2017 IEEE International Conference on Software Maintenance and Evolution

Embroidery: Patching Vulnerable Binary Code of Fragmentized Android Devices

Xuewen Zhang, Yuanyuan Zhang, Juanru Li, Yikun Hu, Huayi Li, Dawu Gu
Shanghai Jiao Tong Univeristy, Shanghai, China
{serenade, yyjess, jarod, yixiaoxian, marcusholloway, dwgu} @sjtu.edu.cn

***Abstract*—The rapid-iteration, web-style update cycle of Android helps fix revealed security vulnerabilities for its latest version. However, such security enhancements are usually only available for few Android devices released by certain manufacturers (e.g., Google's official Nexus devices). More manufactures choose to stop providing system update service for their obsolete models, remaining millions of vulnerable Android devices in use. In this situation, a feasible solution is to leverage existing source code patches to fix outdated vulnerable devices. To implement this, we introduce EMBROIDERY, a binary rewriting based vulnerability patching system for obsolete Android devices without requiring the manufacturer's source code against Android fragmentation. EMBROIDERY patches the known critical framework and kernel vulnerabilities in Android using both static and dynamic binary rewriting techniques. It transplants official patches (CVE source code patches) of known vulnerabilities to different devices by adopting heuristic matching strategies to deal with the code diversity introduced by Android fragmentation, and fulfills a complex dynamic memory modification to implement kernel vulnerabilities patching. We employ EMBROIDERY to patch sophisticated Android kernel and framework vulnerabilities for various manufactures' obsolete devices ranging from Android 4.2 to 5.1. The result shows the patched devices are able to defend against known exploits and the normal functions are not affected.**

I. INTRODUCTION

According to the official statistics released by Google [1], as of March 6, 2017, only 34.1% of the devices are running the latest Android OS version (6.0+). Many devices with previous versions of Android are at a higher risk of malicious attacks due to the disclosed vulnerabilities of older Android systems. Once the vulnerabilities have been disclosed, experienced attackers can quickly utilize them to exploit the device. A successful exploit may lead to immediate malware infection or important information leakage such as bank accounts, even remote users gaining control over infected systems. Therefore, timely patching is always ideal. However, patching existing vulnerabilities for different Android devices is often hindered by the diversification and fragmentation of Android ecosystem. Only a few devices (e.g., Nexus devices of Google) can receive the *Over The Air* (OTA) update related to latest Android's monthly security updates. For most manufacturers, it may take some time to make the patching code available thus the security update often tends to lag behind for a considerable long period. For obsolete devices, the situation is even worse: device manufacturers focus more on promoting new devices rather than providing patches for obsolete devices, and patches are only provided to their in-stock Android devices instead of those obsolete devices due to the complexity of maintaining multiple patching schemes. For instance, Motorola has given up its monthly security updates due to the difficulty of deploying on a monthly basis for all devices [2]. As a result, millions of obsolete Android devices (OS version < 6.0) in use may never receive patches and are exposed to severe security threats nowadays.

The lack of vulnerability patching for Android devices is mainly due to the fragmentation of the Android ecosystem. Unlike iOS devices with unified operating system, the multitude of manufacturer customized Android systems introduced different versions of Linux kernel and diversified program executables/libraries in binary code form. As a result, the standard patching scheme of a specific vulnerability, i.e., patching scheme provided by Android Open Source Project (AOSP)) cannot be directly applied to many customized Android systems. Generally, two main impediments exist for applying one patching scheme to diversified Android devices: First, for one certain code base from AOSP, some manufacturers remove or modify functions to generate an alternative version, and the vulnerabilities may also be moved to different places. Thus a typical patching process fails to work. Second, when a manufacturer stopped supporting one device and did not release the source code related to this device, the patching requires manipulating the binary code of vulnerable OS and applications directly. Working on the binary code determines that the patching scheme is very hard to be universal: manufacturers use different compilers or compilation strategy to generate optimized binary code. Therefore, binary code's diversity of specific executable (e.g., a shared library responsible for media file decoding) on different devices is significant, and existing patches released for the code of standard AOSP cannot be transited to these diversified executables.

To help still-in-use obsolete Android devices defend against publicly revealed exploits, and address the issue of Android fragmentation to generate adaptive patches, we introduce EMBROIDERY, an automated patch generation system. Since it is impossible to obtain each manufacturer's compiling environment and source code, EMBROIDERY deals with binary executable on the device directly. It focuses on the patching of vulnerabilities in kernel and native libraries of obsolete Android devices (OS version < 6.0). The workflow of EMBROIDERY begins with a heuristics based vulnerability locating, which helps find the assembly code line where the vulnerability and

978-1-5386-0992-7/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSME.2017.15

```
ADD             R7, SP, #0x1C8+var_188
LDR             R0, [R7]
LDR             R3, [R4]
LDR.W           R12, =(_ZSt7nothrow_ptr - 0x110418)
ADD             R0, R3
LDR.W           R1, [R11,R12] ; _ZSt7nothrow_ptr ; std::nothrow
BLX             _ZnajRKSt9nothrow_t ; operator new[](uint,std::nc
```

(a) case 1

```
LDR             R0, [SP,#0x1A8+var_168]
LDR             R3, [R7]
ADDS            R0, R0, R3 ; unsigned int
BLX             _Znaj   ; operator new[](uint)
```

(b) case 2

Fig. 1. the fragmentation of CVE-2015-3864

essential patching information for the patching process exist. Then a binary rewriting step is employed to patch existing vulnerabilities. EMBROIDERY generates different patching code for diversified devices. Particularly, EMBROIDERY fulfills a complex dynamic memory modification to implement kernel vulnerabilities patching, which is compatible with some enhanced kernel memory protection features.

To evaluate the effectiveness of EMBROIDERY, we tested it with six popular manufacturers' nine obsolete Android devices to check whether two sets of well-known vulnerabilities can be patched. The evaluation demonstrated that the patches generated by EMBROIDERY can be accurately applied to the vulnerable point in binary executables of those devices, and the vulnerabilities were fixed after the patches were applied. We also summarized the diversities of vulnerability for each device and found that EMBROIDERY is very helpful for code patching when dealing with such diversities.

