Second, if the start address of other subroutines fall within the range of the entry block disassembled at the computed address, the address is regarded as a *data-address*. This approach enables us to identify *data-addresses* accurately.

C. VALUE-SET ANALYSIS

Our value-set analysis is an intraprocedural analysis, and it aims to propagate all *PC-based addresses* and constants to collect *relative ranges* of interest. Unlike generic value-set analysis, our domain is in the form of symbols and is specialized for rewriting PIC.

$$\begin{aligned}
stt \in \quad & State &= \; & Registers + Stack \rightarrow 2^{Value} \\
reg \in \quad & Registers &= \; & \{\mathrm{R0}, \cdots, \mathrm{R12}, \mathrm{SP}, \mathrm{LR}, \mathrm{PC}\} \\
stk \in \quad & Stack &= \; & \{\mathrm{SP}\} \times off \\
imm \in \quad & Immediate &= \; & \left\{-2^{32}, \cdots, -1, 0, 1, \cdots, 2^{32}-1\right\} \\
off \in \quad & Offset &= \; & \mathbb{Z}
\end{aligned}$$

$$\begin{aligned}
addr \in \; & \left\{0, 1, 2, \cdots, 2^{32}-1\right\} \\
mode \in \; & \{\mathrm{ARM}, \mathrm{THUMB}\} \\
val \in \; & Value \\
val ::= \; & \mathrm{IMM}(imm \times addr) \\
& \mid \mathrm{PC}(mode \times addr) \\
& \mid \mathrm{MOV}(val \times addr) \\
& \mid \mathrm{MOVT}(val \times val \times addr) \\
& \mid \mathrm{ADD}(val \times val \times addr) \\
& \mid \mathrm{SUB}(val \times val \times addr) \\
& \mid \mathrm{LDR}(val \times val \times addr) \\
& \mid \mathrm{ADR}(mode \times val \times addr)
\end{aligned}$$

FIGURE 7. Domain of value-set analysis in `REPICA`.

1) ANALYSIS DOMAIN

In ARM, the address is typically computed via multiple stages, and the computed address may be recomputed. Thus, when given a value, it should be possible to accurately track the instructions associated with the value. Figure 7 shows the domain of the value-analysis in `REPICA` designed to process multi-staged address computation.

In our domain, each value has its own generation address, and `PC` and `ADR` values additionally contain instruction mode information. This structure provides information about where and how a given value was generated. Another advantage of this structure is that it is path-sensitive, even if the state of the analysis is not separately identified for each path. Thus, our

TABLE 1. **Transfer function of value-set analysis in `REPICA` (*d*, *m*, and *n* are general-purpose registers, *i* is the immediate value, and *a* is the address of the corresponding instruction.)**

Instruction Format		Propagation
mov	d, m	$stt(d) \leftarrow stt(m)$
mov	d, i	$stt(d) \leftarrow \{\text{MOV}(\text{IMM}(i,a),a)\}$
movt	d, i	$stt(d) \leftarrow \{\text{MOVT}(v,\text{IMM}(i,a),a) \mid stt(d) \neq \phi \wedge v \in stt(d)\}$
add	d, m, n	$stt(d) \leftarrow \{\text{ADD}(v_1,v_2,a) \mid v_1 \in stt(m) \wedge v_2 \in stt(n) \wedge \neg checkIter(v_1,a) \wedge \neg checkIter(v_2,a) \wedge$ $(isPcBased(v_1) \wedge isCodeAddr(v_1) \vee isPcBased(v_2) \wedge isCodeAddr(v_1))\}$
add	d, m, i	$stt(d) \leftarrow \{\text{ADD}(v,\text{IMM}(i,a),a) \mid v \in stt(m) \wedge isPcBased(v) \wedge \neg checkIter(v)\}$
sub	d, m, n	$stt(d) \leftarrow \{\text{SUB}(v_1,v_2,a) \mid v_1 \in stt(m) \wedge v_2 \in stt(n) \wedge isPcBased(v_1) \wedge isCodeAddr(v_1) \wedge \neg checkIter(v_1,a)\}$
sub	d, m, i	$stt(d) \leftarrow \{\text{SUB}(v,\text{IMM}(i,a),a) \mid v \in stt(m) \wedge isPcBased(v) \wedge isCodeAddr(v) \wedge checkIter(v)\}$
ldr	$d, [m, n]$	$stt(d) \leftarrow \{\text{LDR}(v_1,v_2,a) \mid v_1 \in stt(m) \wedge v_2 \in stt(n) \wedge isPcBased(v_1) \wedge \neg checkIter(v_1,a) \wedge \neg checkIter(v_2,a)\}$
ldr	$d, [m, i]$	$stt(d) \leftarrow \{\text{LDR}(v,\text{IMM}(i,a),a) \mid v \in stt(m) \wedge isPcBased(v) \wedge \neg checkIter(v)\}$
ldr	d, [SP, i]	$stt(d) \leftarrow stt(\text{SP}, i)$
str	d, [SP, i]	$stt(\text{SP}, i) \leftarrow stt(d)$
adr	i	$stt(d) \leftarrow \{\text{ADR}(getMode(),\text{IMM}(i,a),a)\}$
others		$stt(d) \leftarrow \phi$ for each d $\in getDestinationRegs()$

domain consists of one state per program point, and each element of the state consists of a set of values.

Theoretically, many instructions can be used for address computation, but the instructions involved in actual address computation are only a fraction of the instructions. According to our observations, only those instructions mentioned in our domain are used in the ARM architecture. This can vary slightly depending on the compiler, but can be solved by adding value types to the domain.

In some cases, an address or a constant value is temporarily stored on the stack and reused. A situation such as this is accommodated by tracking the stack memory by adding a stack offset to the domain. Although these values can also be stored in other memory spaces, this rarely occurs, and our method is not designed to include these cases.

In some instructions, the sign of the offset or index is determined by the instruction encoding rather than the value itself (e.g., `LDR`, `STR`). Even though the arithmetic operations of ARM are based on a complement of these two, in these instructions, the computation of the *effective-address* is not. In our domain this is reflected by setting the type of the immediate value to an integer. Therefore, when the values of the symbols are evaluated, the calculation method is applied differently depending on each instruction semantic.

2) TRANSFER FUNCTION

As is characteristic of PIC, all addresses start with the `PC` value which is *code-address*. In subsequent instructions, the address may be recomputed to either the *code-address* or the *data-address*. If the base address is not *PC-based address*, the address computation is related to the heap or stack area. For this reason, the transfer function of our analysis aims to propagate these *PC-based addresses* and the constant values which may be used as relative-addresses. Table 1 provides the transfer functions specialized for propagating the values related to the address computation of the PIC.

The initial value is an empty set denoted by $\perp$, and the values are propagated forward from the beginning of each subroutine. If any of the operands in the instruction do not receive values from the previous path, the result is no longer propagated. Similarly, for instructions not related to address computation, the result is not propagated. That is, only the values associated with the address computation are propagated, and the absence of an operand value means that the operation is not associated with address computation.

Unlike the existing analysis, our transfer function has two propagation constraints that are based on two underlying assumptions about address computations. The first assumption is that, if the value generated at one point is to be used again at the same point, the computed addresses are for accessing the data area. Another assumption is that if an address is computed once to the *data-address*, it is not recomputed back to the *code-address*. These constraints are intended to prevent the collected values from being over-approximated. The details are as follows:

C1. If the value generated at one point is used again at the same point, the resultant value is not propagated. The `checkIter` function in Table 1 corresponds to this constraint, and its input is the current instruction address and values of each operand. This situation occurs frequently when indexing to an array or accessing a recursive data structure via a loop. Therefore, as the *relative range* related to this case falls within the data chunk region, it can be disregarded. In text segments other than literal pools, this rarely occurs except for self-modifying code. These constraints prevent the collected values from being over-approximated and ensure that our analysis terminates within a finite time. In the absence of this constraint, the analysis shown on (a) in Figure 1 would not terminate, causing the collected values at `0x1100` to become over-approximated.

