NORAX: Enabling Execute-Only Memory for COTS Binaries on AArch64

Yaohui Chen* Dongli Zhang* Ruowen Wang† Rui Qiao*
Ahmed M. Azab† Long Lu* Hayawardh Vijayakumar† Wenbo Shen†

* Stony Brook University † Samsung Research America

***Abstract*—Code reuse attacks exploiting memory disclosure vulnerabilities can bypass all deployed mitigations. One promising defense against this class of attacks is to enable execute-only memory (XOM) protection on top of fine-grained address space layout randomization (ASLR). However, recent works implementing XOM, despite their efficacy, only protect programs that have been (re)built with new compiler support, leaving commercial-off-the-shelf (COTS) binaries and source-unavailable programs unprotected.**

We present the design and implementation of NORAX, a practical system that retrofits XOM into stripped COTS binaries on AArch64 platforms. Unlike previous techniques, NORAX requires neither source code nor debugging symbols. NORAX statically transforms existing binaries so that during runtime their code sections can be loaded into XOM memory pages with embedded data relocated and data references properly updated. NORAX allows transformed binaries to leverage the new hardware-based XOM support—a feature widely available on AArch64 platforms (e.g., recent mobile devices) yet virtually unused due to the incompatibility of existing binaries. Furthermore, NORAX is designed to co-exist with other COTS binary hardening techniques, such as in-place randomization (IPR). We apply NORAX to the commonly used Android system binaries running on SAMSUNG Galaxy S6 and LG Nexus 5X devices. The results show that NORAX on average slows down the execution of transformed binaries by 1.18% and increases their memory footprint by 2.21%, suggesting NORAX is practical for real-world adoption.

I. Introduction

Modern commodity operating systems employ code integrity protection techniques, such as data execution prevention (DEP), to prevent traditional code injection attacks. Consequently, recent attacks [1], [2] increasingly leverage code-reuse techniques to gain control of vulnerable programs. In code reuse attacks, a target application's control flow is manipulated in a way that snippets of existing code (called gadgets) are chained and run to carry out malicious activities.

Knowledge of process memory layout is a key prerequisite for code-reuse attacks to succeed. Attackers need to know the exact binary instruction locations in memory to assemble the chain of gadgets. Commodity operating systems widely adopt address space layout randomization (ASLR), which loads code binaries at random memory locations unpredictable to attackers. Without knowing the locations of needed code or gadgets, attackers cannot build code-reuse chains.

However, *memory disclosure* attacks can use information leaks in programs to de-randomize code locations, thus defeating ASLR. Such attacks either read the program code (*direct* de-randomization) or read code pointers (*indirect* de-randomization). Given that deployed ASLR techniques randomize the load address of a large chunk of data or code, leaking a single code pointer or a small sequence of code allows attackers to identify the corresponding chunk, infer its base address, and calculate the addresses of gadgets contained in the chunk.

More sophisticated fine-grained ASLR techniques [3]–[7] aim at shuffling code blocks within the same module to make it more difficult for attackers to guess the location of binary instructions. Nevertheless, research by Snow et al. [1] proves that memory disclosure vulnerabilities can bypass the most sophisticated ASLR techniques.

Therefore, a robust and effective defense against code-reuse attacks should combine fine-grained ASLR with memory disclosure prevention. Some recent works proposed to prevent memory disclosures using compile-time techniques [8]–[10]. Despite their effectiveness, these solutions cannot cover COTS binaries that cannot be easily recompiled and redeployed. These binaries constitute a significant portion of real-world applications that need protection.

XnR [11] is a recent work that enables executable-only memory (XOM [12]), which prevents code in memory from being read as data, and in turn, blocks leaking of code locations. However, XnR implements XOM at the OS level via paging-based access control, which can cause high overhead. Moreover, XnR cannot directly protect COTS binaries that are not originally built to make use of this protection.

Other defenses against memory disclosure follow the idea of destructive code reads [13], [14]: code is destroyed upon being read, and therefore cannot be later executed as part of a code reuse exploit. Unfortunately, it has been shown that destructive code reads can be bypassed through code reloading [15]. In addition, such defenses are not suitable for Android, where all apps load system libraries at the same locations [16].

Therefore, a memory read in one app enables code reuse attacks in any other app.

In this work, we propose NORAX [1], which protects COTS binaries from code memory disclosure attacks. NORAX allow COTS binaries to be loaded in hardware-enforced XOM, a security feature supported by recent ARM CPUs (i.e., AArch64). Such CPUs are widely seen on today's mobile devices. Without NORAX, to use the XOM feature, binaries need to be (re)built with the necessary compiler support. This requirement stands in between the valuable security feature and a large number of COTS binaries (e.g., all Android system executables and libraries) that are already running on AArch64 CPUs but were not compiled with XOM support. NORAX removes this requirement. It automatically patches existing binaries and loads their code to XOM-enforced memory regions, without affecting binaries' normal execution. As a result, binaries without special (re)compilation can benefit from the hardware-backed XOM feature and be protected against code memory disclosure. Further, when used together with ASLR, NORAX enables robust mitigation against code reuse attacks for COTS binaries. It is worth noting that we use Android as the reference platform for building and evaluating NORAX. However, NORAX's approach and techniques are generally applicable to other AArch64 platforms.

NORAX consists of four major components: *NDisassembler, NPatcher, NLoader, and NMonitor*. The first two perform offline binary analysis and transformation. They convert COTS binaries built for AArch64 without XOM support into one whose code can be protected by XOM during runtime. The other two components provide supports for loading and monitoring the patched, XOM-enabled binaries during runtime. The design of NORAX tackles a fundamentally difficult problem: identifying data embedded in code segments, which are common in ARM binaries, and relocating such data elsewhere so that during runtime code memory pages can be made executable-only while allowing all embedded data to be readable.

As a evaluation, we apply NORAX to Android system binaries running on SAMSUNG Galaxy S6 and LG Nexus 5X devices. The results show that NORAX on average slows down the transformed binaries by 1.18% and increases their memory footprint by 2.21%, suggesting NORAX is practical for real-world adoption.

In summary, our work makes the following contributions:

- We discover and address the gap between the highly valuable XOM feature and existing binaries, which need but cannot use the feature without recompilation.
- We design and implement a comprehensive system that converts COTS binaries to be XOM-compatible without requiring source code or debugging symbols.
- We show that code-data separation problem, although undecidable in principle, is in practice achievable on AArch64 platforms using our novel embedded data detection algorithm.
- We perform rigorous and extensive evaluations with stripped system executables and libraries on Android and show that NORAX is practical, effective and efficient.

The rest of the paper is organized as follows: In § II we lay out the background for execute-only memory and explain the code-data separation challenges tackled by NORAX; In § III we derive the requirements for a practical solution and then present the design of our system; In § IV we discuss in details the system implementation and the optimization for our reference platform Android; We then examine the correctness of NORAX and evaluate its performance in § V. We contrast the related works in § VI and analyze the compatibility of NORAX with other COTS hardening techniques and its current limitations in § VII. We conclude the paper in § VIII.

II. BACKGROUND

NORAX makes use of the modern MMU support in AArch64 architecture to create execute-only memory, which is a hardware feature now widely available yet virtually unused due to compatibility issues. To bridge the gap, NORAX reconstructs COTS binaries running on commodity Android smartphones to enforce the $\mathbf{R} \oplus \mathbf{X}$ policy. In the rest of this section, we explain the necessary technical background and the challenges we face when building the system.

AArch64 eXecute-Only Memory (XOM) Support: AArch64 defines four Exception Levels, from EL0 to EL3. EL0 has the lowest execution privilege, usually runs normal user applications; EL1 is usually for hosting privileged systems, such as operating system kernel; EL2 is designed for hypervisor while EL3 is for secure monitor.

In order to enforce the instruction access permission for different Exception Levels, AArch64 leverages the Unprivileged eXecute Never (UXN) bit, Privileged eXecute-Never (PXN) bit and two AP (Access Permission) bits defined in the page table entry [17]. For the user space program code page, the UXN bit is set to *"0"*, which allows the code execution at EL0, while PXN is set to *"1"*, which disables the execution in EL1. With such UXN and PXN settings, the instruction access permissions defined by AP bits are shown in Table I. It is easy to see that we can set the AP bits in page table entry to *"10"*, so that the kernel running in EL1 will enforce the execute-only permission for user space program, which is running in EL0. In other words, the corresponding memory page will only permit for instruction fetch for user space program, while all read/write data accesses will be denied.

[1] NORAX stands for **NO R**ead **A**nd e**X**ecute.