The main contributions of this paper are:

a. EMBROIDERY transplants official security patches (CVE source code patches) that are only suitable for certain Android devices (e.g., Nexus devices) to a broader spectrum of Android devices. The heuristic matching strategies adopted by EMBROIDERY helps locate not only the appropriate places in binaries for patching code insertion but also analyze the essential patching information for the patching process.
b. EMBROIDERY generates binary code patches of kernel and system framework for different Android devices despite their diversifications. EMBROIDERY employs binary rewriting to adapt typical restrictions of Android devices (e.g., device locking that prohibits the re-flashing of kernel and system partition). Particularly, EMBROIDERY addresses the issue of kernel memory protection measures that hinder common kernel hot patching schemes and achieves stable and generic kernel memory patching.

II. PROBLEM OVERVIEW

A typical scenario for Android system patching often starts with the discovery of a vulnerability that affects not only the latest version of Android but also all past versions. Generally, the vulnerability is published through Google's Android Security Bulletin released monthly with moderate information. Then details and relevant exploits may be disclosed by other researchers or not. In this situation, however, only the latest version of Android system and related devices (e.g., Nexus smartphone) receive the patch. Obsolete versions (e.g., Android 4.4), although are often also affected by such

Listing 1
THE PATCHES OF CVE-2015-3864

```
status_t MPEG4Extractor::parseChunk
(off64_t *offset, int depth) {
...
      size = 0;
    }
+   if ((chunk_size > SIZE_MAX) ||
+       (SIZE_MAX - chunk_size <= size)) {
+     return ERROR_MALFORMED;
+   }
    uint8_t *buffer = new uint8_t[size + chunk_size];
...
```

vulnerability, may never receive corresponding patches. To fix those vulnerable devices, a third-party patching scheme is expected. In particular, this paper focuses on the patching of native code of Android kernel and system framework, since the bytecode patching has been well studied by other researches [3]. The patching of vulnerability in native code is more complicated compared to bytecode patching due to its diversification. Unlike Java bytecode that is unified for different platforms, the released native code varies significantly because of the fragmentation of Android ecosystem. Take the assembly code of the CVE-2015-3864 as an example. Figure 1 illustrates two different compiled versions of the same vulnerable function (source code of the vulnerable function can be seen in Listing 1). The patch of this vulnerability needs to add a check before the memory allocation (*new*). However, the assembly implementation is different at the level of binary code. As a result, the patching process must consider such differences at the assembly level (different register allocation, instruction reordering, branch inversion, etc.) due to the diversity of compilation. Moreover, to apply the patch at the binary code level, essential patching information is needed. This involves the binary code analysis of related variables and addresses. For instance, the patch in Listing 1 requires a binary analysis to identify the registers storing *size* and *chunk_size*, the address of return code when the check fails (which involves operations about loading the value of *ERROR_MALFORMED*, stack adjusting and arguments restoring).

In detail, the following issues should be carefully handled to fulfill an effective binary level patching:

a. *Assembly differences*. Due to the compilation differences, the assembly code varies much (e.g., different register allocation, instruction reordering). To solve the problem, the patching utilizes heuristic matching strategies to locate and perform assembly a code analysis for patching

information.

b. *OS variances*. A specific vulnerability may be disclosed in a higher version of Android (e.g., 6.0). However, the host function of this vulnerability may have a different name or even not exist in past versions of Android. In this case, the patching has to firstly search the location of vulnerability with code similarity strategy.
c. *Structure definition*. A typical diversification is that the definitions of some data structures have been changed by the manufacturer. This modification leads to the changes at assembly level including the offset between a certain member function or data member, and the structure instance address. Thus the binary code patching should not rely on such hard-coded constants.
d. *Reordering*.
Sometimes basic blocks may reorder. For example, the location of return code (stack restore and function return) may be placed in the middle of program control flow. Thus the locating process cannot just search the code lines straightforwardly according to the logic order of source code.

Another main obstacle of our discussed patching is how to deal with kernel vulnerabilities. Although compared with system framework vulnerabilities, kernel objects diversification are less complex, kernel patching comes out some other fragmentation issues. Different manufacturers apply different kernel compilation mode and kernel protection features, which may cause trouble for kernel patching. First, some patching methods rely on the use of Loadable Kernel Module (LKM) to patch the vulnerable kernel. Nonetheless, it is difficult to generate a generic pre-compiled kernel module due to the fragmentation of Android ecosystem. What's more, many devices including Google Nexus disable the LKM support or implement module signing. The kernel specifically prohibits loading a module compiled against a different kernel version. In addition, some manufacturers make use of TrustZone to ensure the kernel module cannot be loaded even the device was compromised. The kernel module will go through a mandatory digital signature verification happening in TrustZone. As a result, kernel patching with LKM is not feasible. Second, directly rewriting the kernel binary code is not straightforward. Since updating the kernel is strictly restricted by bootloader lock, it is not able to apply statical binary patching for vulnerable kernel image. Hence, the only way is to dynamically rewrite the kernel memory. However, Android kernel nowadays has utilized many features to harden itself and ensure the integrity. Direct kernel memory modification violates common *read-only kernel text/data* protection policies such as *CONFIG_STRICT_MEMORY_RWX* [4] and *CONFIG_DEBUG_RODATA* [5]. Besides, the patching should also consider how to find an executable memory area to place the patching code, which may be hindered by the memory $W \hat{} X$ protection.

III. EMBROIDERY

We put forward EMBROIDERY, a patching system which fulfills binary code patching task for Android kernel and framework vulnerabilities with adaption to Android diversification and fragmentation. The purpose of EMBROIDERY is to generate patches for those obsolete devices that cannot receive corresponding security update. The key challenges here are how to find a universal approach to pinpoint all vulnerabilities precisely in binary code, and how to patch them according to existing security updates. In general, EMBROIDERY considers six crucial factors to implement an effective binary level patching for obsolete Android devices.