C2. If a base address is not a *code-address* in an address computation, the resultant value is not propagated. The `isCodeAddr` function in Table 1 corresponds to this constraint, and its input is a value corresponding to the base address. This means that once a *data-address* is computed, it is used only for data access, and it is not used again to compute the address of an instruction. In fact, this is an auxiliary constraint, without which our analysis can perform the same work. Retention of this constraint enables unnecessary propagation to be reduced, whereby the unrelated *relative ranges* can be reduced in advance. As a result, the rewriting technique becomes scalable.

This transfer function cannot process the indirect jump via jump table completely. Specifically, because the index value of the jump table may be indirectly generated from an external value, the target values may not be propagated. Thus, in our disassembler, this is taken care of by a heuristic. The disassembler finds the addresses the jump table is targeting by using the table base address and the jump base address obtained from the analysis. In addition to this, the disassembler manages the landing pad addresses included in the exception handling section. These addresses are encoded in the DWARF format [39].

D. OFFSET CORRECTOR

1) INSTRUMENTATION SPECIFICATION

Figure 8 shows the basic structure of the instrumentation specification of `REPICA`. This specification has a list structure, with each element consisting of the address to be applied (*addr*), data to be inserted (*data*), and size of byte to be skipped (*skip*). Here, *skip* indicates the size of the original binary instructions to be removed from the target address at the time the binary is rewritten. To insert data, *data* is required to contain more than zero bytes and *skip* should be zero. Removing data requires *data* to contain zero bytes, and *skip* to contain more than one. Further, to replace data, *data* needs to contain more than one bytes and *skip* should be equal to the size of the data. This specification structure is also used to manage the correction information generated in the correction process.

$$
\begin{array}{rrcl}
spec \in & Specification & = & [addr \times data \times skip]^{+} \\
addr \in & Address & = & \left\{0, 1, 2, \cdots, 2^{32} - 1\right\} \\
data \in & InsertData & = & byte^{*} \\
byte \in & Byte & = & \{0, 1, 2, \cdots, 255\} \\
skip \in & SkipSize & = & \mathbb{N}
\end{array}
$$

FIGURE 8. Basic structure of instrumentation specification of `REPICA`.

2) SYMBOL AND RELATIVE RANGE

The symbol consists of the address to be corrected and the type of relative-address, and one symbol is paired with one *relative range*. When a modification falls within a *relative range*, the corresponding symbol is used to determine where and how to correct the relative-address. Depending on the way by which a relative-address is obtained, the correction method, which is distinguished by the symbol type, is different.

For example, if a relative-address is obtained via `LDR`, the symbol consists of the address which is dereferenced by the `LDR` and the `CONST` type. This type means that the correction target is a constant value. In another example, if a relative-address is obtained via `MOVW`, the symbol consists of the instruction address and the `INST` type. This type means that the correction target is an immediate value of the corresponding instruction. Different type of symbols can be added or removed depending on the relative-address generation pattern.

3) CORRECTION PROCESS

The offset corrector receives the disassembled code, instrumentation specification, and result of value-set analysis as the input. Algorithm 1 shows the correction process. First, it examines all instructions to identify which instructions contain *relative ranges* (`#3,has_relative_range`) based on the result of the value-set analysis. Subsequently, it creates `symbol` information for the *relative ranges* of each instruction (`#7,make_symbol`), and these symbols are added to the worklist (`#10`).

In the next step, it computes the amount of change within the *relative range* of the symbol (`#19,calc_stretched_size_within`), and then corrects the relative-address by the variation (`#21,correct_offset_at`). Relative-address variation is affected by two factors: instrumentation specifications and extension of the relative-address for correction. After the correction, the algorithm checks whether the size of the corrected instruction or data increased (`#23`). If the phase is increased, it identifies all the symbols of the *relative ranges* affected by the extended instructions caused by the correction (`#25,get_symbol_affected_from`). It then adds the affected symbols to the worklist (`#26`). The correction process will keep running until the worklist is empty (`#15`). In the final step, it corrects the meta data contained in the ELF header, the program header table, and the section header table (`#32,correct_elf_meta_info`).

Unlike the previous technique [18], in which the correction process was performed linearly, our correction process is based on the worklist. In a way that corrects the instructions in place, one correction may cause a series of different corrections, and these corrections may also cause other corrections. Because of this characteristic, the linear correction process may not be able to handle all the modifications that occur during the entire correction process. For example, suppose that the last relative-address correction during the linear correction process. Due to this correction, some relative-addresses may need additional corrections, and these corrections may also result in other corrections. Therefore, because the linear approach cannot perform all corrections perfectly, we process the correction based on a worklist.

4) ALIGNMENT

The ARM architecture requires an alignment by 2, 4, 8, and 16 bytes depending on the type of data or instruction. Similarly, each section also requires an alignment by a page size or by a specific size. If each alignment is adjusted after all corrections are completed, it causes a problem in that all of the relative-addresses must be corrected again. In this case, the correction of the relative-addresses may result in an alignment again because the change may affect the phase of

Algorithm 1 Pseudocode to Correct Relative-Addresses

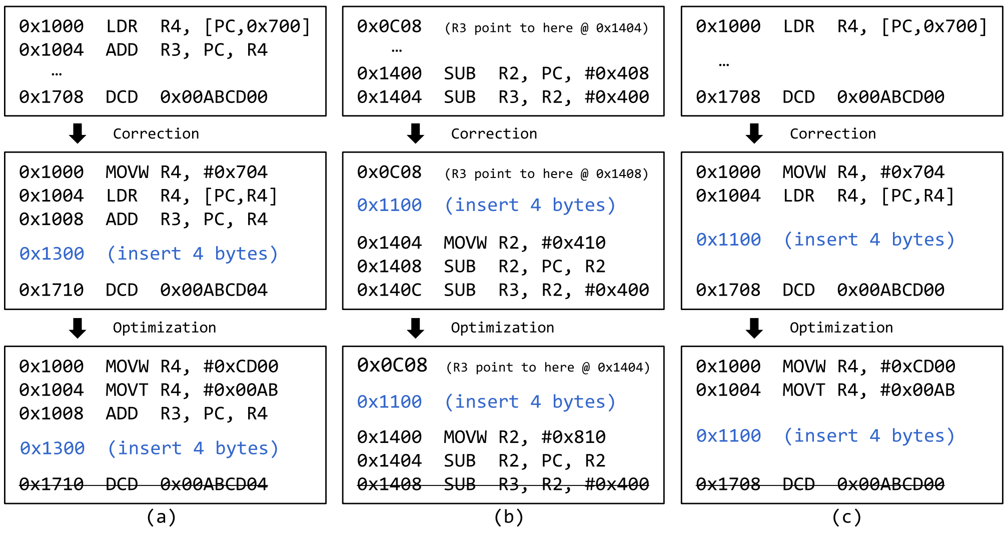
```
1:  procedure REARRANGE_BINARY($whole\_insts$, $state\_map$, $spec$)
2:      for $i \in whole\_insts$ do
3:          if has_relative_range($i$) then
4:              $base \leftarrow$ eval($state\_map$.get_base_value($i$))          ▷ Base address is always one
5:              for $v \in state\_map$.get_offset_values($i$) do
6:                  $target \leftarrow base +$ eval($v$)
7:                  $symbol \leftarrow$ make_symbol($v$)
8:                  $symbol\_to\_range[symbol] \leftarrow (base, target)$
9:                  $symbol\_to\_extension\_phase[symbol] \leftarrow 0$          ▷ Constant 0 means initial phase
10:                 $worklist \leftarrow symbol::worklist$          ▷ Add the symbol to worklist
11:             end for
12:         end if
13:     end for
14:
15:     while $worklist \neq []$ do          ▷ Iterate until worklist is empty
16:         $symbol \leftarrow worklist$.head()          ▷ Get a symbol from head of worklist
17:         $worklist \leftarrow worklist$.tail()          ▷ Remove head of worklist
18:         $range \leftarrow symbol\_to\_range[symbol]$
19:         $cur\_stretched\_size \leftarrow spec$.calc_stretched_size_within($range$)
20:         $cur\_phase \leftarrow symbol\_to\_extension\_phase[symbol]$
21:         $new\_phase \leftarrow spec$.correct_offset_at($symbol$, $cur\_phase$, $cur\_stretched\_size$)
22:
23:         if $cur\_phase < new\_phase$ then          ▷ Check the phase change
24:             $symbol\_to\_extension\_phase[symbol] \leftarrow new\_phase$
25:             for $symbol\_affected \in$ get_symbol_affected_from($symbol$) do
26:                 $worklist \leftarrow symbol\_affected::worklist$          ▷ Add the symbol affected to worklist
27:             end for
28:         end if
29:     end while
30:
31:     align_literal_pool()
32:     correct_elf_meta_info()
33: end procedure
34:
35: procedure MAKE_SYMBOL($value$)
36:     if type($value$) = LDR then
37:         return (CONST, $value$.get_reference())
38:     else if type($value$) = MOVT then
39:         return (MOVT, $value$.get_root_address(), $value$.get_child_address())
40:     else
41:         return (INST, $value$.get_root_address())
42:     end if
43: end procedure
```