TABLE I: Access permissions for stage 1 EL0 and EL1

AP[2:1]	EL0 Permission	EL1 Permission
00	Executable-only	Read/Write
01	Read/Write, Config-Executable	Read/Write
10	Executable-only	Read-only
11	Read, Executable	Read-only

However, the kernel still has the read permission to that page, which means that it can help the user space program read the intended memory area if necessary, but need to perform security checks beforehand.

Position-Independent Binaries in Android: Position-independent code (PIC) is the kind of code compiler generates for a module that does not assume any absolute address, that is, no matter where the module is loaded, it will be able to function correctly. The mechanism works by replacing all the memory accesses using hard-coded addresses with PC-relative addressing instructions. Position-independent executables (PIE) are executables that employ PIC code. In Android, ever since version 5 (codename: Lolipop), in order to fully enjoy the benefit of ASLR, all the executables are required to be compiled as PIE. To enforce this, Google removed the support for non-PIE loading from the Bionic Linker [18]. Nowadays, smartphones equipped with AArch64 CPU are most likely running Android OSes after Lolipop, meaning the majority of them will only have binaries, including both executables and shared libraries, that are compiled to be position independent.

Code-Data Separation: To convert a stripped binary to be XOM-compatible, there is one fundamental problem to solve, namely *code-data separation*. Note that separating data from code for COTS binaries is, in general, undecidable as it is equivalent to the famous *Halting Problem* [19]. But we found that in the scope of ARM64 position-independent binaries, which are prevalent in modern Android and iOS [20] Phones, a practical solution is possible. Basically, a feasible solution should address the two following challenges.

1) Locating Data In Code Pages: We generally refer to data residing in executable code regions as ***executable data***. There are *two types* of executable data allowed in ELF binaries.

- **Executable sections**: The first kind of data are those ELF sections consisting of pure read-only data which could reside in executable memory. Defined by contemporary ELF standard, a typical ELF file has two views: linking view and loading view, used by linker and loader respectively. Linking view consists of ELF sections (such as *.text, .rodata*). During linking, the static linker bundles those sections with compatible access permissions to form a segment – in this case, executable indicates readable. The segments then comprise the loading view. When an ELF is being loaded, the loader simply loads each of the segments as a whole into memory, and grant the corresponding access permissions. A standard ELF has two loadable segments. One is readable and executable, which is normally referred as "code segment". This segment contains all the sections with instructions (*.plt* and *.text*, etc.), and read-only data (*.gnu.hash, .dynsym, etc.*); the other segment is readable and writable, referred as "data segment", it contains the program data as well as other read/writ-able sections. For our goal to realize *non-readable code*, we mainly focus on the code segment. In this segment, generally only *.plt* and *.text* contain instructions used for program execution, but as explained before, they are mixed with other sections that only need to be read-only, thus we cannot simply map the memory page to execute-only as oftentimes these sections could locate within the same page. For instance, Table II shows the code segment layout of an example program, all except the last two sections in this code segment are placed within the same page. To make things more complex, the segment layout varies for different ELFs.
- **Embedded data:** The second kind of data in the code pages is those embedded data in the *.text* section. For optimization purpose, such as exploiting spatial locality, compilers emit data to places nearby their accessing code. Note that albeit recent study [21] shows that in modern x86 Linux, compilers no longer generate binaries that have code interleaved with data, to the opposite of our discovery, we found this is not the case for ARM, we examined the system binaries extracted from smartphone Nexus 5X running the factory image MMB29P, Table III reveals that code-data interleaving still prevails in those modern ARM64 Linux binaries, indicating this is a real-world problem to be solved.

TABLE II: ELF sections that comprise the code segment of the example program, the highlighted ones are locate in the same page.

Section Name	Address	Type
.interp	0000000000000238	PROGBITS
.note.android.ident	0000000000000250	NOTE
.note.gnu.build-id	0000000000000268	NOTE
.gnu.hash	0000000000000288	GNU_HASH
.dynsym	00000000000002c8	DYNSYM
.dynstr	00000000000005b0	STRTAB
.gnu.version	00000000000006e2	VERSYM
.gnu.version_r	0000000000000720	VERNEED
.rela.dyn	0000000000000740	RELA
.rela.plt	0000000000000830	RELA
.plt	00000000000009a0	PROGBITS
.text	0000000000000ab0	PROGBITS
.rodata	0000000000000f08	PROGBITS
.eh_frame_hdr	00000000000010d0	PROGBITS
.eh_frame	0000000000001110	PROGBITS

TABLE III: Android Marshmallow system binaries that have *embedded data* in Nexus 5X.

	# of binaries	# of binaries w/ embeded data	Percentage
/system/bin	237	167	70.46%
/system/lib64	255	101	39.61%
/vendor/lib64	111	39	35.14%
/vendor/bin	4	2	50.00%

2) Updating Data References: In addition to finding out the locations of executable data, we also need to relocate them and update their references. It turns out that references updating is also non-trivial. In our system, as shown in Table IV, the majority of the ELF sections inside code segment are expected to be relocated to a different memory location so that appropriate permission can be enforced. The sections that are left out, such as *.interp* and *.note.*$*$ are either accessed only by OS or not used for program execution so we can leave them untouched. For those sections listed in Table IV, they have complex interconnections, both internally and externally. As shown in Table V, various types of references exist in a given ELF. Due to this complexity, the references collection is conducted across the whole NORAX system by different components in different stages including both offline and during load-time.

TABLE IV: Sections in the executable code page that are handled by NORAX

(.gnu).hash	.dynsym	.dynstr	.gnu.version	.rela.dyn
.rela.plt	.text (embedded data)	.rodata	.eh_frame	.eh_frame_hdr

TABLE V: ELF section reference types

Reference Type	Example
Intra-section references	.text refers to .text (embedded data)
Inter-section references	.text refers to .rodata
External references	dynamic linker refers to .dynsym, .rela.$*$
Multiple external references	C++ runtime/debugger refer to .eh_frame

III. NORAX DESIGN

A. System Overview

The goal of NORAX is to allow COTS binaries to take advantage of execute-only memory (XOM), a new security feature that recent AArch64 CPUs provide and is widely available on today's mobile devices. While useful for preventing memory disclosure-based code reuse [1], [2], XOM remains barely used by user and system binaries due to its requirement for recompilation. NORAX removes this requirement by automatically patching COTS binaries and loading their code to XOM. As a result, existing binaries can benefit rom the hardware-backed protection against direct code memory disclosure attacks. While we demonstrate NORAX on Android, the ideas behind NORAX are generally applicable to any AAarch64 platform.

Design Principles: To make NORAX widely useful in practice, we set the following design principles for NORAX:

- **P1** - **Backward compatibility**: Changes introduced by NORAX to binaries must not break their standard structures or compilation conventions (i.e., patched binaries can run on devices without NORAX support). Otherwise, patched binaries may become incompatible with existing loaders, linkers, or orthogonal binary-hardening solutions (e.g., code diversification techniques). Furthermore, NORAX must not make special assumptions about binaries to facilitate analysis and patching.
- **P2** - **Completeness**: NORAX must have complete coverage of embedded data. It must detect all embedded data in a binary accessed by code and ensure that these accesses still succeed when XOM enforcement is in place. On the other hand, NORAX can only have very few, if not zero, false positives (i.e., misidentifying code as data).
- **P3** - **Correctness**: NORAX must not alter or break a patched binary's original function or behavior, needless to say crashing the binary.
- **P4** - **Low Overhead**: NORAX should not introduce impractical overheads to the patched binaries, including both space overhead (e.g., binary sizes and memory footprint) and runtime slowdown.