1) *Patch details*. EMBROIDERY leverages the source of the monthly Android Security Bulletin [6]. Since the security updates released by Google are usually related to Common Vulnerability and Exposures (CVEs), we focus on generating patches based on specific CVEs. Our EMBROIDERY patching system relies on a manual input of the patching details according to the details of CVE, which includes the vulnerable functions and the corresponding patching code. Since Google publishes Android Security Bulletin every month, containing details of security vulnerabilities that affect Android devices, we can obtain the latest Android security information including the CVE ID, security vulnerability details, and the assessed severity. We firstly collect the details of these CVEs to obtain patch details (vulnerable functions and how to patch them), which are listed in the AOSP source code diff information.
2) *Vulnerability locating*. Since it is impossible to obtain each manufacturer's source code and publicly released patches are often based on latest version of Android, a vulnerability locating procedure is required to search vulnerable binary executable and find the vulnerability in different versions of Android OS and different devices.
Particularly, the locating of vulnerability is to firstly find a host function, and to further determine the patching point and obtain the essential patching information (e.g., related registers storing arguments, return address) for patching. EMBROIDERY locates the vulnerable functions with the help of function symbols. A patching point is the exact assembly code line that EMBROIDERY overwrites with a branch instruction, which links to the patching code. In Figure 1, the patching point is the *blx* line. The vulnerability here requires an argument check before the *new* call to fix. Thus EMBROIDERY overwrites the patching point and replaces it with a branch instruction, leading the control flow to the inserted argument checking procedure. EMBROIDERY introduces some heuristic matching strategies used in previous studies, which leverage code similarity metrics to help determine the position of patching point and patching information.
3) *Binary rewriting*. The vulnerable functions identified by EMBROIDERY are in kernel memory or shared objects, which are both binary executables. To patch them, a

binary rewriting process is required. It places the patching code in a new executable memory area, and then redirects the original control flow by rewriting patching point with branch instructions to lead the execution to the patching code. An important point here is that the binary rewriting should not replace the entire vulnerable function. After manually reversing and analyzing various related binaries, we have found that one specific function in different devices may change much due to the manufacturer's modification. Therefore, replacing it with a patched version compiled from the original source code base introduces potential corruption, although fixing the vulnerability. To ensure functionality in every device from different manufacturers, EMBROIDERY only modifies the vulnerable assembly code lines.

4) *Device modification*. Existing Android devices do not always provide the permission for users to modify the original image of critical system partitions. As a result, EMBROIDERY requires root permission to circumvent the restriction and fulfill such modification. The feasibility of this requirement is due to the existence of universal Android exploits for obsolete devices. Although EMBROIDERY leverages existing exploits to obtain root permission, it only utilizes the permission to fulfill the patching instead of other privilege operations. After the patching, EMBROIDERY would release the permission so that it will never be abused. Only if the manufacturers supply a ROM applying the newest security patch available in time for a certain device, users can directly flash a new whole OS to the most recent Android version. But the manufacturers need time to adapt the device and sometimes they even ignore the security update. EMBROIDERY can provide a timely patching for these obsolete devices.
5) *Kernel hot patching*. Compared to conventional static patching approach, the patching of Android kernel requires adopting hot patching due to the infeasibility of re-compiling kernel image and re-flashing kernel image partition. Considering the restriction of many devices that does not allow the installation of kernel module, EMBROIDERY directly accesses and rewrites the kernel virtual memory through */dev/kmem* [7] [8] with the obtained root permission. Despite similar code disassembling and vulnerability searching, a distinguishing step in EMBROIDERY's hot patching is the circumvention of kernel memory protections. As memory modification is in conflict with kernel protections such as *read-only kernel text/data*, a straightforward modification fails to work. EMBROIDERY elaborately resets the kernel page tables and refreshes the permission to guarantee that the modification would not trigger a violation and crash the system. Therefore, EMBROIDERY adopts a universal dynamic kernel patching method with compatibility of kernel memory protection features, which can be successfully applied to most devices with Android OS version 4.x and 5.x.
6) *Bootstrap*. Since we apply the kernel hot patching, EMBROIDERY is required to launch at Android bootstrap time. We can put the system in Android's boot partition and modify the booting script file to launch system service during the early stage of Android system's booting.

In the following sections, we detailed how an Android device with a vulnerable kernel or system framework is fixed by EMBROIDERY despite the Android fragmentation issues.

IV. LOCATING

This section details how EMBROIDERY locates the vulnerability and obtains the essential patching information. EMBROIDERY focuses on the patching of Android kernel and system shared objects. The disassembling results of these binaries are usually considered as reliable since they were generated using conventional compilation options. Based on the disassembling results, EMBROIDERY introduces heuristic matching strategies to address the Android fragmentation issue.

A. Locating Vulnerable Functions

EMBROIDERY firstly locates the vulnerable function with the help of function symbols. For kernel vulnerabilities, EMBROIDERY obtains the kernel symbols by reading */proc/kallsyms* file. Until Android 5.1, ASLR is still not applied on kernel level and symbol addresses are still being determined at compile-time [9]. Hence the addresses of kernel symbols can be found in the */proc/kallsyms* file with sysctl1 *kptr_restrict* [10] enabling at runtime (with the root privilege, it can be enabled by setting the *kptr_restrict* to 0). As for framework vulnerabilities, the symbols of the shared objects exported are utilized, as Android system objects are not stripped. In most CVEs, the vulnerable function is an exported function. Otherwise, EMBROIDERY makes use of the cross-reference of the vulnerable function in other exported functions. According to the AOSP source code, EMBROIDERY evaluates which exported function has called the vulnerable functions and locates this function with symbols. If the caller is still not an exported function, EMBROIDERY traces back to the upper caller recursively until finding an exported function. Then EMBROIDERY relies on the call flow graph of the exported function to locate the vulnerable function.

B. Locating Patching Point

After locating the vulnerable function, EMBROIDERY further locates the exact patching point with heuristic matching strategies. Summarized by previous studies and empirical binary analysis of the realistic vulnerabilities, our proposed heuristic matching strategies leverage many code similarity metrics to help determine not only the patching point but also the essential patching information. Thus, by making use of the existing CVE knowledge and realistic analysis results, EMBROIDERY summarizes for a certain CVE the common code similarity metrics shared by different devices. Then it utilizes these code similarity metrics to perform matching and locating despite Android fragmentation. Generally, the following code similarity metrics are leveraged:

- *Normalized operands*. A patching point often involves a certain instruction in assembly code, and this instruction is generally related to particular mnemonics and operands. Due to the different development and compilation environments, there are different representations at the assembly level (register allocation, instruction reordering, branch inversion, etc.), which hinder the analysis of binary code. Thus, EMBROIDERY normalizes the operands and partitions them into different categories: register references, memory references, immediate, coprocessor register type and real number like the method proposed by Sbjrnsen et al. [11]. This helps normalize the assembly instructions and solve the problem that compilers may make different operands choices.
- *Call reference*. A call to an exported function is conspicuous and can be quickly located in assembly code lines. Thus, EMBROIDERY makes use of cross-reference of the function call to help locating like the method proposed by Yamaguchi et al. [12] [13]. If there is an exported function call nearby the patching point, EMBROIDERY locates that function call with function symbols to approach the patching point. Since the candidate area is reduced, EMBROIDERY can conduct a precise search around the neighbor code area to identify the patching point by specific mnemonics (e.g., add, bl).
- *Constants*. If the vulnerable function involves special constants, it is helpful to utilize these constants to quickly match and locate the related code line [14].
- *Specific-purpose registers*. If the patching point involves specific-purpose registers (e.g. the parameters or return value of a function call), EMBROIDERY identifies these registers and locates the related code line by virtue of ARM calling conventions. For example, there is patching point involving the first parameter in a function call. EMBROIDERY locates that function call and performs a simple taint analysis to search all the references of register *r0* before the function call (the first parameter is stored in register *r0*). As all the code lines involving references of register *r0* are obtained, EMBROIDERY evaluates them to find the patching point. Moreover, for the buffer overflow patching, it may be a good choice to target at register *sp* and *fp*. These specific-purpose registers are useful to help understand the program crash information.
- *Offsets*. Certain offsets to particular pointers (e.g., *this* pointer, a certain *struct*) can be leveraged to help locating. If the vulnerability involves a member function or data members of a C++ class, we can locate it by observing how *this* pointer is used to invoke member function or access data member. As *this* pointer is always stored in register *r0* in ARM as the first argument in C++ member functions, EMBROIDERY performs a simple taint analysis to search all the references of it and finds those concerned ones. And the offsets of certain members can be calculated according to the definition of *struct* or *class* in AOSP source code. It should be noticed that definitions may be modified by some manufacturers so that these modified binaries have different offsets for a certain member. To solve this problem, we have analyzed different manufacturers' binaries and found that in most cases the definitions of *struct* or *class* are the same as the official ones in AOSP. While for special cases, we found that although the definition has been changed, the offset between two neighbor members usually remains. So based on this feature, EMBROIDERY uses the relative offset to find out target members.

C. Analyzing Patching Information

Once the patching point is located, EMBROIDERY continues to analyze the assembly code beside the patching point to provide essential patching information for the patching process. For example, as for logic vulnerabilities, many bugs are usually fixed by adding the missing sanitization checks. If the check fails, the function should return a result representing failure to the caller. The next step in those cases is to find the return code (load the return value representing failure, adjust the stack, return to the caller). Thus, EMBROIDERY conducts a binary code analysis on related registers and memory with heuristic matching strategies to obtain the patching information.

- *Relevant registers and memory analyzing*. In most cases, the patching information is related to specific registers or memory addresses, and around the patching point. EMBROIDERY conducts a search around the patching point and analyzes the relevant registers and memory. And the code metrics we discussed above are utilized to help analysis. Take the CVE-2015-3864 as an example, EMBROIDERY backtracks the nearby instructions around the patching point to find the related registers storing *chunk_size* and *size*. Since the registers are related to function parameters, they can be identified with the *Specific-purpose registers* method.
- *Return code analysis*. As we discussed, the return code has to be found if an added check fails. The return code is the address of the control flow leading to the code about stack adjusting and argument restoring, which is unique in one function. For ARM, processors have featured the Thumb instruction set to improve compiled code-density. The crucial consideration is the switch between ARM and Thumb state. There are four common no-condition branch instructions in Thumb: branch (B), branch with link (BL), branch with the mode switch (BX), branch with link and mode switch (BLX). A failure return code always uses simple branch instruction, as it does not need link and mode switch. Besides the return value is loaded before the branch instruction. The location where the instruction jumps to can also be inferred by analyzing the branch instruction. So the normal return code pattern is that program jumps to another address with simple branch instruction after loading an immediate value. Hence, EMBROIDERY can locate the return code by matching these features.

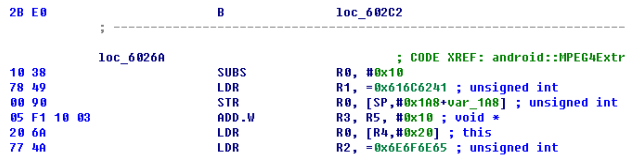

Fig. 2. Choosing patching point

V. PATCHING

With patching point and essential patching information collected in the locating process, EMBROIDERY automatically generates different patches for various Android devices. Then EMBROIDERY conducts the dynamic memory rewriting for kernel vulnerabilities and static object rewriting for framework vulnerabilities respectively to apply the patches.

A. Generating Patches

For a certain CVE, EMBROIDERY introduces a binary code template and fills the collected information into it to generate the adapted patch for various Android devices. The key is to link the patching code to existing code. EMBROIDERY rewrites the patching point with branch instruction and redirects the control flow to the patching code. There are two points needed to pay attention to. One is the switch between ARM and Thumb mode. Since the patching code is usually short and pithy, EMBROIDERY generates the code in Thumb mode to reduce memory requirements and cost. Thus, EMBROIDERY makes use of branch instruction with link to redirect the control flow to patching code (the return address is automatically stored in *lr*), and utilizes the instruction *mov pc, lr* in patching code to get back to the normal routine. Another is the instruction size problem. When the size of patching point is not fit for the size of rewritten branch instruction, we need to choose a new patching point. Figure 2 gives a concrete example of how to appropriately choose the patching point when considering the instruction size: the size of the original patching point *subs r0, #10* is just two bytes. However, since the target address of the patching code area is usually far from the patching point, EMBROIDERY has to rewrite it with a four-byte branch instruction (long jump). The essential step here is to evaluate that if the neighbors of the patching point involve PC relative information. If the neighbor code line does not involve PC relative information and is 4-byte length, we choose it as the new patching point. In this case, opcodes of the neighbor instructions *b* and *ldr* vary with the current position address (*b* and *ldr* are PC-relative addressing). Thus it is infeasible to rewrite them. As a result, EMBROIDERY expands the range of searching area and it finally chooses the *add.w* instruction as the new patching point.

B. Kernel Dynamic Memory Rewriting

There are two main stages in applying patches for kernel vulnerabilities in EMBROIDERY system. The first step is to obtain an executable memory region to place the patching code, and the next is to rewrite the patching point to link the patching code. These steps both involve kernel code modification.

1) Kernel Code Modification: As Section II discussed, kernel code modification is in conflict with the kernel memory protection feature READ-ONLY KERNEL TEXT/DATA which makes the kernel text segment read-only. EMBROIDERY elaborately resets the kernel page tables and refreshes the permission to guarantee that the modification would not trigger a violation and crash the system.