the instruction extension. To solve this problem, we present two methods.

First, we set the amount of change that occurs in the relative-address correction and the size of each instrumentation code to a multiple of 4 bytes. The ARM instructions and most of the data in the literal pools require a 2-byte or 4-byte alignment. Processing each alignment individually during the correction process is challenging. Therefore, in order to avoid the 2-byte and 4-byte re-alignment, we handle instrumentations and corrections with a multiple of 4 bytes. Consequently, neither of these types of re-alignment needs to be considered.

Second, if the literal pool contains data that requires an 8-byte or 16-byte alignment,[12] we will insert `NOP` instructions at the beginning of the literal pool, which is similar to the way used by a modern compiler. After the disassembly

[12] Some SIMD & FP instructions access an 8-byte or 16-byte aligned data.



```
(a)
0x1000  LDR   R4, [PC,0x700]
0x1004  ADD   R3, PC, R4
  …
0x1708  DCD   0x00ABCD00
        ⬇ Correction
0x1000  MOVW R4, #0x704
0x1004  LDR   R4, [PC,R4]
0x1008  ADD   R3, PC, R4
0x1300  (insert 4 bytes)
0x1710  DCD   0x00ABCD04
        ⬇ Optimization
0x1000  MOVW R4, #0xCD00
0x1004  MOVT R4, #0x00AB
0x1008  ADD   R3, PC, R4
0x1300  (insert 4 bytes)
~~0x1710  DCD   0x00ABCD04~~
```

```
(b)
0x0C08  (R3 point to here @ 0x1404)
  …
0x1400  SUB   R2, PC, #0x408
0x1404  SUB   R3, R2, #0x400
        ⬇ Correction
0x0C08  (R3 point to here @ 0x1408)
0x1100  (insert 4 bytes)
0x1404  MOVW R2, #0x410
0x1408  SUB   R2, PC, R2
0x140C  SUB   R3, R2, #0x400
        ⬇ Optimization
0x0C08  (R3 point to here @ 0x1404)
0x1100  (insert 4 bytes)
0x1400  MOVW R2, #0x810
0x1404  SUB   R2, PC, R2
~~0x1408  SUB   R3, R2, #0x400~~
```

```
(c)
0x1000  LDR   R4, [PC,0x700]
  …
0x1708  DCD   0x00ABCD00
        ⬇ Correction
0x1000  MOVW R4, #0x704
0x1004  LDR   R4, [PC,R4]
0x1100  (insert 4 bytes)
0x1708  DCD   0x00ABCD00
        ⬇ Optimization
0x1000  MOVW R4, #0xCD00
0x1004  MOVT R4, #0x00AB
0x1100  (insert 4 bytes)
~~0x1708  DCD   0x00ABCD00~~
```

FIGURE 9. Examples of corrected instruction and optimized instruction.

is completed, we find the corresponding literal pool and insert padding at the start point of the literal pool as much as the maximum size of the alignment byte required. The alignment of each section is also padded as such. After the relative-address correction is completed, the remaining literal pool is aligned by removing a portion from the pre-padded one (`#31, align_literal_pool` in Algorithm 1). It finally recorrects the affected *relative ranges* and completes the alignment process. In some cases of section alignment, the alignment is performed by modifying the meta information of the ELF file instead of inserting the padding. As a result, it is possible to avoid increasing the size of the binary unnecessarily.

E. OPTIMIZATION

1) CORRECTION OVERHEAD

Our correction algorithm does not consider the semantic information between the instructions because it focuses on correcting relative-addresses individually. Therefore, it incurs a greater overhead than the operation or memory space required for the actual execution. Figure 9 shows the related examples. In the case (a), two instructions and 4 bytes of memory space are used to create a single address. Subsequent to the correction, the `MOVW` is added such that a total of three instructions and 4 bytes of memory space are used. Rather than obtaining the relative-address from the literal pool, reorganizing these instructions to acquire the relative-address via `MOVW` and `MOVT` is more efficient. In the case (b), a single address is computed by using two `SUB` instructions. Because the two `SUB` instructions are processed individually, the `MOVW` instruction is additionally used to correct the `SUB` at `0×1004` although it is possible to correct it by using two `SUB` instructions or a pair consisting of `SUB` and `MOV`. In the case (c), a 4-byte constant value is read by using `LDR`. In this case, the `MOVW` is additionally used to access the distant literal pool although memory space can be saved by using `MOVT` and `MOVW`. As such, it is possible to reduce the overhead incurred in the correction process by reconstructing the related instructions and data.

2) OPTIMIZATION PROCESS

To reconstruct the related instructions and data, when disassembly is completed, we find the three cases shown in Figure 9, based on the result of the value-set analysis. First, we search the `LDR` instructions that loads the constant into the literal pool and arithmetic instructions such as `ADD` and `SUB` that are *PC-relative instructions*. Second, in the case of the `LDR` instruction, we identify the type of the constant depending on whether it is computed with a *PC-based address*. Finally, we group the related instructions and data using the symbol information and some patterns. However, our proposed technique is neither designed to process cases in which the constant or relative-address of a literal pool is accessed at multiple points nor to process cases in which more than one address is computed at one point.

Our optimization process is performed after all corrections are completed. In the first step, we verified how the correction was performed at the previous identified group of the instructions and data, and subsequently verified whether it can be optimized by using some patterns. If this is possible, we reconstruct the bundle in an optimized manner. In the next step, we recorrect the *relative ranges* within which the modification points fall as a result of optimization. The optimization process proceeds similarly as the correction process except that the relative-addresses are decremented. Although this optimization does not apply to every case, most cases can be optimized.