NORAX Workflow: NORAX consists of four major components: *NDisassembler, NPatcher, NLoader, and NMonitor*, as shown in Figure 1. The first two components perform offline binary analysis and transformation and the last two provide runtime support for loading and monitoring the patched, XOM-compatible executables and libraries. In addition to disassembling machine code, NDisassembler scans for all executable code that needs to be protected by XOM. A major challenge it solves is identifying various types of data that ARM compilers often embed in the code section, including jump tables, literals, and padding. Unlike typical disassemblers, NDisassembler has to precisely differentiate embedded data from code in order to achieve $P2$ and $P3$ (§III-B). Taking input from NDisassembler, NPatcher transforms the binary so that its embedded data are moved out of code sections and their references are collected for later adjustment. After the transformation, NPatcher inserts a unique magic number in the binary so that it can be recognized by NLoader during load-time. NPatcher also stores NORAX metadata in the binary, which will be used by NLoader and NMonitor (§III-C). When a patched binary is being loaded, NLoader takes over the loading process to (*i*) load the NORAX metadata into memory,

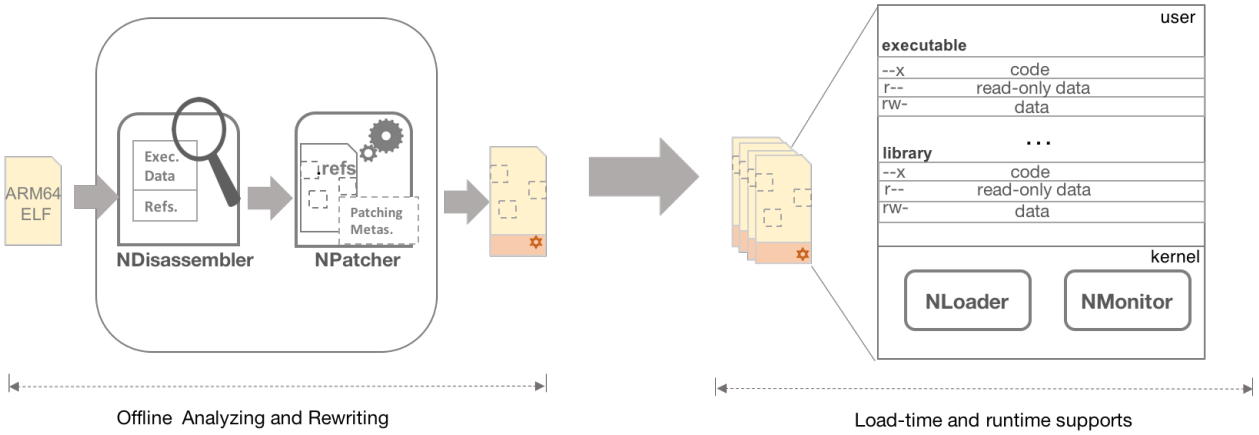

Fig. 1: NORAX System Overview: the offline tools (left) analyze the input binary, locate all the executable data and their references (when available), and then statically patch the metadata to the raw ELF; the runtime components (right) create separated mapping for the *executable data sections* and update the recorded references as well as those generated at runtime.

(*ii*) adjust the NPatcher-collected references as well as those dynamically created references to the linker-related sections (e.g .hash, .rela.*), and (*iii*) map all memory pages that contain code to XOM (§III-D). During runtime, NMonitor, an OS extension, handles read accesses to XOM. While such accesses are rare and may indicate attacks, they could also be legitimate because NPatcher may not be able to completely recognize dynamic references to the relocated embedded data (e.g., those generated at runtime). When there are missed data references, the access will trigger an XOM violation, which NMonitor verifies and, if legitimate, facilitates the access to the corresponding data (§III-E).

B. NDisassembler: Static Binary Analyzer

NDisassembler first converts an input binary from machine code to assembly code and then performs analysis needed for converting the binary into an XOM-compatible form. It disassembles the binary in a linear sweep fashion, which yields a larger code coverage than recursive disassembling [21]. However, the larger code coverage comes at a cost of potentially mis-detecting embedded data as code (e.g., when such data happen to appear as syntactically correct instructions).

NDisassembler addresses this problem via an iterative data recognition technique. Along with this process, it also finds instructions that reference embedded data. The data recognition technique is inspired by the following observations:

- Although it is difficult to find all instructions referencing some embedded data at a later point in the running program, it is relatively easy to locate the code that computes these references in the first place.
- To generate position-independent binaries, compilers can only use PC-relative addressing when emitting instructions that need to reference data inside binaries.
- AArch64 ISA only provides two classes of instructions for obtaining PC-relative values, namely the `ldr (literal)` instructions and `adr(p)` instructions.

NDisassembler uses Algorithm 1 to construct an initial set of embedded data (IS) and a set of reference sites (RS). For embedded data whose size cannot be precisely bounded, NDisassembler collects their seed addresses (AS) for further processing. As shown in Line 5–9 in Algorithm 1, since the load size for `ldr-literal` instructions is known, the identified embedded data are added to IS. On the other hand, the handling for `adr` instructions is more involved, as shown in Line 10–27. NDisassembler first performs forward slicing on `xn` — the register which holds the embedded data address. All instructions that have data dependencies on `xn` are sliced, and `xn` is considered *escaped* if any of its data-dependent registers is either (*i*) stored to memory or (*ii*) passed to another function before being killed. In either case, the slicing also stops. If not all memory dereferences based on `xn` can be identified due to reference escaping, the size of the embedded data cannot be determined. Therefore, NDisassembler only adds the initial value of `xn` to AS, as a seed address (Line 24–26).

Line 10–23 of Algorithm 1 deal with the sliced instructions. If a memory load based on `xn` is found, RS is updated with the location of the original address-taking instruction. Moreover, NDisassembler analyzes the address range for each memory load. Note that oftentimes the address range is bounded because embedded data are mostly integer/floating point constants, or jump tables. In the former case, the start address of

Algorithm 1 Initial embedded data and references collection

INPUT:
$code[]$ - An array of disassembly output
OUTPUT:
IS - Initial set of embedded data
AS - The set of seed addresses for embedded data
RS - The set of reference sites to embedded data

```
1:  procedure INITIALSETCOLLECTION
2:      IS = {}
3:      AS = {}
4:      RS = {}
5:      for each (ldr-literal addr) ∈ code[] at curr do
6:          size = MemLoadSize(ldr)
7:          IS = IS ∪ {addr, addr+1, ..., addr+size-1}
8:          RS = RS ∪ {curr}
9:      end for
10:     for each (adr xn, addr) ∈ code[] at curr do
11:         escaped, depInsts = ForwardSlicing (xn)
12:         unbounded = False
13:         for each inst ∈ depInsts do
14:             if inst is MemoryLoad then
15:                 RS = RS ∪ {curr}
16:                 addr_expr = MemLoadAddrExpr(inst)
17:                 if IsBounded(addr_expr) then
18:                     IS = IS ∪ {AddrRange(addr_expr)}
19:                 else
20:                     unbounded = True
21:                 end if
22:             end if
23:         end for
24:         if escaped or unbounded then
25:             AS = AS ∪ {addr}
26:         end if
27:     end for
28: end procedure
```

memory load is typically `xn` plus some constant offset, while the load size is explicit from the memory load instruction. In the latter, well-known techniques for determining jump table size [22] are utilized. In both cases, the identified embedded data are added into IS. However, if there is a single memory load whose address range cannot be bounded, NDisassembler adds the seed address to AS.

If Algorithm 1 is not able to determine the sizes of all embedded data, the initial set (IS) is not complete. In this case, the seed addresses in AS are expanded using Algorithm 2 to construct an over-approximated set of embedded data (DS). The core functions are $BackwardExpand$ (line 4) and $ForwardExpand$ (line 5). The backward expansion starts from a seed address and walks backward from that

Algorithm 2 embedded data set expansion

INPUT:
AS - The set of seed addresses for embedded data
IS - Initial set of embedded data
OUTPUT:
DS - conservative set of embedded data

```
1: procedure SETEXPANSION
2:     DS = IS
3:     for addr in AS do
4:         c1 = BackwardExpand (addr, DS)
5:         c2 = ForwardExpand (addr, DS)
6:         DS = DS ∪ c1 ∪ c2
7:     end for
8: end procedure
```

address until it encounters a *valid* control-flow transfer instruction: i.e., the instruction is either a *direct* control-flow transfer to a 4-byte aligned address in the address space, or an *indirect* control-flow transfer. All bytes walked through are marked as data and added to DS. On the other hand, the forward expansion walks forward from the seed address. It proceeds aggressively for a conservative inclusion of all embedded data. It only stops when it has strong indication that it has identified a valid code instruction. These indicators are one of the following: (*i*) a *valid* control-flow transfer instruction is encountered, (*ii*) a direct control-flow transfer target (originating from other locations) is reached, and (*iii*) an instruction is confirmed as the start of a function [23]. In the last case, comprehensive control-flow and data-flow properties such as parameter passing and callee saves are checked before validating an instruction as the start of a function.

Finally, DS contains nearly all embedded data that exists in the binary. Although we could further leverage heuristics to include *undecodable* instructions as embedded data, it is not necessary because our conservative algorithms already cover the vast majority (if not all) of them, and the rest are mostly padding bytes which are never referenced. Theoretically, failure to include certain *referenced* embedded data could still happen if a chunk of data can be coincidentally decoded as a sequence of instructions that satisfies many code properties, but in our evaluation of *over 300* stripped Android system binaries (V-A), we never encountered such a case.