READ-ONLY KERNEL TEXT/DATA features can be implemented in two ways. For *CONFIG_DEBUG_RODATA* (kernel 3.18+), all the section entries that overlap the kernel text section are replaced with page mappings. For obsolete Android devices (OS $<$ 6.0), page-mapping is a two-level mapping. Each active entry in the Page Global Directory (PGD) table points to a page frame containing an array of Page Middle Directory (PMD) entries of type *pmd_t*, which in turn points to the page frames containing Page Table Entries (PTE) of type *pte_t*. And PTE points to the page frames containing the actual user data and holds the relevant protection flags [15]. As for *CONFIG_STRICT_MEMORY_RWX*, the kernel page mapping uses the section mapping, which is a one-level mapping, and the protection flags are just stored in the relevant locations. Thus, according to the differently configured page mapping, EMBROIDERY resets the protection flags to modify the page *rwx* attributes.

EMBROIDERY firstly locates the kernel page table. At the hardware level, ARM supports two page table trees simultaneously with the hardware registers *TTBR0* and *TTBR1*. *TTBR0* is unique per-process, and is stored in *current→mm.pgd* (i.e., *current→mm.pgd* == *TTBR0* for that process). When a context switch occurs, the kernel sets *TTBR0* with the *current→mm.pgd* of the new process. On the other hand, *TTBR1* is global for the whole system, and points to the page tables of the kernel. It is referred in the global kernel variable *swapper_pg_dir*, which is statically initialized at compile time. Almost all the obsolete Android devices (OS version $<$ 6.0) install 32-bit kernel, in which the `0-0xBFFFFFFF` area belongs to user space while the `0xC0000000-0xFFFFFFFF` area belongs to kernel space. The swapper page is usually located at address `0xC0004000-0xC0008000`, and the *swapper_pg_dir* is usually `0xC0004000`. As for the rest devices, EMBROIDERY locates it by finding the idle process (PID = 0), whose *mm.pgd* always points to the *swapper_pg_dir*. Thus, EMBROIDERY locates the kernel page table by leveraging the information of *swapper_pg_dir*.

However, modifying the kernel page table is not enough to make the page attributes take effect in reality. Actually, every process has a copy of this master page table for entries between `0xC0000000` and `0xFFFFFFFF`. The above step only modifies the kernel page table's attributes. If EMBROIDERY overwrites the kernel text segment immediately after modifying, the system would crash since the kernel still considers the kernel text segment as read-only. In the normal logic process, kernel flushes the modified page table after page table rewriting. Overwriting a data pointer to *flush_tlb_all* function

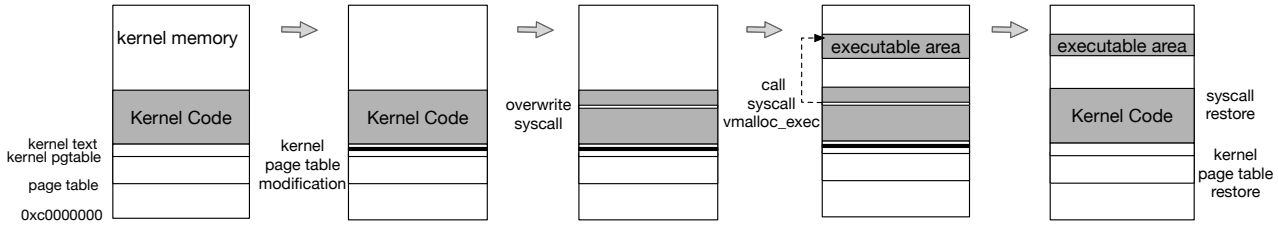

Fig. 3. allocate an executable kernel memory region

and calling it to flush the modified page table by force seems a good choice. But our experiment shows that the method does not work since kernel still regards the kernel text segment as read-only. One idea is to find the *mm_struct* structure of itself (the main patching process) and modify the kernel page table the process refers to (*current→mm.pgd*) directly. But as the location of the process is unknown and the patching process is in user space (cannot call the kernel function and refer to the *current→mm.pgd*), it requires a search in a wide range of memory to find every process's own kernel page table and modify all of them (it does not know which the patching process's page table is). EMBROIDERY applies a more efficient method. When a new user process launches, the kernel creates a new page directory and copies all the kernel mappings from the swapper page (page frames from 3-4GB) to the new page table and clears the user pages. Therefore, EMBROIDERY starts a new process to write the kernel text segment, and the new process inherits the global kernel page table EMBROIDERY has already modified. Now EMBROIDERY can modify the kernel code, and EMBROIDERY would restore the kernel page table immediately after kernel text segment writing at last.

2) Patching: EMBROIDERY has to prepare an extra executable memory region to place patching code. As the kernel memory has been strictly segmented, it is hard to find such a suitable region. Allocating a physmap region as the executable region using ret2dir techniques [16] seems a choice. The physmap is a large, contiguous virtual memory region inside kernel address space that contains a direct mapping of part or all (depending on the architecture) physical memory. But according to the study of Xu et al. [17] and our experiment, actually, the physmap region is not executable in ARM. Besides, the method of modifying page attributes does not work since it is required to modify all the processes' kernel page tables forever in case of a crash, which is a massive work.

EMBROIDERY utilizes the kernel function *vmalloc_exec* (a kernel-internal function to allocate enough pages to cover the page level allocator and map them into contiguous and executable kernel virtual space [18]) to allocate an executable memory region. The process is depicted in Figure 3. Firstly EMBROIDERY needs to place the calling *vmalloc_exec* code in an existing executable region. The executable region is required to be seldom used in case of process crash and could be triggered in the user space. What's more, after the calling, EMBROIDERY should obtain the address of the allocated memory region. Considering all these conditions, EMBROIDERY overwrites a seldom used system call with the shellcode to invoke *vmalloc_exec*. EMBROIDERY triggers the system call, and the shellcode is executed to allocate an executable kernel memory region. The allocated memory address is obtained as the system call's return result. EMBROIDERY restores the system call and kernel page table. Then EMBROIDERY places the generated patches in this allocated memory region and rewrites the patching point. Thus, EMBROIDERY implements the kernel dynamic memory patching work.