3) OPTIMIZATION CONSTRAINT

Depending on the design of the instruction extension, cases occur in which the optimization is difficult. The optimization is not problematic if the range of expressible relative-addresses of every instruction extension phase is continuous. However, the optimization is difficult to perform even if the range of one phase is discontinuous. In the first example of Figure 4, the range of expressible relative-address of the second phase is discontinuous, and the range is as follows:

$$ROR(imm8, m) + ROR(imm8, n) \quad where \quad m, n \in \{0, 2, \ldots, 28, 30\}$$

Suppose the relative-address `0x10300(=0b100000011 00000000)` is corrected as the second phase in the correction process mentioned above. In the optimization process, if 10 bytes are optimized within the *relative range* corresponding to the relative-address, the relative-address `0x10300` is changed to `0x102F6(=0b100000010 11110110)`. However, the relative-address `0x102F6` cannot be expressed by the second phase, and can only be expressed by the third phase. If the phase is increased, it is difficult to guarantee the termination of the optimization for the reasons similar to those of the *correction cyclic problem*. Therefore, the optimization process is only applicable if all instruction extensions are designed to possess continuity.

F. BINARY REWRITER

The binary rewriting process receives the original binary and revision specifications as the input, as shown in Algorithm 2. The revision specification includes both the instrumentation specification and the relative-address correction information. This specification consists of the file offset, byte size to be skipped, and data to be written, as shown in Figure 8. Rewriting is achieved by taking the unchanged part from the original binary and taking the instrumented or corrected part from the revision specification. At the end of this process, the binary applied with the instrumentation code is generated.

VI. EVALUATION

This section describes the dataset, evaluates the two components of REPICA, and exhibit the effectiveness of REPICA by implementing a shadow stack on it.

A. DATASET

For a more accurate evaluation, we used two sets of data for a separate purpose. The first dataset is the 1,217 shared libraries and 106 executable files included in the Galaxy S8 (SM-G955N) with Android 8.1 (R16NW.G955NKSU1-CRD7). We selected this dataset because of two features. First, the code in this dataset is PIC in the ELF format. Second, these binaries are used for various purposes in the real world, and the pattern of the compiled code varies; therefore, we can encompass a wide range of code.

The second dataset contains 12 executable files of SPECint2006. Compared to the previous one, this dataset is suitable for performance measurements. The first one is disadvantageous in that it is difficult to properly run or measure performance, compared to the advantage that it can be used to test various patterns. Therefore, we evaluated the runtime overhead and space overhead with the second dataset. In addition, for the purpose of evaluation on various compilers, we compiled this dataset through `GCC 4.8` (first dataset is compiled with `Clang 3.8`)

B. DISASSEMBLY CORRECTNESS

Our rewriting technique requires the accurate identification between the instruction area and the data area in text segments. We consider the result of disassembly as correct when the result fully covers the text segments. Therefore, we evaluate whether the instruction area identified via disassembly and the data area dereferenced via the instructions (e.g., `LDR`) fully cover the text segments. In this subsection, we experiment the correctness of our disassembly module with the first dataset.

Our disassembly technique uses recursive traversal only; hence, it cannot identify the area in which the `NOP` instructions, which is used as padding, are placed before or after the literal pools, neither can it identify areas occupied by data chunks such as arrays or strings. Therefore, these areas are checked by manual inspection instead via commercial binary analysis tools such as IDA Pro. Except these cases, if there exist other unidentified areas, the disassembly is considered as failed. Moreover, we verified whether the instruction area overlapped with the area dereferenced via the `LDR` instruction. If an overlapped area exists, the result of the disassembly is considered to be incorrect. This is because such area may exist when the no-return function is not handled by our disassembly module.

Table 2 shows the experimental results for 1,319 binaries in the first dataset. According to the experimental results, the 121 binaries were not with completely covered. The non-full coverage is attributed to two primary reasons: First, the most typical case is the unreachable unknown areas such as the dead code and the auto-generated code. We start disassembly only using recursive traversal at the start address of the program, the exported function, the address of landing pad in the exception handling table, and other addresses in the dynamic section. If there exists an unidentified area that cannot be reached from these addresses, the result of disassembly will be considered as failed. Second, the case is that mis-disassembly occurred owing to the no-return function. Some functions such as `abort`, `exit`, `stack_chk_fail`, and `_Unwind_Resume` do not return. If these functions cannot be processed properly, overlapped areas may be found. We handle functions specified in the C++ ABI (application binary interface) and the LSB (linux standard base) [40] through the relocation information and heuristics. Except

TABLE 2. Rate of full coverage of disassembly by REPICA.

Path	/system/lib*/*		/system/bin/*		Total	Proportion
Type	AArch32	AArch64	AArch32	AArch64		
Succ	590	540	22	65	1,217	92.26%
Fail	56	42	1	3	102	7.74%
Total	646	582	23	68	1,319	100.00%

Algorithm 2 Pseudocode to Rewrite the Target Binary

```
1:  procedure REWRITE_BINARY($ob$, $spec$)            ▷ $ob$ = original binary, $spec$ = revision specification
2:      $nb \leftarrow$ create_file()
3:      $cur\_offset \leftarrow 0$
4:      for $offset\_to\_modify \in spec$.get_list_of_offset_to_modify() do      ▷ Sorted by ascending order
5:          $new\_data \leftarrow spec$.get_new_data($offset\_to\_modify$)
6:          $skip\_to\_size \leftarrow spec$.get_skip_to_size($offset\_to\_modify$)
7:          $nb$.write($ob$[$cur\_offset$:$offset\_to\_modify$])      ▷ Copy original binary to new binary
8:          $nb$.write($new\_data$)                                   ▷ Copy new data to new binary
9:          $cur\_offset \leftarrow offset\_to\_modify + skip\_to\_size$
10:     end for
11:     $nb$.write($ob$[$cur\_offset$:EOF])                  ▷ Copy remaining original binary to new binary
12:     return $nb$
13: end procedure
```

TABLE 3. Correctness of relative-address correction in REPICA.

File type		# of file	File size	# of inst.	# of relative range	# of error
Shared Lib.	AArch32	590	148,629 KB	13,263,638	4,564,534	0
	AArch64	540	156,036 KB	14,697,273	4,251,620	0
Excutable	AArch32	22	1,014 KB	128,947	53,377	0
	AArch64	65	7,988 KB	1,143,589	331,157	0
Total		1220	313,667 KB	29,233,447	9,200,688	0

these functions, if other functions exist, which no-return function places in the end, our disassembly fails.

C. CORRECTION CORRECTNESS

To evaluate the accuracy of the relative-address correction, we inserted one NOP instruction between all the instructions. That is, we verified whether relative-address correction can be processed at all the possible locations that can be instrumented. To validate the result, we re-analyzed the instrumented binaries by using our disassembly module to derive new *relative ranges*. Based on the original *relative range*, we verified whether each relative range has been corrected to the right width, and whether the instruction or data at the target address is identical. This experimentation was performed with 1,217 binaries that were disassembled with the full coverage mentioned above; the experimental results are as shown in Table 3.

We verified whether the instruction or data, which is pointed by the target address of the *relative range*, is same with original one. In the case where the instructions or data at the target address are modified during the correction process, the comparison is processed by the correction information. For the convenience of comparison, we do not apply the optimization to the target binary. Our disassembly detects the NOP instruction for various purposes, such as verifying for no-return functions; therefore, inserting the NOP instruction at all points can lead to incorrect results. Therefore, we inserted an alternative instruction such as MOV R2,R2 instead of NOP into some points.

Finally, to verify whether the instrumented binaries are effective, we selected and executed the `gzip`, `ls`, `grep`, `echo` and `cat` applications among the GNU core utilities. We performed all the command options to include all paths as much as possible. All the applications above executed without problems.

D. PERFORMANCE

To evaluate the performance overhead of REPICA, we tested the second dataset SPECint2006 that contains 12 benchmarks purposed for the CPU integer processing power. Because these benchmarks are not specifically designed for Android, when compiling the experimentation target, some library files are required that the basic Android NDK does not provide. Therefore, we compiled the benchmarks in SPECint2006 with the GCC compiler in an expanded NDK named `Crystax NDK 10.3.2`, which provide more libraries. Such that the benchmarks can operate on an Android device, the necessary compile option we used is as follows:

```
-march=armv7 -O2 -fPIE -pie
```

In this subsection, the results of all the experiments are performed 10 times on the NEXUS 6P with Android 8.1.