RS contains a large subset of reference sites to the embedded data. Since statically identifying all indirect or dynamic data references may not always be possible, NDisassembler leaves such cases to be handled by NMonitor.

C. NPatcher: XOM Binary Patcher

With the input from NDisassembler, NPatcher transforms the binary in two steps. First, it relocates data out of the code

segment so that the code segment can be loaded to XOM and protected against leaks and abuses. Next, it collects and prepares the references from code (*.text*) to the embedded data (*.text*) and to *.rodata* section.

Data Relocation: An intuitive design choice is to move the executable data out of the code segment. But doing so violates the design principle $P1$ as the layout of the ELF and the offsets of its sections will change significantly. Another approach is to duplicate the executable data, but this would increase binary sizes and memory footprint significantly, violating $P4$.

Instead, NPatcher uses two different strategies to relocate those executable data without modifying code sections or duplicating all read-only data sections. For data located in code segment but are separated from code text (i.e., read-only data), NPatcher does not duplicate them in binaries but only records their offsets as metadata, which will be used by NLoader to map such data into read-only memory pages. For data mixed with code (i.e., embedded data), NPatcher copies them into a newly created data section at the end of the binary. The rationale behind the two strategies is that read-only data usually accounts for a large portion of the binary size and duplicating it in binary is wasteful and unnecessary. On the other hand, embedded data is usually of a small size, and duplicating it in binaries does not cost much space. More importantly, this is necessary for security reasons. Without duplication, code surrounding data would have to be made readable, which reduces the effectiveness of XOM.

Data Reference Collections: NPatcher only collects the references from *.text* to *.text* (embedded data) and to *.rodata* because they can be statically recognized and resolved. Other types of references listed in Table V are either from outside the module or statically unavailable, which are handled by NLoader.

For references to embedded data, NPatcher can directly include them based on NDisassembler's analysis results. But there is one caveat – the instructions used to reference embedded data (i.e., *adr* and *ldr-literal*) have a short addressing range. Therefore, when we map their target data to different memory pages, it is possible that the instructions cannot address or reach the relocated data. To solve this issue without breaking $P1$ (i.e., maintaining binary backward-compatibility), NPatcher generates stub code to facilitate access to out-of-range data. The instructions of short addressing range are replaced with an unconditional branch instruction[2], which points to the corresponding stub entry. The stub code only contains unconditional load and branch instructions pointing to fixed immediate offsets. This design ensures that these stub entries cannot be used as ROP gadgets.

For references to the *.rodata*, there is no addressing capability problem, because *adrp* is used instead of *adr*. However, a different issue arises. There are multiple sources from which such references could come. We identify 5 sources in our empirical study covering all Android system executables and libraries. NPatcher can only prepare the locations of the first three offline while leaving the last two to be handled by NLoader after relocations and symbol resolving are done.

- **References from code (*.text*)**: these are usually caused by access to constant values and strings.
- **References from symbol table (*.dynsym*)**: when a symbol is located in *.rodata*, there will be an entry in the symbol table, whose *value* field contains the address of the exposed symbol.
- **References from relocation table (*.rela.dyn*)**: for a relocatable symbol located in *.rodata*, the relocation table entry's *r_addend* field will point to the symbol's address.
- **References from global offset table (*.got*)**: when a variable in *.rodata* cannot be addressed due to the addressing limit(e.g., *adrp* can only address +/- 4GB), an entry in the global offset table is used to address that far-away variable.
- **References from read-only global data (*.data.rel.ro*)**: most binaries in Android disable lazy-binding. The *.data.rel.ro* section contains the addresses of global constant data that need to be relocatable. After the dynamic linker finishes relocating them, this table will be marked as read-only, as opposed to the traditional *.data* section.

Finally, the metadata (duplicates and references), the data-accessing stub code and the NORAX header are appended to the end of the original binary, as shown in Figure 2. Note that by appending the NORAX-related data to the end of the binary, we allow patched binaries to be backward-compatible, thus meeting $P1$. This is because the ELF standard ignores anything that comes after the section header table. As a result, binaries transformed by NPatcher can run on devices without NORAX support installed. They can also be parsed and disassembled by standard ELF utilities such as *readelf* and *objdump*. Moreover, NORAX-patched binaries are compatible with other binary-level security enhancement techniques.

D. NLoader: Plugin for Stock Loader and Linker

Binaries rewritten by NPatcher remain recognizable by and compatible with the stock loader and linker. They can still function albeit without the XOM protection. New data sections added by NORAX, however, are transparent to the toolchain. They require NLoader's support to complete the binary loading and references updating process before their code can be mapped in XOM. Other than the ones prepared by NPatcher,

[2]*ADR* can address +/- 1MB, while *B(ranch)* can access +/- 128MB, which is far enough for regular binaries.

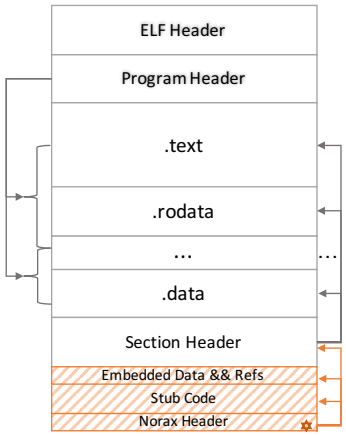

Fig. 2: The layout of ELF transformed by NORAX. The shaded parts at the end are the generated NORAX-related metadata.

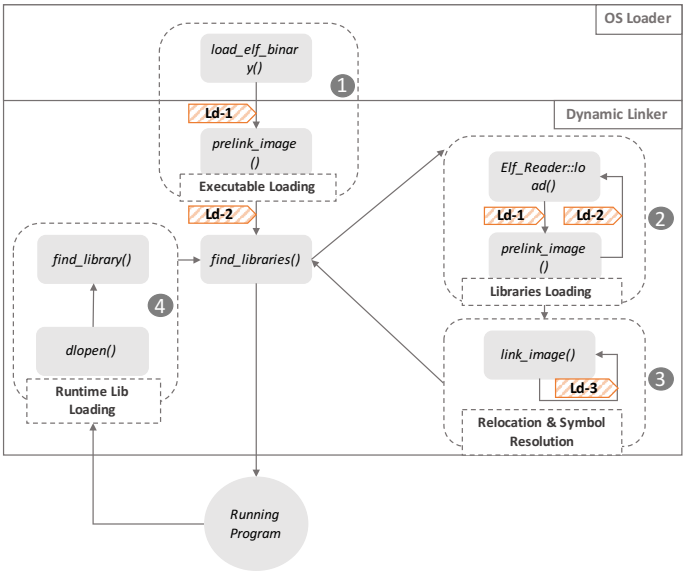

Fig. 3: Bionic Linker's binary loading flow, NLoader operates in different binary preparing stages, including module loading, relocation and symbol resolution.

as mentioned in § III-A, there are several types of references to executable data (Table V) which are related to the linker and only available at runtime. Built as a linker/loader plugin, NLoader adjusts these references in the following steps:

- **Ld-1:** It parses and loads NORAX header into memory, including information about the embedded data in *.text* and the stub code accessing embedded data. Then, it creates duplicated mappings for *.rodata* and the linker-referencing sections[3], which have been loaded by the stock linker/loader.
- **Ld-2:** It updates the *.dynamic* section to redirect linker to use the read-only copy of those relocated data sections.
- **Ld-3:** It collects the *.rodata* references from *.got* and *.data.rel.ro*, which are only populated after the relocation is done. It then adjusts all the collected data references in one pass. Eventually, the memory access level of the loaded module is adjusted to enforce the $\mathbf{R} \oplus \mathbf{X}$ policy.

The overall workflow of NLoader is shown in Figure 3. It starts with the executable loading, which is done by the OS ELF loader (Step ①). Then, the OS loader transfers the control to the dynamic linker, which in turns creates a book-keeping object for the just-loaded module. Meanwhile, **Ld-1** is performed to complete the binary loading. Next, the binary's corresponding book-keeping object is then populated with references to those ELF sections used by the linker to carry out relocation and symbol resolution in a later stage. **Ld-2** is then invoked to update these populated references. At this point, the preparation for the executable is done. The linker then starts preparing all the libraries (Step ②). This process is similar to the preparation of executable, thus **Ld-1** and **Ld-2** are called accordingly. When all the modules are loaded successfully in previous steps with their book-keeping objects populated, the linker walks through the book-keeping objects to perform relocation and symbol resolution (Step ③). In this step, **Ld-3** is called for each of the relocated modules to update all those collected references, including the ones from *.got* and *.data.rel.ro* to *.rodata*. This is feasible because the *.got* entries which reference to *.rodata* are populated upfront, same as those in *.data.rel.ro*.