C. Framework Static Objects Rewriting

For framework vulnerabilities, EMBROIDERY applies static object rewriting and replacing method. In a word, EMBROIDERY adds a text segment in the original shared objects, inserts patching code in that new segment, and links the patching code to the existing code. EMBROIDERY firstly checks the shared object and calculates a suitable start address of the new text segment. It modifies the header of the shared object to generate a new text segment according to the ELF format [19] [20]. An object file's section header table stores all the locations of the sections, and the ELF header's *e_shoff* member gives the byte offset from the beginning of the file to the section header table, while *e_phoff* member holds the program header table's file offset in bytes, etc. EMBROIDERY adjusts these arguments to insert a new text segment. Then it adds the new text segment information by modifying the object's header. After the inserting, EMBROIDERY rewrites the patching point with branch instruction and places the generated patches in the new segment. Finally, EMBROIDERY replaces the vulnerable object on the device with the patched one (usually with third-party recovery).

VI. EVALUATION

In this section, we evaluate EMBROIDERY with different real world devices which contain multiple vulnerabilities. To check whether EMBROIDERY can be applied to both kernel and framework vulnerabilities patching, we test it using two sets of vulnerabilities that affect most obsolete Android devices. The first set of vulnerabilities contains two **cross-platform Linux kernel vulnerabilities**, CVE-2014-3153 [21] and CVE-2015-3636 [22], that lead to universal root from Android version 2.3 to 5.x. The second set of vulnerabilities are related to the

TABLE I
THE SIMILARITY OF *parseChunk* FUNCTION. "FAILED" MEANS THAT BINDIFF MATCHES THE *parseChunk* FUNCTION TO ANOTHER INCORRECT FUNCTION. "1" MEANS THE TWO FUNCTIONS ARE IDENTICAL, WHICH IS IN REGARD TO THEIR INSTRUCTIONS INSTEAD OF THEIR MEMORY ADDRESSES.

	Nexus4	Nexus5	Galaxy Nexus	VEBW1D07	meizu	Smartisan	Samsung S5	Samsung note3
OS version	4.2.2	5.0.1	4.2.2	4.2.2	5.0.1	4.4.4	5.0.0	5.0.0
Nexus4	-	0.38	1	0.49	0.35	0.56	0.30	0.28
Nexus5	0.38	-	0.38	failed	0.82	0.43	failed	failed
Galaxy Nexus	1	0.38	-	0.49	0.36	0.56	0.30	0.28
VEBW1D07	0.49	failed	0.49	-	failed	0.36	0.42	0.42
meizu	0.35	0.82	0.36	failed	-	0.51	failed	failed
Smartisan	0.56	0.43	0.56	0.36	0.51	-	0.50	0.48
Samsung S5	0.30	failed	0.30	0.42	failed	0.50	-	0.98
Samsung note3	0.28	failed	0.28	0.42	failed	0.48	0.98	-

Stagefright library [23] in Android framework. The Stagefright library (*/system/lib/libstagefright.so*) is used as a back-end engine for playing various multimedia formats such as MP4 files. Due to the complexity of media format parsing, multiple vulnerabilities were discovered in this library that allow a remote attack with a malicious media file to execute arbitrary code with system privilege on the device. For the first set, we choose those two CVEs for two main reasons: 1) the huge amount of influenced Android devices not only made the patching process a complicated one (our study reveals that these vulnerabilities still exist in most obsolete Android devices and can be exploited by attackers), but also lead to diverse forms of vulnerable code on different devices; 2) these two CVEs are well-known for their applicability–most Android devices with vulnerable kernel can be exploited. The exploits against these two CVEs, known as the "Towelroot" and the "Ping Pong Root", were publicly released and can be used by any attackers. For the second set, we choose them because of the feature that a single shared library containing various vulnerabilities, which is a suitable test case for our evaluation. According to the reports [24] [25] from Google's security bulletin and Zimperium, the number of related vulnerabilities related to Stagefright is over one hundred that many manufacturers were in no rush to update older models, leaving many devices in danger. We believe that those two sets of vulnerabilities could help evaluate the effectiveness of EMBROIDERY's patching.

Targeting obsolete devices, our evaluation chooses nine Android devices of six different manufacturers. The set of devices not only include the Google official devices–Galaxy Nexus (a.k.a. Samsung I9520), Nexus 4, and Nexus 5, but also include heavily modified devices with installed OS ranging from Android 4.2 to 5.1. Among them, we check two kernel vulnerabilities (CVE-2014-3153, CVE-2015-3636) and 17 representative vulnerabilities related to the Stagefright library including CVE-2015-1538, CVE-2015-1539, CVE-2015-3823, CVE-2015-3824, CVE-2015-3826, CVE-2015-3827, CVE-2015-3828, CVE-2015-3829, CVE-2015-3864, CVE-2015-3867, CVE-2015-3868, CVE-2015-3871, CVE-2015-3876, CVE-2015-6598, CVE-2015-6599, CVE-2015-6603, and CVE-2015-6604. Before the testing of EMBROIDERY, we firstly manually analyzed those devices to check the detailed implementations of those vulnerabilities. The CVE-2014-3153 is due to the *futex_requeue* function in kernel through version 3.14.5 does not ensure that calls have two different futex addresses, which allows local users to gain privileges. The CVE-2015-3636 vulnerability is due to the *ping_unhash* function in *net/ipv4/ping.c* in the Linux kernel before version 4.0.3 does not initialize a certain list data structure during an unhash operation, which allows local users to gain privileges or cause a denial of service (use-after-free and system crash) by leveraging the ability to make a *SOCK_DGRAM* socket system call for the *IPPROTO_ICMP* or *IPPROTO_ICMPV6* protocol, and then making a connect system call after a disconnect. Thus we reverse engineered the kernel images of those devices to locate them. For Stagefright vulnerabilities, our location revealed that not every device contains all 17 vulnerabilities. After manually analyzing, we found that some vulnerabilities have been fixed in some devices, or certain functions in the Stagefright shared library are removed or modified for particular devices and thus some vulnerabilities do not exist.

The vulnerability detection using EMBROIDERY proves the accuracy of our proposed system. For all tested devices, 97 concrete CVEs are detected and located in kernel and framework binaries (91 in framework and 6 in kernel). Particularly, EMBROIDERY could accurately locate vulnerabilities with no **false positive**. In our kernel vulnerabilities locating test (devices for kernel evaluation are listed in Table II), only the devices with OS version 4.x contain both CVE-2014-3153 and CVE-2015-3636, while the devices in 5.x contain CVE-2015-3636. EMBROIDERY distinguished this accurately. In the framework vulnerabilities locating test, the situation is even more sophisticated due to the Android fragmentation and diversification. EMBROIDERY reported the number of the Stagefright CVEs located in Table III, which precisely corresponds to our aforementioned manual analyzing results.