1) ENTIRE-INSTRUMENTATION

To evaluate the lower bound of the overhead when instrumentation occurred at all the instrumentable points, we inserted one NOP instruction between all the instructions except the jump instruction in the jump table and instructions in

TABLE 4. Runtime and space overhead for the benchmarks with entire-instrumentation (RR: relative range, EI: extended instructions, OH: overhead.)

Binary	.text size	# of inst.	# of RR	# of `NOP` (A)	# of EI (B)	(A+B)	Runtime OH	Space OH
400.perlbench	973,180	231,175	86,059	226,894	2,171	229,065	6.74%	0.89%
401.bzip2	63,360	15,681	3,666	15,510	27	15,537	54.70%	0.17%
403.gcc	2,814,664	613,114	253,215	619,804	5,122	624,926	57.12%	0.73%
429.mcf	10,412	2,596	936	2,538	0	2,538	101.60%	0.00%
445.gobmk	739,492	176,457	125,422	174,999	1,077	176,076	39.97%	0.58%
456.hmmer	250,244	61,004	19,366	59,721	361	60,082	83.71%	0.58%
458.sjeng	136,560	32,716	11,202	32,117	612	32,729	46.70%	1.79%
462.libquantum	41,912	10,459	3,361	10,291	6	10,297	95.09%	0.06%
464.h264ref	529,536	129,416	25,885	128,709	1,143	129,852	75.90%	0.86%
471.omnetpp	710,976	206,322	106,267	178,238	4,433	182,671	47.86%	2.49%
473.astar	116,560	28,979	9,754	27,867	74	27,941	35.88%	0.25%
483.xalancbmk	4,151,232	980,006	476,271	967,164	15,557	982,721	43.06%	1.50%

TABLE 5. Comparison of overhead for the benchmarks with entire-instrumentation and iCFT-instrumentation (EI: extended instructions, OH: overhead.)

Binary	Inst. type	# of `NOP` (A)	# of EI (B)	(A+B)	Runtime OH	Space OH
400.perlbench	entire-intr.	226,894	2,171	229,065	6.74%	0.89%
	iCFT-intr.	14,613	191	14,804	2.27%	0.08%
401.bzip2	entire-intr.	15,510	27	15,537	54.70%	0.17%
	iCFT-intr.	519	0	519	-0.26%	0.00%
403.gcc	entire-intr.	619,804	5,122	624,926	57.12%	0.73%
	iCFT-intr.	48,331	512	48,843	3.19%	0.07%
429.mcf	entire-intr.	2,538	0	2,538	101.60%	0.00%
	iCFT-intr.	89	0	89	-0.84%	0.00%
445.gobmk	entire-intr.	174,999	1,077	176,076	39.97%	0.58%
	iCFT-intr.	11,122	130	11,252	1.75%	0.07%
456.hmmer	entire-intr.	59,721	361	60,082	83.71%	0.58%
	iCFT-intr.	5,522	20	5,542	1.31%	0.03%
458.sjeng	entire-intr.	32,117	612	32,729	46.70%	1.79%
	iCFT-intr.	1,447	70	1,517	-0.38%	0.21%
462.libquantum	entire-intr.	10,291	6	10,297	95.09%	0.06%
	iCFT-intr.	895	0	895	-2.16%	0.00%
464.h264ref	entire-intr.	128,709	1,143	129,852	75.90%	0.86%
	iCFT-intr.	4,626	37	4,663	-0.71%	0.03%
471.omnetpp	entire-intr.	178,238	4,433	182,671	47.86%	2.49%
	iCFT-intr.	25,423	571	25,994	3.00%	0.32%
473.astar	entire-intr.	27,867	74	27,941	35.88%	0.25%
	iCFT-intr.	2,266	23	2,289	0.93%	0.08%
483.xalancbmk	entire-intr.	967,164	15,557	982,721	43.06%	1.50%
	iCFT-intr.	94,657	1,052	95,709	1.23%	0.10%

the PLT section. Generally, the runtime overhead is caused by two factors: overhead of instrumented instructions, and overhead of extended instructions generated because of the relative-address correction. To evaluate the overhead caused by extended instructions rather than the overhead of the instrumented instructions accurately, we inserted the `NOP` instruction that owns the minimum overhead (we call this *entire-instrumentation*).

The result is shown in Table 4. Even though the number of extended instructions increases as the number of *relative range* increases, the runtime overhead is not proportional to the number of extended instructions. It is difficult to obtain a relationship between the runtime overhead and extended instructions, especially in the case of 429.mcf and 462.libquantum; the number of extended instructions is almost 0 while the runtime overhead is 101.6% and 95.09%, respectively. Moreover, the number of *relative ranges* between 400.perlbench and 471.omnetpp is close but the performance overhead gap is large. Hence, in our opinion, the most runtime overhead is caused by the instrumented `NOP` instructions.

Consequently, we found that approximately 60% of the runtime overhead is caused by the instrumented code on average. Compared to the number of total instructions, the number of extended instructions generated owing to relative-address correction is less than 1%. Moreover, in the case of space overhead, the average is 0.80%, in which the minimum is 0% and the maximum is 2.49%. Thus, the runtime and space overhead caused by the relative-address correction itself is sufficiently small to be ignored.

2) iCFT-INSTRUMENTATION

In general, binary instrumentation is not performed at all points between the instructions but near the point of the function entry, function call, jump, and return instruction. Hence, we need to evaluate the runtime and space overhead of the binaries instrumented for practical use, and select the CFI technique to evaluate. CFI is the typical technique that uses the binary instrumentation technique, in which the technique verification routine is inserted at the iCTF (indirect control flow transfer) to enforce the valid control flow. To evaluate the lower bound of the overhead caused by implementing the CFI, we inserted one `NOP` instruction at the general target points such as the indirect call, indirect jump, and return instruction (we call this *iCFT-instrumentation*). Table 5 shows the comparison of overhead between *entire-instrumentation* and *iCFT-instrumentation*.

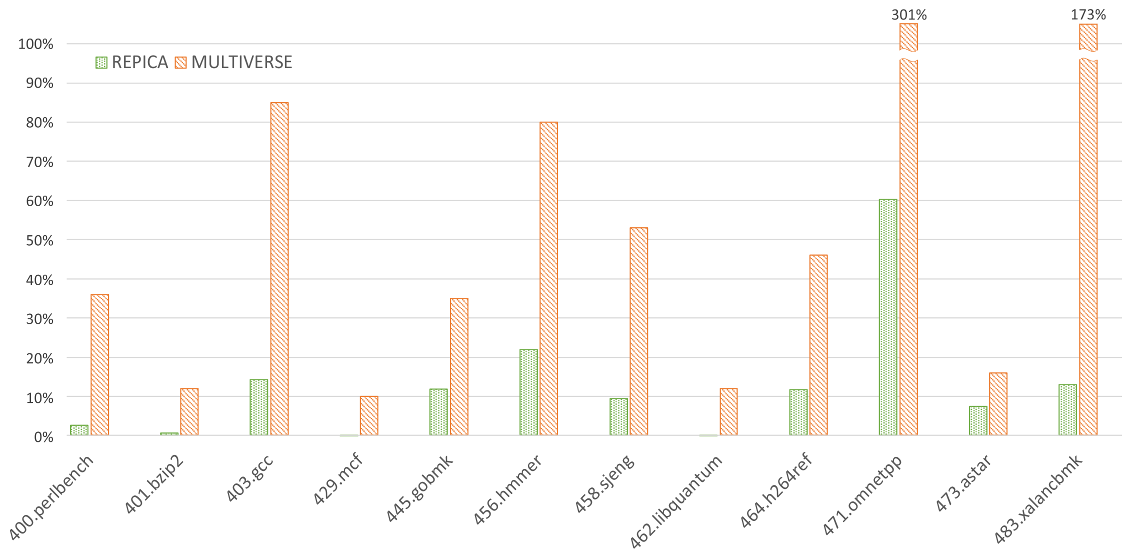


FIGURE 10. Percent runtime overhead for the benchmarks instrumented with a shadow stack.