During runtime, the program may dynamically load or unload new libraries (Step ④), as shown in Figure 3, which is also naturally handled by NLoader. To boost performance, once NLoader finishes updating the offline-updatable references, it caches the patched binary so that it can directly load the cached version without going through the whole references adjustment process again next time.

E. NMonitor: Runtime Enforcement and Safety-net

After being processed by the last three NORAX components, a patched binary that follows the $\mathbf{R} \oplus \mathbf{X}$ policy is ready to run, which is assisted by NMonitor. At runtime, the converted program could still be running with some unadjusted references to the executable data, which belong to the two following possible categories.

- **Missed references to embedded data**: Although in our evaluation we rarely see cases where an access violation is triggered by missed embedded data references, such situation, if mishandled, will cause a program crash. NDisassembler is unable to discover such cases due to the limitation of static analysis. These missed data references

[3]The linker-referencing sections include *.(gnu).hash, .dynsym, .dynstr, .gnu.version, .gnu.version_r, .rela.dyn, .rela.plt.*, etc.

would trigger access violations. Note that references to *.rodata* from *.text* do not have this problem, because whenever an address is calculated that happens to point at *.rodata* section, NDisassembler will mark it as a valid reference regardless of whether a corresponding memory load instruction is detected or not.

- **References to *.eh_frame_hdr* and *.eh_frame***: These sections provide auxiliary information such as the address range of functions, the stack content when a C++ exception is triggered, etc. The previous components are unable to update them because they are used neither by the converted module itself nor by the dynamic linker. Instead, we found that C++ runtime and debuggers such as gdb would reference and read into these two sections for exception handling or stack unwinding.

NMonitor dynamically handles both categories of unadjusted references. NMonitor responds to memory violations caused by any attempted read access to XOM. It checks the context and the data being accessed. If the context matches the two cases discussed above and the address being accessed does belong to the relocated data, NMonitor permits and facilitates the access; otherwise, it terminates the program.

Specifically, NMonitor whitelists these two kinds of data and ensures legitimate accesses to them can go through while potential abuses by attackers cannot. For instance, NMonitor only allows C++ runtime module to access the *.eh_frame* sections (updatable through sysctl). For the *.text* embedded data, NMonitor only allows code from the over-approximated hosting function to read them. Note that while this design helps our system cope with those corner cases, the security of our system is barely undermined for two reasons: (*i*) the majority of the whitelisted data are indeed *real* data, which are not even decodable or surrounded by non-decodable data(§ V). (*ii*) Different data require the code from different regions to access them; attackers cannot simply exploit one memory leak bug to read across all these embedded data.

IV. Implementation Details

NORAX is fully implemented based on two commercial mobile phones, Samsung Galaxy S6 and LG Nexus 5X. In this section, we present the implementation of NORAX on LG Nexus 5X, which is equipped with Qualcomm Snapdragon 808 MSM8992 (4 x ARM Cortex-A53 & 2 x ARM Cortex-A57) and 2GB RAM. The phone is running Android OS v6.0.1 (Marshmallow) with Linux kernel v3.14 (64-bit). Table VI shows the SLoC of NORAX on Nexus 5X. In the following, we provide more details about the implementation.

TABLE VI: The SLoC for all NORAX components.

System Modifications	Norax Components	SLoC	Language
Linux Kernel	NLoader, NMonitor	1947	C
Bionic Linker	NLoader	289	C++
Analysis & Rewriting Modules	NDisassembler, NPatcher	3580	Python & Bash Shell Script

A. Kernel Modification

We modified several OS subsystems in order to implement the design discussed in § III. To start off, the memory management (MM) subsystem is modified to enable the execute-only memory configuration (§ II) and securely handle the legitimate page fault triggered by data abort on reading the execute-only memory. Specifically, we intercept the page fault handler, the *do_page_fault()* function, to implement the design of *NMonitor* discussed in § III-E. Implementing the semantics for all kinds of memory load instructions is error-prone and requires non-trivial engineering effort, but above that, there is one additional caveat, as page fault is one of the most versatile events in Linux kernel that has very diversified usages, such as copy-on-write (COW), demand paging and memory page swappings etc. Also, accessing the same virtual address could fault multiple times (e.g., First triggered by demand paging, and then by XOM access violation). If not carefully examined, irrelevant page fault events could be mistakenly treated as XOM-related ones, which may cause the entire system to be unstable or even crash. The solutions proposed in prior works [11], [14] are not directly applicable here, because in ARM64 Linux kernel, to the best of our knowledge, there is not one handy feature such as a flag pushed by the kernel, or a register populated by the hardware to directly indicate whether the fault is really triggered due to a read into the execute-only page that we configure.

To precisely pinpoint the related page fault events, we devise a series of constraints to filter the irrelevant ones. when a page fault happens, the following checks are performed:

- Check if the faulting process contains NORAX converted module, this is indicated by a flag set by *NLoader* when loading a converted binary. This flag will be propagated when the process *forks* a new child, and properly removed if the new child does an *exec* to run a new program.
- Check the exception syndrome register on exception level one (ESR_EL1 [24]) for two fields: (*i*) Exception class and (*ii*) Data fault status code. This ensures the fault is triggered by the user space program, and it faults on the last level page table entry (we only enforce XOM at pte entries) because of permission violation.
- Check the VMA permission flags and only handle the case of reading an execute-only page. All these restrictions together ensure that we do not mistake other page fault events with ours.

To verify the integrity of a violation triggered by XOM,

we extend the *task_struct* to maintain a list of access policies (§ III-E), one for each module. We also instrument the *set_pte* function to ensure the permission of a page must follow the $\mathbf{R} \oplus \mathbf{X}$ policy. This way, we prevent the attacker from tricking the OS to remap the execute-only memory through high-level interfaces. The modified kernel subsystems also include the file system (FS) and system calls where we instrument the executable loader and implement the design of *NLoader* plugins (§ III-D) respectively.

B. Bionic Linker Modification

In a running program, all the libraries needed by the executable are loaded by the linker. In order to handle those converted libraries and make the code regions of the whole process execute-only, we directly modify the linker's source code to place hooks before the library loading and symbol resolution routines as described in § III-D. One quirk of the Bionic linker is that when loading libraries, it places those modules right next to each other, leaving no space in-between. This causes problems from multiple perspectives. Firstly, it lowers the entropy of the address space randomness thus undermines the effectiveness of ASLR. Secondly, it also "squeezes" out the space for *NLoader* to load the NORAX-related metadata. To resolve this issue, *NPatcher* encodes the size of the total metadata into the NORAX header when it recomposes the binary, and we instrument the linker such that when it is loading a library it will leave a gap with the size of the sum of the encoded number (zero for the unconverted binaries) and a randomly generated nuance.

C. System Optimization

NORAX is designed with optimizations inherited in the system, such as updating all possible and updatable references of the relocated *executable data* to avoid page faults. However, given that our implementation is targeting the commercial Android systems, more optimizations could be done by taking advantages of several handy features on Android. For example, we can avoid triggering any page faults by deliberately delay enabling the execute-only configuration during the loading of a program until all the necessary modules are loaded and have their symbols resolved. This is feasible because in Android, for performance reasons, majority of the modules are compiled with lazy binding disabled, that is to say, when loading such module, the linker will promptly resolve all symbols it needs to execute, instead of walking through the loaded modules on a demanding basis during runtime to resolve symbols if compiled otherwise.

Last but not least, a more precise accessing policy for *embedded data* is achievable using the commonly available *.eh_frame* section. This section is compiled into pretty much all the binaries shipped to user phones based on our preliminary survey on multiple user-build AArch64 based Android phones from major OEMs like Samsung, LG, HTC etc. For clarity, we will not expand too much on the technical detail of the *.eh_frame* section. Basically, we can take advantage of the PC range field for each Frame Description Entry (FDE) to facilitate the analysis of NDisassembler.

V. EVALUATION AND ANALYSIS

In this section, we evaluate four aspects of NORAX: (i) whether it breaks the functioning of patched binaries? (ii) how accurate is its data analysis? (iii) how much overhead it incurs? and (vi) how practical is it for wide adoption?