To illustrate the fragmentation and diversification EMBROIDERY faces, shared objects on different devices are evaluated using a state-of-the-art binary code similarity evaluation tool Bindiff [26]. We make use of Bindiff to compare the *android::MPEG4Extractor::parseChunk* function (1000+ LOC), one commonly used function involving many Stagefright bugs (e.g. CVE-2015-3823, CVE-2015-3824, CVE-

TABLE II
KERNEL DIVERSIFICATION AND FRAGMENTATION

	1JIa B0195ZRD4Q	VEBW1D07	Nexus4	Nexus5
Android version	5.1	4.2.2	4.2.2	5.0.1
LKM support	n	y	n	n
/dev/kmem support	y	y	y	y
read-only kernel text/data	y	n	n	y
image signing	n	n	n	n

TABLE III
THE CVES DETECTED AND PATCHED

	Nexus4	Nexus5	Galaxy Nexus	VEBW1D07	meizu	Smartisan	Samsung S5	Samsung note3
OS version	4.2.2	5.0.1	4.2.2	4.2.2	5.0.1	4.4.4	5.0.0	5.0.0
CVEs (Total: 17)	11	13	11	11	13	11	8	13

2015-3827, CVE-2015-3829, CVE-2015-3864) in *libstagefright.so* of each two devices. We set the Google official devices (Galaxy Nexus, Nexus 4, and Nexus 5) as the template to verify the distinction of binary code more precisely, BinDiff works on the abstract structure of an executable, ignoring the concrete assembly-level instructions in the disassembly. Every function gets a signature, based on the structure of the (normalized) flow graph of the function [27]. Table I shows the similarity of the compared *parseChunk* function between each two devices. Here the metric of similarity is a value between zero and one, indicating how similar two matched functions are. BinDiff only considers basic blocks, edges and mnemonics for calculating similarity values. In particular, instructions may differ in their operands, immediate values and addresses, but it will still be considered equal if the mnemonics match. [27].

It can be observed in Table I that the library in Android 5.0.1 is obviously distinct from that in other Android versions. Bindiff even fails to match *parseChunk* function of Android 5.0.1 to some implementations in other Android versions, which indicates those shared objects distinctly differ from each other. And even with the same Android OS version, the libraries are still distinct with different manufacturers. Furthermore, most of the devices have less than the 50% similarity compared with another device, indicating that each library varies greatly in different manufacturer devices with different OS versions. The diversification and fragmentation of kernel objects is similar to that of shared objects, and the details are not listed in this paper.

Considering those diversifications, our test showed that EMBROIDERY is still effective to locate the patching point and patching information with the heuristic matching strategies. The heuristic matching strategies EMBROIDERY applies helped normalize the assembly code diversification. And based on patch details of the corresponding CVE information, EMBROIDERY has located all the patching points in all these 97 cases. Our manual verification later demonstrated that the patching points located by heuristic matching strategies are all appropriate locations to insert the patching code. With the ability to adapt to the significant differences presented in Table I, EMBROIDERY has performed well in locating for those 19 CVEs on nine different manufacturer devices ranging from Android 4.2 to 5.1. Besides, for kernel vulnerabilities, as Section II discussed, different manufacturers apply different kernel compilation mode and kernel protection mitigation, which may cause trouble for kernel patching. We choose devices from different manufacturers ranging from Android 4.2 to 5.1 and investigate the kernel mitigation features they applied. The investigation result is shown in Table II. In response, EMBROIDERY performed patching with adaption to Android fragmentation and memory protection features. All of the CVEs detected in various devices have been successfully patched in our evaluation. The results show that the heuristic vulnerability matching strategy we applied is generic, and EMBROIDERY can adapt to the Android diversification and kernel memory protection features independent of devices differences.

To evaluate the compatibility of EMBROIDERY and make sure all of the generated patches work stably without affecting the normal functionality, we validate them from different aspects. We collect all the publicly revealed exploits and Proof of Concepts (PoCs) for these CVEs to evaluate the patched devices. This time the added patching code is executed when the PoC or exploit triggers the vulnerability, and the control flow of the function returns with the specific value as expected. For kernel vulnerabilities, we inserted the *dump_stack* kernel function into patching code to verify if the code has been successfully inserted and executed. And we found the stack trace has been logged in *dmesg* messages, which means that the patching code was executed. The results showed that the heuristic vulnerability matching and page table modification strategies we applied are generic, and EMBROIDERY conducts a universal, cross-device kernel patching solution adapting to kernel memory protection features. For framework vulnerabilities, after manually analyzing and evaluating the patched binary code and the call flow graph it generated, we found that EMBROIDERY has successfully located the patching points and patching information, and applied patches to all devices. We also inserted a logging function into the patches to evaluate

that if patches have been successfully executed by the program. Besides PoCs, we also conducted experiments with some normal MP3 and MP4 files, which also leads the control flow into the patching code but pass the checks and return to the normal routine. With the help of the logging function under a dynamic debugging process, we observed that the execution flow of the program runs normally. It demonstrates that the patched shared object works well without affecting the normal use of devices. Also, we asked volunteers to operate the patched device for three days and they report no crash. All of the patched devices worked as usual after our patching.

VII. Limitation and Discussion

The current EMBROIDERY system implementation has some limitations. For framework vulnerabilities patching, as we perform binary rewriting, it may be against the static integrity protection mechanisms (e.g., image signing) some devices apply. One solution is to do dynamic memory rewriting like our kernel rewriting method.

For kernel vulnerabilities patching, there are some devices with */dev/kmem* disabled. In such cases, we can combine the kmem method with the kernel module method. But when the manufacturer implements memory and page table protection, both of the methods cannot take effect. For example, Samsung proposed RKP [28]. RKP ensures that translation tables cannot be modified by the Normal World through making them read-only to the Normal World kernel. Hence, the only way for the kernel to update the translation tables is to request these updates from RKP. As a result, RKP guarantees that this interception is non-bypassable. Actually, */dev/kmem* combined with kernel module method can perform the patching work for most obsolete devices. EMBROIDERY system can be further improved if there exists a joint effort between the device manufacturers and the third party patch developers.