Compared with the result of *entire-instrumentation*, the runtime and space overhead of *iCFT-instrumentation* are almost negligible. In the case of runtime overhead, the result range is from −2.16% to 3.19%. In particular, in the cases of 401.bzip2, 429.mcf, 458.sjeng, 462.libquantum, and 464.h264ref, the instrumented binaries execute slightly faster than the original binaries; we believe this is caused by the cache effect and experimental errors from the clock granularity. As for the space overhead, every benchmark is less than 0.32%. By comparing with the number of total instructions, the percentage of the extended instructions in each benchmark is less than 0.001%. In conclusion, REPICA can implement security applications such as CFI, shadow stack, and SFI (software-based fault isolation) with negligible runtime and space overhead.

E. CASE STUDY: SHADOW STACK

REPICA can be used for a variety of purposes, and we demonstrate its effectiveness through the implementation of a shadow stack. Shadow stack is a technique that protects the return address stored in the stack when calling a function. In this technique, whenever a function call occurs, it records the return address additionally in a separate memory space called *shadow stack*, and verifies or simply overwrites the return address using the corresponding one recorded on the *shadow stack* before return. This prevents the program control flow from moving to unintended addresses caused by attack techniques such as ROP attack and stack overflow. To show the effectiveness of REPICA through comparison, we implemented the overwriting, no-zeroing, parallel shadow stack [17] used in the Multiverse experimentation [32].

This technique is easy to implement because REPICA can insert any code at anywhere in the text segment. We inserted the code that saved the original return address to the shadow stack before each call instruction (i.e., BL, BLX). In ARM, two instruction sets exist, which are separated by the least significant bit; therefore, we added 1 when calculating the return address in the THUMB mode. The following is an example of the code we inserted:

```
STMFD     SP!, {R0,R1}
MOVW      R1,  #shadow_stack_offset
MOVT      R1,  #shadow_stack_offset
ADD       R0,  PC, #8 // +1 in THUMB mode
STR       R0,  [SP,-R1]
LDMFD     SP!, {R0,R1}
```

To protect the return address, we inserted a code that overwrites the return address of the current stack with the return address stored in the shadow stack before each return instruction (i.e., LDMFD {...,PC},BX LR). According to the ABI for ARM, in general, R0−R3 registers are not guaranteed to be preserved during function call, therefore, the R2 and R3 registers can be used without being stored; R0 and R1 register are used to store the return value. However, for the simplicity of implementation, we implemented all the inserted codes to temporarily store the R0 and R1 registers in the stack. In ARM, by default, the return address is stored in the LR register when the function is called, and the return value is stored in the stack depending on the situation. Therefore, two kinds of codes are inserted according to the situation, and the following is an example of the code we inserted:

```
<LDMFD {...,PC}>                  <BX LR>
STMFD  SP!, {R0,R1}               STMFD  SP!, {R0,R1}
MOVW   R0,  #shadow_stk_off       MOVW   R0,  #shadow_stk_off
MOVT   R0,  #shadow_stk_off       MOVT   R0,  #shadow_stk_off
LDR    R1,  [SP,-R0]              LDR    LR,  [SP,-R0]
STR    R1,  [SP,#8]               LDMFD  SP!, {R0,R1}
LDMFD  SP!, {R0,R1}
```

Figure 10 shows the runtime overhead of the SPECint2006 benchmark rewritten by REPICA for implementing the shadow stack. The highest runtime overhead is 60.33% for 471.omnetpp and the lowest runtime overhead is -0.26% for 429.mcf. In cases 429.mcf, 462.libquantum, and 401.bzip2, almost no overhead occurred; we believe this is because the majority of the performed code are simple computations. In contrast, 456.hmmer and 471.omnetpp exhibit the highest overhead. In particular, compared to other experiments [17], they exhibited a higher overhead than the previous experimental result. In our opinion, this is caused by a combination of architectures, operating systems, and binary instrumentation methods.

To compare the effectiveness of REPICA, we included the results of the Multiverse in Figure 10. Multiverse is a dynamic binary instrumentation technique that enables the PIC to be disassembled and rewritten without heuristics, and can be instrumented anywhere in the text segment. The reason we chose Multiverse in spite of it being a dynamic technique is that, similar with REPICA, it can almost perfectly process the PIC and can flexibly rewrite the text segment. Multiverse has inherent runtime overhead owing to the nature of dynamic techniques, and it demonstrates an average runtime overhead of 75% when the shadow stack is applied to the benchmarks. For our REPICA, the average runtime overhead is 12.74%, which is expected to improve slightly after some optimizations.

VII. LIMITATIONS

A. SELF-MODIFYING CODE

Our technique cannot handle self-modifying codes. The primary reason is that our technique can neither statically identify the code that is exposed at runtime, nor can it correctly handle overlapping codes. Next, our assumptions do not fit the self-modifying code. According to our assumption A2, one or more *code-addresses* cannot be transferred to a base register, but it can occur frequently in the self-modifying code. Therefore, a problem may occur in that not only the relative-addresses but also the base addresses need to be corrected. Generally, most programs do not contain self-modifying codes, and other static binary instrumentation techniques [4], [25], [26], [32], [33] exhibit the same limitation.

B. POSITION-DEPENDENT CODE

Our technique is specifically designed for the PIC; therefore, it is difficult to apply it to non-PIC using only our technique. Unlike the PIC, absolute addresses are placed in the data or text segments of the non-PIC. Hence, our technique cannot be immediately applied to some PIC in non-PIC binaries. In a recent research [25], [26], techniques for identifying pointer-like data have been developed. These techniques are primarily designed for handling non-PIC. With these techniques, our technique can be applied to process PIC included in non-PIC.

C. IMPERFECT DISASSEMBLY

The limitations of the disassembler prevent our technique from handling all typical binaries. It is unsolvable to completely disassemble any binaries statically [25], [26], [41] because many semantic information are erased during compilation [42]. Our technique can handle many cases by using heuristics, but our disassembly technique still cannot completely identify all indirect jumps, no-return function, and special code pattern resulted by the combination of optimization and unknown reasons. Some limitations are inherited from using only recursive disassembly. However, REPICA is a prototype, and can be improved through combining with various existing techniques [43], [44].

VIII. RELATED WORK

Early research on binary instrumentation was studied for non-security purposes such as optimization, performance measurement, and profiling. These techniques (e.g., PIXE [45], ATOM [46], QPT [47], and EEL [48]) are based on RISC architecture (e.g., SPARC and MIPS). Because the ARM architecture is also a RISC, these studies are highly relevant to our study. However, as the implementation details of those studies are not open, the exact way in which the PIC is processed remains unclear. We consider it highly likely that those studies used heuristics to process the PIC.

Since the late 1990s, many related studies have been conducted on ×86 architecture. In early studies on ×86, Etch [28] targets Win32/×86 binaries and is developed for performance measurement and optimization. SASI [49] targets ×86 binaries that are compiled with GCC, and is developed for the SFI. Because ×86 architecture uses a variable-length instruction, a 4-byte relative-address can be expressed by a single instruction. This means that the address computation in ×86 can be completed at one time rather than in multiple stages (as is the case in ARM). Therefore, it is easy for the existing ×86 binary instrumentation techniques to handle PIC by heuristics; however, for ARM, it is difficult to handle PIC only by these heuristics.