A. Functioning of Transformed Binaries

For this test, we selected 20 core system binaries to transform, including both programs and libraries (Table IX). These binaries provide support for basic functionalities of an Android phone, such as making a phone call, installing apps, and playing videos. We obtain these binaries from a Nexus 5X phone that runs Android OS v6.0.1 (Marshmallow). These stock binaries are compiled with compiler optimization and without debugging metadata.

We tested the functionality of the transformed binaries using our own test cases as well as the Android Compatibility Test Suite (CTS) [25]. We modified the system bootstrapping scripts ($*.rc$ files), which direct Android to load the system binaries patched by NORAX. Table VII shows the specific tests we designed for each system executable and library. For example, $surfaceflinger$ is the UI composer, which depends on two libraries: $libmedia.so$ and $libstagefright.so$. Zygote ($app\_process64$) is the template process from which all app processes are forked. It uses all of the patched binaries. While running our functionality tests, we observed an attempt by the linker to read the ELF header, which is located in the pages marked executable-only. While this attempt was allowed and facilitated by NMonitor, our system can be optimized to handle this case during the patching stage instead.

We also ran the Android Compatibility Test Suite (CTS) on a system where our transformed binaries are installed. The suite contains around 127,000 test packages, and is mandatory test performed by OEM vendors to assess the compatibility of their modified Android systems. The test results are shown in Table VIII. NORAX did not introduce any additional failure than those generated by the vendor customization on the testing devices. The results from both tests show that the functioning of patched binaries is not interrupted or broken by NORAX.

B. Correctness of Data Analysis

To thoroughly test the correctness of our embedded data identification algorithm described in § III-B, we ran the data

analysis module of NDisassembler against a large test set consisting of all 313 Android system binaries, whose sizes span from 5.6KB (*libjnigraphics.so*) to 16.5MB (*liblog.so*), totaling 102MB. For these binaries, we compare the data identified by *NDisassembler* with the real embedded data. Our ground truth is obtained by compiling debugging sections (.debug_*) [26] into the binaries. We use an automatic script to collect bytes in file offsets that fall outside any function range and compare them with the analysis results from *NDisassembler*. For the bytes that are not used by any of the functions, we found that some of them are *NOP* instructions used purely for the padding purpose; whilst some are just "easter eggs", for instance, in the function *gcm_ghash_v8* of *libcrypto.so*, the developers left a string *"GHASH for ARMv8, CRYPTOGAMS by <appro@openssl.org>"*. These kinds of data were not collected by NORAX. Since there are not references to them, making them non-readable will not break any function.

For the tested binaries, *NDisassembler* correctly identified all the embedded data. Only for 28 out of the 313 binaries did NDisassembler reported false positives (i.e., code mistakenly identified as embedded data), due to the over-approximate approach we use (§ III-B). These rare false positive cases are expected by our design and are handled by NMonitor during runtime, as we discussed in § III-B. Table X shows a subset of the results[4].

TABLE VII: Rewritten program functionality tests.

Module	Description	Experiment	Success
vold	Volume daemon	mount SDCard; umount	Yes
toybox	***115*** **nix utilities*	try all commands	Yes
toolbox	***22*** *core *nix utilities*	try all commands	Yes
dhcpcd	DHCP daemon	obtain dynamic IP address	Yes
logd	Logging daemon	collect system log for 1 hour	Yes
installd	APK install daemon	install 10 APKs	Yes
app_process64 (zygote)	Parent process for all applications	open 20 apps; close	Yes
qseecomd	Qualcomm's proprietary driver	boot up the phone	Yes
surfaceflinger	Compositing frame buffers for disply	Take 5 photos; play 30 min movie	Yes
rild	Baseband service daemon	Have 10 min phone call	Yes

C. Overheads and Security Impact

Size Overhead: In our functionality test, the sizes of our selected binaries range from ≈14K to ≈7M, as shown in Table IX. After transformation, the binary sizes increased by an average of 3.91%. Note that $libm.so$ is an interesting case, as its file size increased much more than others. After manual inspection, we found that this math library has a lot of constant values hardcoded in various mathematical functions such as $casinh()$, $cacos()$. As an optimization, the compiler embeds this large set of constant data into the code section to fully exploit spatial locality, which translates to more metadata generated by NORAX during the patching stage.

[4]This subset was chosen to be consistent with the binaries used in the other tests in this section. The complete set of all 313 Android system binaries, which can be easily obtained, are not shown here due to the space limit.

TABLE VIII: System compatibility evaluation, the converted zygote, qseecomd, installd, rild, logd, surfaceflinger, libc++, libstagefright are selected randomly to participate the test to see whether they can run transparently with other unmodified system components.

	Pass	Fail	Not Executed	Plan Name
CTS normal	126,457	552	0	CTS
CTS NORAX	126,457	552	0	CTS

TABLE IX: Binary transformation correctness test.

Module	Size (Stock)	Size (NORAX)	File Size Overhead	# of Rewrite Errors
vold	486,032	512,736	5.49%	0
toybox	310,800	322,888	3.89%	0
toolbox	148,184	154,632	4.35%	0
dhcpcd	112,736	116,120	3.00%	0
logd	83,904	86,256	2.80%	0
installd	72,152	76,896	6.58%	0
app_process64 (zygote)	22,456	23,016	2.49%	0
qseecomd	14,584	15,032	3.07%	0
surfaceflinger	14,208	14,448	1.69%	0
rild	14,216	14,784	4.00%	0
libart.so	7,512,272	7,772,520	3.46%	0
libstagefright.so	1,883,288	1,946,328	3.35%	0
libcrypto.so	1,137,280	1,157,816	1.81%	0
libmedia.so	1,058,616	1,071,712	1.24%	0
libc.so	1,032,392	1,051,312	1.83%	0
libc++.so	944,056	951,632	0.80%	0
libsqlite.so	791,176	805,784	1.85%	0
libbinder.so	325,416	327,072	0.51%	0
libm.so	235,544	293,744	24.71%	0
libandroid.so	96,032	97,208	1.22%	0
AVG.			3.91%	0

Performance Overhead: We used Unixbench [27] to measure the performance of our system. The benchmark consists of two types of testing programs: (*i*) User-level CPU-bound programs; (*ii*) System benchmark programs that evaluate I/O, process creation, and system calls, etc. We ran the benchmark on both the stock and patched binaries, repeating three times in each round. We then derived the average runtime and space

TABLE X: Embedded data identification correctness, empirical experiment shows our analysis works well in AArch64 COTS ELFs, with **zero** false negative rate and very low false positive rate in terms of finding embedded data. The last column shows the negligible number of leftover gadgets in the duplicated embedded data set.

Module	#. of Real Inline Data (Byte)	#. of Inline Data Flagged by Norax (Byte)	#. of Gadgets found in extracted inline Data
vold	0	0	0
toybox	8	8	0
toolbox	20	20	0
dhcpcd	40	40	4
Logd	0	0	0
installd	0	0	0
app_process64 (zygote)	0	0	0
qseecomd	N/A	0	0
surfaceflinger	0	0	0
rild	0	0	0
libart.so	17716	17716	8
libstagefright.so	296	296	5
libcrypto.so	2472	2512	25
libmedia.so	3936	3936	0
libc.so	4836	4836	5
libc++.so	12	12	0
libsqlite.so	932	1004	13
libbinder.so	0	0	0
libm.so	20283	20291	48
libandroid.so	0	0	0
Total	50551	50671	108

overhead, which are given in Figure 4.

For the runtime overhead, the average slowdown introduced by NORAX is 1.18%. The overhead mainly comes from the system benchmark programs, among which $Execl$ shows the maximum slowdown. Investigating its source code, we found that this benchmark program keeps invoking the *exec* system call from the same process to execute itself over and over again, thus causing NLoader to repeatedly prepare new book-keeping structures and destroy old ones (§ III-D). This process, in turn, leads to multiple locking and unlocking operations, hence the relatively higher overhead. Fortunately, we do not find this behavior common in normal programs. In addition, some simple optimizations are possible: (*i*) employing a more fine-grained locking mechanism; (*ii*) reusing the book-keeping structures when *exec* loads the same image.

For space overhead, on average NORAX introduces 1.78% increase in maximum resident memory and 3.90% increase in file sizes. Table IX shows the file size increase for individual programs. Both resutls indicate negligible space overhead for NORAX system. As explained in § III-C, the space overhead is proportionate to the amount of embedded data as well as their references. On the other hand, the NORAX header incurs a fixed amount of space overhead. If not much embedded

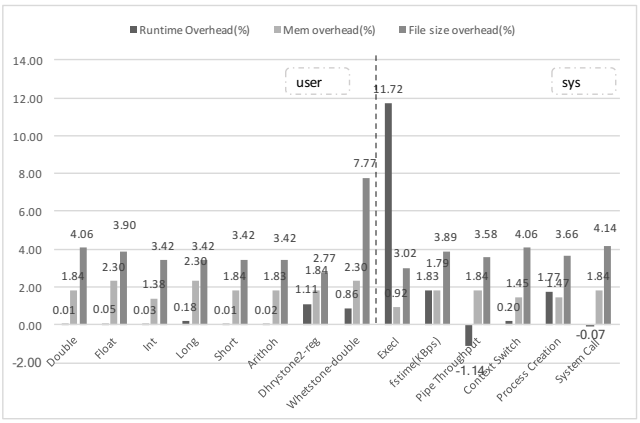

Fig. 4: Unixbench performance overhead for unixbench binaries, including runtime, peak resident memory and file size overhead (left: user tests, right: system tests)

data exist, the references and the header become the major contributor to the space overhead.

Security Impact: The goal of NORAX is to retrofit the $\mathbf{R} \oplus \mathbf{X}$ property into ARM64 COTS binaries. It makes code sections unreadable and redirects references of *embedded data* to duplicate data in read-only memory pages. However, since we duplicate *embedded data*, they are in theory still reusable by adversaries. we conduct a gadget searching experiment in the duplicated *embedded data* appended at the end of the converted binaries. Table X shows the number of available gadgets we found in those data. As the result shows, available gadgets are actually very rare even in the binaries that have a lot of *embedded data* such as $libm.so$, we believe this is because the majority of those duplicated bytes are by themselves not decodable. Also note that the shown numbers are actually an upper bound of the available gadgets. Because, in the executable code section, where the original *embedded data* reside, the bytes that form the gadgets may not be placed next to each other.

D. Practicality Assessment

Szekeres et al. [28] presented three main requirements for a security solution to be practical:

- **Protection**: The security feature must enforce a strict policy and has relatively low false positives and false negatives.
- **Cost**: A practical system should incur negligible runtime slowdown and space overhead.
- **Compatibility**: The security system should not depend on the source code availability. In addition, it should be able to handle different modules individually and the processed modules should work with those unmodified ones.

We examine if NORAX meets these three criteria. NORAX enforces the $\mathbf{R} \oplus \mathbf{X}$ policy, similar to previous defenses that needed source code [8], [9], which is the strongest defense along the line of thwarting direct code read. Regarding the cost, NORAX only introduces $\tilde{1}\%$ slowdown for the majority of the test cases, $\tilde{2}\%$ extra memory and $\tilde{4}\%$ disk consumptions, showing its negligible cost. Finally, NORAX can protect COTS binaries which come without any auxiliary information, and it converts and loads different modules individually. Those converted modules can run seamlessly with the unmodified ones, indicating good compatibility.

VI. RELATED WORK

A. Code Reuse Attack Mitigations

Over the years, there has been an ongoing race between code reuse attacks (or ROP in short) and corresponding defense countermeasures. Such code reuse attacks keep evolving into new forms with more complex attack steps (e.g., Blind-ROP [2], JIT-ROP [1]). To defend against them, three categories of countermeasures (e.g., ASLR, CFI, XOM) have been proposed from different perspectives. Here we briefly review these defenses, especially execute-only memory, which is the category of this paper.

Control Flow Integrity (CFI): Enforcing CFI is a general defense against attacks that hijack control flows, including code reuse attacks. Proposed a decade ago by Abadi et al. [29], CFI has been tuned by researchers over the years [30]–[35], from its early form coarse-grained CFI to its current mature appearance as fine-grained CFI. The fundamental difference is that a coarse-grained CFI allows forward edges in the control flow graph (CFG) to point at any node in the graph and backward edges to return to any call preceded destination, whilst a fine-grained CFI has a more precise set of destinations for both forward and backward edges. bin-CFI [36] and CCFIR [37] enforce the coarse-grained CFI policy on Linux and windows COTS binaries respectively. Unfortunately, enforcing a fine-grained CFI requires a more precise CFG to be built as the ground truth, which is difficult to obtain in practice based on static analysis, even when source code is available. In addition, researchers found that it is still possible to launch code reuse attacks when fine-grained CFI solution is in place due to the difficulty of extracting a perfect CFG in practice [38]–[41].

Address Space Layout Randomization (ASLR): ASLR is a practical and popular defense deployed in modern operating systems to thwart code reuse attacks [42]. It randomizes the memory address and makes the locations of ROP gadgets unpredictable. However, the de-facto ASLR only randomizes the base address of code pages. It becomes ineffective when facing recent memory-disclosure-based code reuse attacks [1], [2]. Such attack explores the address space on-the-fly to find ROP gadgets via a memory disclosure vulnerability. Although fine-grained ASLR increases the entropy of randomization, such as compile-time code randomization [43] and load-time randomization [3], [5]–[7], the memory disclosure attack is not directly addressed, since code pages can still be read by attackers [1]. Runtime randomization [44]–[46] is thus proposed to introduce more uncertainty into the program's address space. Their effectiveness depends on who acts faster, attacker or the re-randomization mechanism. Due to the need of tracking all the code and data objects and correct their references, these solutions either require compiler's assist or rely on runtime translation, which limit their applications and incur non-trivial overhead.

eXecute-only Memory (XOM): To address the memory disclosure attack, researchers proposed execute-only but non-readable ($\mathbf{R} \oplus \mathbf{X}$) memory pages to hinder the possibility of locating reusable code (or ROP gadgets). However, one fundamental challenge to achieve this defense is that it is non-trivial to identify and separate legitimate data read operations in code pages.

When source code is available, existing works like Readactor [8], [9] and LR^2 [10] rely on compilers to separate data reads from code pages and then enforcing XOM via either hardware-based virtualization or software-based address masking. On the other hand, for COTS binaries, which are more common in the real-world scenario, XnR [11] blocks direct memory disclosure by modifying the page fault handler in operating systems to check whether a memory read is inside a code or data region of a process. However, it cannot handle embedded data mixed in code region mentioned in Section III-A. HideM [47] utilizes split-TLB features in AMD processors to direct code and data access to different physical pages to prevent reading code. Unfortunately, recent processors no longer support split-TLB.

Unlike previous works that mostly target x86, NORAX is designed to transform legacy COTS to support XOM on top of latest AArch64 processors. In particular, NORAX focuses on the code-data separation problem of COTS binary on ARM, which has not been systematically investigated before.

Destructive Code Read: Apart from execute-only memory, a different type of approach is to prevent already-disclosed executable memory from being executed. Rather than being execute-only, code segments are not executable after their addresses and values have been leaked. Heisenbyte [13] and NEAR [14] achieve this by overwriting the values of the disclosed code addresses with random values (i.e., invalid opcodes), while keeping the disclosed values in different memory pages for legitimate data reads. Unfortunately, such

approach has to monitor every read access to the code pages, which incurs more page faults and high overhead. In addition, Snow et al. show that such destructive code read can still be bypassed by reloading multiple code copies or inferring code layout without reading it [15]. Since NORAX does not allow the code regions to be read at all, it is not vulnerable to such attacks.

B. Static Binary Analyses

In this subsection, we compare COTS solutions that have analysis goal overlap with NORAX.

Executable Data Identification: Zhang et al. [36] and Tang et al. [13] develop algorithms to identify jump tables embedded in the code using heuristics based on well-defined data structure patterns. This result is not sufficient for $\mathbf{R} \oplus \mathbf{X}$ policy enforcement. NEAR [14] and HideM [47] adopt a more aggressive analysis approach, computing CFG based on similar heuristics for the analyzed binary, and mark all the unknown regions as data. Although this approach has merits such as being architecture-generic and is able to tackle x86-specific challenges like various-length instructions, it inevitably incurs relatively high false positives had the CFG construction process miss any indirect control flow transfer target. Making a different design choice, NORAX does not rely on overly aggressive approach or assumptions about data structures. Instead, its analysis exploits the basic semantics of AArch64 ISA and achieves a larger and more precise coverage.

Executable Data Access Check: Similar to HideM [47], NORAX undertakes the route of whitelisting the range of executable data. This is a strategy to achieve maximum compatibility in the case of missing reference update. However, HideM does not impose restrictions on the accessing subject, plus the fact that it has more false positives on identifying embedded data which exposes more gadgets, hence weakening the security. On the contrary, NORAX enforces configurable data-read policy to ensure only the legitimate reads can succeed, such as embedded data should only be read by (over-approximated) hosting function, and linker-related sections should only be read by the dynamic linker.