One of the problems in binary instrumentation is that it is difficult to identify pointer-like data in binary. To overcome this problem, many studies use the symbolic information generated from compilers [28], [50]–[54]. Because symbolic information includes the symbol type, the scope (block scope or global scope), the size, etc., these techniques can handle PIC properly. Other studies proposed runtime techniques to solve this problem [55]–[58]. A limitation of this approach is that instrumentation cannot be performed only with standalone binaries.

More closely related to our study, BISTRO [33] creates a space at the entry point of each function to insert the instrumentation code for the functions. This approach enables the instrumented code to be inserted more flexibly than techniques that simply replace the instructions and jump to the *trampoline*, such as Dyninst [27], [59] and Detours [29]. SecondWrite [60] is a binary writer that converts disassembled code into LLVM IR [61] for optimization without symbolic information; however, it cannot handle the PIC [62]. Other methods such as REINS [42] and BinCFI [4] are specific to the ×86 architecture only, and handle the ARM architecture with difficulty because of the different characteristics between ARM and ×86.

Recent works have led to the development of Uroboros [25] and Ramblr [26] to identify all pointer-like data in binary via a technique known as *symbolization*. Both are reassembly techniques, which can flexibly modify the instruction area. These techniques handle the PIC by identifying a specific function call that returns the next instruction address, and trace the return value to find the instruction containing the return value that is used to compute the address. This heuristic method can handle most cases of the ×86 architecture, but cannot properly handle the address computation patterns in ARM.

In the most recent study, RevARM [18] was developed as an insertion-based binary instrumentation technique. It can insert the instrumented code flexibly into any location on the text segment in binary. This technique, when handling the PIC, simply corrects the relative-addresses that are computed directly with the PC register. Therefore, the heuristic cannot handle the relative-addresses used by all address computations. To the best of our knowledge, no insertion-based binary instrumentation targeting ARM, other than that in this study, has been proposed.

Multiverse [32] is a binary rewriter based on shingled disassembly, and can handle PIC without a heuristic. When handling a PIC, it finds a specific function that returns the next instruction address, and translates it to an instruction that returns the old return address. This enables any subsequent address arithmetic to compute the old code addresses correctly. The rewriter dynamically remaps an old address to a new address, and can therefore safely handle the PIC. However, techniques such as this, because of the features of the approach they follow, incur large amounts of space overhead and runtime overhead.

To summarize, most existing ×86 techniques identify the call instruction that returns the address of the next instruction, after which they modify the instruction or modify an instruction that uses the return value. Similar to the ×86 techniques, the existing ARM techniques find instructions with the PC register as the operand, and trace the index value for correcting it. Most existing techniques for handling PIC consider only *PC-relative instructions*, which can handle many address computations that use relative-addresses. However, they cannot handle all patterns of address computations in ARM because they do not consider the relative-addresses in the *base-relative instructions*.

IX. CONCLUSIONS

PIC is becoming more important owing to the need for enhanced security. In particular, the use of PIC is increasing in ARM-based systems, which are gaining in importance owing to the expansion of the mobile and embedded markets. However, static binary instrumentation techniques on PIC are still immature. In addition, the characteristics of the ARM architecture present a new problem that does not occur with the ×86 architecture. With respect to rewriting the PIC composed by the ARM architecture instructions, we specified two primary challenges: (1) Difficulty in relative-address correction in all address computations, (2) Difficulty in efficient relative-address correction. In the second challenge, we found the *cyclic correction problem* and are the first to address this problem.

To overcome these challenges, we proposed REPICA, a static binary instrumentation technique capable of instrumenting a binary without symbolic information. Localized value-set analysis, of which the domain is in the form of symbols, was used to enable REPICA to accurately collect values and identify the points of correction. Furthermore, the use of a stepwise instruction extension allows REPICA to solve the *cyclic correction problem* effectively. We evaluated the correctness and performance of REPICA by experimenting with SPECint2006 and approximately 1,200 shared libraries and executables included in the deployed Android 8.1 build. Our experimental results showed that REPICA could rewrite all of the binaries that were disassembled completely, and that little space and runtime overhead was incurred by the newly rewritten binaries. Finally, our shadow stack implementation showed that REPICA can be used to solve various security problems.

REFERENCES

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proc. CCS*, Alexandria, VA, USA, 2005, pp. 340–353.
[2] J. Ansel *et al.*, "Language-independent sandboxing of just-in-time compilation and self-modifying code," in *Proc. PLDI*, San Jose, CA, USA, 2011, pp. 355–366.
[3] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: Software guards for system address spaces," in *Proc. OSDI*, Seattle, WA, USA, 2006, pp. 75–88.
[4] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proc. USENIX Secur.*, Austin, TX, USA, 2013, pp. 337–352.
[5] B. Niu and G. Tan, "Modular control-flow integrity," in *Proc. PLDI*, Edinburgh, U.K., 2014, pp. 577–587.
[6] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proc. SOSP*, Asheville, NC, USA, 1993, pp. 203–216.
[7] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe, "Efficient and language-independent mobile programs," in *Proc. PLDI*, Philadelphia, PA, USA, 1996, pp. 127–136.
[8] S. L. Graham, S. Lucco, and R. Wahbe, "Adaptable binary programs," in *Proc. USENIX Secur.*, Salt Lake City, UT, USA, 1995, pp. 1–14.
[9] B. Ford and R. Cox, "Vx32: Lightweight user-level sandboxing on the x86," in *Proc. USENIX ATC*, Boston, MA, USA, 2008, pp. 293–306.
[10] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. CCS*, Washington, DC, USA, 2003, pp. 290–299.
[11] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," in *Proc. USENIX Secur.*, Boston, MA, USA, 2007, pp. 275–290.
[12] H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P. Yew, "Control flow obfuscation with information flow tracking," in *Proc. MICRO*, New York, NY, USA, 2009, pp. 391–400.
[13] T.-C. Chiueh and F.-H. Hsu, "RAD: A compile-time solution to buffer overflow attacks," in *Proc. ICDCS*, Phoenix, AZ, USA, 2001, p. 0409.
[14] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Proc. SP*, San Francisco, CA, USA, 2013, pp. 48–62.
[15] M. Prasad and T.-C. Chiueh, "A binary rewriting defense against stack based buffer overflow attacks," in *Proc. USENIX ATC*, San Antonio, TX, USA, 2003, pp. 211–224.
[16] A. Baratloo, N. Singh, and T. Tsai, "Transparent run-time defense against stack smashing attacks," in *Proc. ATEC*, San Diego, CA, USA, 2000, pp. 1–13.

[17] T. H. Y. Dang, P. Maniatis, and D. Wagner, "The performance cost of shadow stacks and stack canaries," in *Proc. ASIA CCS*, Singapore, 2015, pp. 555–566.

[18] T. Kim *et al.*, "RevARM: A platform-agnostic arm binary rewriter for security applications," in *Proc. ACSAC*, San Juan, PR, USA, 2017, pp. 412–424.

[19] *PaX Address Space Layout Randomization (ASLR)*. Accessed: Jul. 15, 2018. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt

[20] *Android: Security Enhancements in Android 5.0*. Accessed: Jul. 15, 2018. [Online]. Available: https://source.android.com/security/enhancements/enhancements50

[21] *iOS Developer Library*. Accessed: Jul. 15, 2018. [Online]. Available: https://developer.apple.com/library/archive/qa/qa1788_index.html

[22] J. Drake, "Stagefright: Scary code in the heart of Android," presented at BlackHat, Las Vegas, NV, USA, 2015.