VII. DISCUSSIONS

A. Compatibility with Other COTS Hardening Solutions

Execute-only memory alone cannot defend against the ever-evolving code reuse attacks. Thus, we bear in mind that it is important to design NORAX to be compatible with other COTS hardening solutions that provide fine-grained randomization and control flow integrity. Following our design principles (§ III-A), NORAX makes minimum structural changes to binaries programs, which do not preclude running other binary analysis and hardening solutions. For example, the size and location of code and data objects remain unchanged. The control flow properties are preserved. As a result, changes by NORAX are self-contained and will not interfere with the operations of those other solutions.

We examine two representative binary hardening solutions as examples, In-place randomization (IPR) [4] and bin-CFI [36]. IPR is a fine-grained ASLR solution and can be used in tandem with NORAX. It has three transformation passes. First, it substitutes instructions with semantically equivalent ones. The only instruction NORAX could replace is adr used to reference *.text* inline data. Since adr is the only instruction available in AArch64 ISA that can obtain PC-relative reference directly, IPR does not have any alternative candidates to use. Second, IPR reorders instructions sequences that do not have dependencies. NORAX is transparent to such reordering because it does not alter or assume instruction sequences. Third, IPR performs register reassignments. NORAX preserves register usages and thus does not affect register reassignments.

Bin-CFI is a coarse-grained CFI solution for COTS binaries and conceptually compatible with NORAX. It performs indirect control flow (ICF) analysis and then instruments all the ICF transfer instructions to ensure they follow control flow graphs. NORAX only modifies data reference and data accessing instructions and does not impact CFG. Note that albeit designed with maximum compatibility, NORAX does assume to run as the last pass among other binary hardening techniques. This ensures NORAX preserves all data references planted by other passes if any.

B. Current Limitations

Unforeseeable Code: NORAX relies on static binary analysis and rewriting. The current implementation cannot patch dynamically generated code (JIT Compilation) or self-modifying code. In addition, NORAX cannot patch customized ELF files consisting of unrecognizable sections that may contain code and data. For instance, the *.ARM.exidx* and *.ARM.extab* sections contained in the $dex2oat$ program[5] are not recognized by the current implementation of NORAX. Nevertheless, these limitations are shared by almost all static binary rewriting works. It is worth noting that modules converted by NORAX can run alongside programs of this kind seamlessly without suffering any functionality lost.

Indirect Memory Disclosure: NORAX prevents attackers from directly reading the code to search for gadgets loaded in memory. However, code pointers residing in data areas such as stack and heap are still vulnerable to indirect memory disclosure attacks, which can lead to whole function reuse or

[5]An optimization tool to convert applications' byte code to native code.

call-preceded gadget reuse attacks [48], [49]. This limitation, however, is shared by all related solutions using binary rewriting [11], [13], [14], [47]. In addition, a recent study [50] shows even the most advanced source-code based techniques [8], [9] are subject to attacks of this kind. We argue that defense against indirect memory disclosure is another research topic that warrants separate studies and is out of the scope for this work.

VIII. Conclusion

We present NORAX, a comprehensive and practical system that enables execute-only memory protection for COTS binaries on AArch64 platforms. NORAX shows that identifying data from code in COTS binaries, albeit generally undecidable, is in practice feasible under the scope of AArch64 platforms. To demonstrate its practicability, we implemented NORAX on commodity mobile phones including Samsung Galaxy S6 and LG Nexus 5X, and protect their stock system binaries from direct memory disclosure attacks. Our evaluation shows NORAX enforces strong protection, while at the same time incurs negligible overhead–average 1.18% slowdown and 2.21% memory footprint, suggesting it is suitable for real-world adoption.

IX. Acknowledgment

We thank the anonymous reviewers for their insightful comments. We also thank Michalis Polychronakis, Michael Grace, Jia Ma and Xun Chen for the helpful discussions during the development of NORAX. This project was supported by the Office of Naval Research (Grant#: N00014-17-1-2227) and the National Science Foundation (Grant#: CNS-1421824). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 574–588.

[2] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 227–242.

[3] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013, pp. 299–310.

[4] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 601–615.

[5] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "Ilr: Where'd my gadgets go?" in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 571–585.

[6] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software." in *ACSAC*, vol. 6, 2006, pp. 339–348.

[7] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 157–168.

[8] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 763–780.

[9] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, "It's a trap: Table randomization and protection against function-reuse attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 243–255.

[10] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi, "Leakage-resilient layout randomization for mobile devices," in *Proceedings of the 2016 Network and Distributed System Security (NDSS) Symposium*, 2016.

[11] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1342–1353.

[12] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *ACM SIGPLAN Notices*, vol. 35, no. 11, pp. 168–177, 2000.

[13] A. Tang, S. Sethumadhavan, and S. Stolfo, "Heisenbyte: Thwarting memory disclosure attacks using destructive code reads," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 256–267.

[14] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, "No-execute-after-read: Preventing code disclosure in commodity software," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 35–46.

[15] K. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis, "Return to the zombie gadgets: Undermining destructive code reads via code inference attacks," in *IEEE Symposium on Security and Privacy*, 2016.

[16] B. Lee, L. Lu, T. Wang, T. Kim, and W. Lee, "From zygote to morula: Fortifying weakened aslr on android," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 424–439.

[17] "El_0 execute-only memory configuration," https://armv8-ref.codingbelief.com/en/chapter_d4/d44_1_memory_access_control.html.

[18] "Android executables mandatorily need to be pie," https://source.android.com/security/enhancements/enhancements50.html.

[19] R. Wartell, Y. Zhou, K. W. Hamlen, M. Kantarcioglu, and B. Thuraisingham, "Differentiating code from data in x86 binaries," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2011, pp. 522–536.

[20] "Apple warn developers when the binaries are not compiled as position-indepent," https://developer.apple.com/library/content/qa/qa1788/_index.html.

[21] 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."

[22] C. Cifuentes and M. Van Emmerik, "Recovery of jump table case statements from binary code," in *IEEE International Workshop on Program Comprehension*, 1999.

[23] R. Qiao and R. Sekar, "Function interface analysis: A principled approach for function recognition in COTS binaries," in *The 47th IEEE/IFIP International Conference on Dependable Systems and Networks*, 2017.

[24] "Exception syndrome register(esr) interpretation," http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500e/CIHFICFI.html.

[25] "Android compatibility test suite," https://source.android.com/compatibility/cts/index.html.

[26] "Dwarf standards," http://www.dwarfstd.org.

[27] D. Niemi, "Unixbench 4.1. 0."

[28] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 2013, pp. 48–62.

[29] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-Flow Integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*. ACM, 2005, pp. 340–353.

[30] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 941–955.

[31] B. Niu and G. Tan, "Rockjit: Securing just-in-time compilation using modular control-flow integrity," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1317–1328.

[32] ——, "Per-input control-flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 914–926.

[33] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "Ccfi: cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 941–951.

[34] V. Mohan, P. Larsen, S. Brunthaler, K. W. Hamlen, and M. Franz, "Opaque control-flow integrity." in *NDSS*, 2015.

[35] P. Team, "grsecurity: RAP is here," 2016.

[36] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013, pp. 337–352.

[37] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical Control Flow Integrity and Randomization for Binary Executables," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 559–573.

[38] E. Göktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, "Out of control: Overcoming control-flow integrity," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 575–589.

[39] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 401–416.

[40] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 161–176.

[41] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, "Control jujutsu: On the weaknesses of fine-grained control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 901–913.

[42] P. Team, "PaX address space layout randomization (ASLR)," 2003.

[43] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Efficient techniques for comprehensive protection from memory error exploits." in *Usenix Security*, 2005.

[44] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming." in *NDSS*, 2015.

[45] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand live randomization," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM, 2016, pp. 50–61.

[46] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, "Timely rerandomization for mitigating memory disclosures," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 268–279.

[47] J. Gionta, W. Enck, and P. Ning, "Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. ACM, 2015, pp. 325–336.

[48] M. Conti, S. Crane, L. Davi, M. Franz, P. Larsen, M. Negro, C. Liebchen, M. Qunaibit, and A.-R. Sadeghi, "Losing control: On the effectiveness of control-flow integrity under stack attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 952–963.

[49] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 745–762.

[50] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, C. L. Stephen Crane, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi, "Address-Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity," in *Proceedings of the Network and Distributed System Security Symposium (NDSS'17)*, Feb 2017.