[23] *Android Keystore Stack Buffer Overflow*. Accessed: Jul. 15, 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3100

[24] *Android DRM Services—Buffer Overflow*. Accessed: Jul. 15, 2018. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-13253

[25] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *Proc. USENIX Secur.*, Washington, DC, USA, 2015, pp. 627–642.

[26] R. Wang *et al.*, "Ramblr: Making reassembly great again," in *Proc. NDSS*, San Diego, CA, USA, 2017, pp. 1–15.

[27] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 4, pp. 317–329, 2000.

[28] T. Romer *et al.*, "Instrumentation and optimization of Win32/Intel executables using Etch," in *Proc. USENIX Windows NT Workshop*, Seattle, WA, USA, 1997, pp. 1–8.

[29] G. Hunt and D. Brubacher, "Detours: Binaryinterception ofwin 3 2 functions," in *Proc. 3rd USENIX Windows NT Symp.*, Seattle, WA, USA, 1999, pp. 1–10.

[30] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proc. CCS*, Raleigh, NC, USA, 2012, pp. 157–168.

[31] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," in *Proc. ISPASS*, White Plains, NY, USA, 2010, pp. 175–183.

[32] E. Bauman *et al.*, "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *Proc. NDSS*, San Diego, CA, USA, 2018, pp. 40–47.

[33] Z. Deng, X. Zhang, and D. Xu, "BISTRO: Binary component extraction and embedding for software security applications," in *Proc. ESORICS*, Egham, U.K., 2013, pp. 200–218.

[34] M. Weiser, "Program slicing," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 352–357, Jul. 1984.

[35] G. Balakrishnan and T. Reps, "Analyzing memory accesses in x86 executables," in *Proc. CC*, Grenoble, France, 2014, pp. 5–23.

[36] P. Cousot and R. Cousot, "Abstract interpretation frameworks," *J. Logic Comput.*, vol. 2, no. 4, pp. 511–547, 1992.

[37] P. Cousot and R. Cousot, "Comparing the Galois connection and widening/narrowing approaches to abstract interpretation," in *Proc. PLILP*, Leuven, Belgium, 1992, pp. 269–295.

[38] *Capstone: The Ultimate Disassembler*. Accessed: Jul. 15, 2018. [Online]. Available: http://www.capstone-engine.org

[39] *The DWARF Debugging Standard*. Accessed: Jul. 15, 2018. [Online]. Available: http://press-pubs.uchicago.edu/founders/

[40] *Linux Foundation: Referenced Specifications*. Accessed: Jul. 15, 2018. [Online]. Available: https://refspecs.linuxfoundation.org

[41] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, "An in-depth analysis of disassembly on full-scale x86/x64 binaries," in *Proc. USENIX Secur.*, Austin, TX, USA, 2016, pp. 1–19.

[42] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Securing untrusted code via compiler-agnostic binary rewriting," in *Proc. ACSAC*, Orlando, FL, USA, 2012, pp. 299–308.

[43] D. Andriesse, A. Slowinska, and H. Bos, "Compiler-agnostic function detection in binaries," in *Proc. EuroS&P*, Paris, France, 2017, pp. 177–189.

[44] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in COTS binaries," in *Proc. DSN*, Denver, CO, USA, 2017, pp. 201–212.

[45] F. C. Chow, M. I. Himelstein, E. Killian, and L. Weber, "Engineering a RISC compiler system," in *Proc. COMPCON*, San Francisco, CA, USA, 1986, pp. 132–137.

[46] A. Srivastava and A. Eustace, "ATOM: A system for building customized program analysis tools," in *Proc. PLDI*, Orlando, FL, USA, 1994, pp. 196–205.

[47] J. R. Larus and T. Ball, "Rewriting executable files to measure program behavior," *Softw., Pract. Exper.*, vol. 24, no. 2, pp. 197–218, 1994.

[48] J. R. Larus and E. Schnarr, "EEL: Machine-independent executable editing," in *Proc. PLDI* La Jolla, CA, USA, 1995, pp. 291–300.

[49] U. Erlingsson and F. B. Schneider, "SASI enforcement of security policies: A retrospective," in *Proc. NSPW*, Inglewood, ON, Canada, 1999, pp. 287–295.

[50] B. Schwarz, S. Debray, G. Andrews, and M. Legendre, "PLTO: A link-time optimizer for the Intel IA-32 architecture," in *Proc. WBT*, Barcelona, Spain, 2001, pp. 1–7.

[51] R. Muth, S. K. Debray, S. Watterson, and K. De Bosschere, "Alto: A link-time optimizer for the compaq alpha," *Softw. Pract. Exper.*, vol. 31, no. 1, pp. 67–101, 2011.

[52] A. Srivastava, A. Edwards, and H. Vo, "Vulcan: Binary transformation in a distributed environment," Microsoft Res., Redmond, WA, USA, Tech. Rep. MSR-TR-2001-50, Apr. 2001.

[53] B. De Sutter, B. De Bus, and K. De Bosschere, "Link-time binary rewriting techniques for program compaction," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 5, pp. 882–945, 2005.

[54] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *Proc. USENIX Secur.*, Vancouver, BC, Canada, 2006, Art. no. 15.

[55] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electron. Notes Theor. Comput. Sci.*, vol. 89, no. 2, pp. 44–66, 2003.

[56] D. L. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Massachusetts Inst. Technol., Cambridge, MA, USA, 2004.

[57] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. PLDI*, Chicago, IL, USA, 2005, pp. 190–200.

[58] F. Bellard, "QEMU, a fast and portable dynamic translator," in *Proc. ATEC*, Anaheim, CA, USA, 2005, pp. 1–6.

[59] A. R. Bernat, K. Roundy, and B. P. Miller, "Efficient, sensitivity resistant binary instrumentation," in *Proc. ISSTA*, Toronto, ON, Canada, 2011, pp. 89–99.

[60] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. D. Keromytis, "Retrofitting security in COTS software with binary rewriting," in *Proc. SEC*, Lucerne, Switzerland, 2011, pp. 154–172.

[61] K. Anand *et al.*, "A compiler-level intermediate representation based binary analysis and rewriting system," in *Proc. Eurosys*, Prague, Czech, 2013, pp. 295–308.

[62] M. Smithson, K. ElWazeer, K. Anand, A. Kotha, and R. Barua, "Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness," in *Proc. WCRE*, Koblenz, Germany, 2013, pp. 52–61.

DONGSOO HA was born in Pohang, Gyeongsang, South Korea. He received the B.S. degree in computer science and engineering from Hanyang University, South Korea, in 2010, where he is currently pursuing the Ph.D. degree in computer science and engineering. His research interests are program language, program analysis, binary analysis, automated vulnerability analysis and detection, binary obfuscation, and security issues in smart mobile.

WENHUI JIN was born in Hegang, Heilongjiang, China. He received the B.S. degree in software and engineering from Heilongjiang University, China, in 2013. He is currently pursuing the Ph.D. degree in computer science and engineering from Hanyang University, South Korea. His research interests are binary obfuscation, binary analysis with machine learning, malware analysis and detection, and security issues in mobile.

HEEKUCK OH received the B.S. degree in electronics engineering from Hanyang University, South Korea, in 1983, and the M.S. and Ph.D. degrees in computer science from Iowa State University, Ames, IA, USA, in 1989 and 1992, respectively. In 1994, he joined the Faculty of the Department of Computer Science and Engineering, Hanyang University, ERICA campus, South Korea, where he is currently a Professor. His current research interests include network and system security. He is a member of the Advisory Committee for Digital Investigation in Supreme Prosecutor's Office, South Korea, and the Advisory Committee on Government Policy under the Ministry of Government Administration and Home Affairs. He is also a President Emeritus of the Korea Institute of Information Security and Cryptology.