VIII. Related Work

Before our work, the most well-known system for Android's framework vulnerabilities patching is PatchDroid [3]. PatchDroid is based on dynamic, in-memory patching of running processes. It supports patching of vulnerabilities on Dalvik bytecode and uses dynamic binary instrumentation to inject patching code into running processes on Android. The POLUS [29] system uses the similar solution with Patchdroid other than it requires access to the source code of the target application. EMBROIDERY applies a static binary rewriting based patching on system objects and also demonstrates dynamic patching method on kernel vulnerabilities. Compared with dynamic patching, our static framework patching does not need to set up a building environment that accommodates a large assortment of devices. Moreover, it can be applied to stock or third-party Android ROMs directly.

Another patching scheme [30] leverages Mobile Device Management (MDM) to push security update without the participation of the manufacturers. However, it does not discuss how to generate patching code if manufacturers do not provide source code.

Some third-party patch developers also release their own patches [31] for specific vulnerabilities. They recompile the source code of Android to generate shared objects without vulnerabilities, but the released binaries are only available for specific devices only, while EMBROIDERY can adapt to varied Android versions in different manufacturers' devices.

Also, some frameworks have been proposed for vulnerabilities patching. IntPatch [32] automatically fixes Integer-Overflow-to-Buffer-Overflow vulnerabilities in C/C++ programs at compile time. ClearView [33] is a system for automatically patching errors in deployed software. It works on stripped Windows x86 binaries without any need for source code, debugging information, or other external information, and without human intervention. Here are some recent work on automatic program repair [34] [35].

For Linux kernel patching, the academia and industry have proposed some live kernel patching solutions such as kpatch [36], ksplice [37] and livepatch [38]. They allow users to apply security patches to a running kernel without rebooting. Unlike previous hot update systems, Ksplice operates at the object code layer, which allows it to transform many traditional source code patches into hot updates with little or no programmer involvement. But they also require the kernel source code and are specified for each vulnerability and device. In addition, these solutions are not compatible with ARM. If we want to make them work on ARM, kernel source code modification is required. Thus they are not conventional solutions.

IX. Conclusions

In this paper, we discussed the feasibility of locating and patching critical vulnerabilities in binary executables against the fragmentation of the Android ecosystem. We designed and implemented EMBROIDERY, a binary rewriting system to achieve a generic binary level code patching for most obsolete Android devices. EMBROIDERY locates vulnerabilities in kernel and system framework of the device, and is able to patch them using both static and dynamic binary rewriting techniques. EMBROIDERY thoroughly considers many restrictions and adopts a universal patching scheme available for commodity Android products. The evaluation with two sets of representative Android vulnerabilities demonstrated that even the diversification of binary executables from different devices is significant, EMBROIDERY is able to accurately locate vulnerabilities and generate corresponding patches.

Acknowledgment

We would like to thank the reviewers for their insightful comments which greatly helped to improve the manuscript. This work was partially supported by the Key Program of National Natural Science Foundation of China (Grants No.U1636217), the Major Project of the National Key Research Project (Grants No.2016YFB0801200), and the Technology Project of Shanghai Science and Technology Commission under Grants No.15511103002.

REFERENCES

[1] Google, "Android devices statistics," https://developer.android.com/about/dashboards/index.html.

[2] "Motorola confirms that it will not commit to monthly security patches," http://arstechnica.com/gadgets/2016/07/motorola-confirms-that-it-will-not-commit-to-monthly-security-patches/.

[3] C. Mulliner, J. Oberheide, W. Robertson, and E. Kirda, "Patchdroid: scalable third-party security patches for android devices," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 259–268.

[4] J. J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski, *Android hacker's handbook*. John Wiley & Sons, 2014.

[5] "Config_debug_rodata: Make kernel text and rodata read-only," http://cateee.net/lkddb/web-lkddb/DEBUG_RODATA.html.

[6] "Android security bulletins," https://source.android.com/security/bulletin/.

[7] S. Cesare, "Runtime kernel kmem patching," *VX Heavens, Nov*, 1998.

[8] S. Devik, "Linux on-the-fly kernel patching without lkm," *Phrack Magazine*, 2001.

[9] J. Oberheide, "Exploit mitigations in android jelly bean 4.1," *The Duo Bulletin*, vol. 16, 2012.

[10] D. Rosenberg, "kptr restrict for hiding kernel pointers from unprivileged users," https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=455cd5ab305c90ffc422dd2e0fb634730942b257.

[11] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 117–128.

[12] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 797–812.

[13] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 359–368.

[14] M. Bourquin, A. King, and E. Robbins, "Binslayer: accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*. ACM, 2013, p. 4.

[15] R. W. Carr, "Virtual memory management," in *UMI Research Press, Ann Arbor, Mich*. Citeseer, 1984.

[16] V. P. Kemerlis, M. Polychronakis, and A. D. Keromytis, "ret2dir: Rethinking kernel isolation," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 957–972.

[17] W. Xu, J. Li, J. Shu, W. Yang, T. Xie, Y. Zhang, and D. Gu, "From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 414–425.

[18] "Linux kernel code cross reference," http://lxr.free-electrons.com/source/mm/vmalloc.c#L1877.

[19] T. I. S. Committee *et al.*, "Executable and linkable format (elf)," *Specification, Unix System Laboratories*, 2001.

[20] H. Lu, "Elf: From the programmer's perspective," in *NYNEX Science & Technology Inc*. Citeseer, 1995.

[21] "cve-2014-3153," https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-3153.

[22] "cve-2015-3636," https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-3636.

[23] "Stagefright: Everything you need to know about google's android megabug," http://fortune.com/2015/07/28/stagefright-google-android-security/.

[24] "Stagefright: Vulnerability details, stagefright detector tool released," https://blog.zimperium.com/stagefright-vulnerability-details-stagefright-detector-tool-released/.

[25] "Stagefright (bug)," https://en.wikipedia.org/wiki/Stagefright_(bug).

[26] H. Flake, "Structural comparison of executable objects," *DIMVA 2004, July 6-7, Dortmund, Germany*, 2004.

[27] "Bindiff manual," https://www.zynamics.com/bindiff/manual/.

[28] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 90–102.

[29] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "Polus: A powerful live updating system," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 271–281.

[30] C. V. Bockhaven and J. van Kerkwijk, "Android patching: From a mobile device management perspective," 2014.

[31] "Stagefright-vulnerability-fix," http://forum.xda-developers.com/galaxy-note-3/themes-apps/stagefright-vulnerability-fix-note-3-t3175087.

[32] C. Zhang, T. Wang, T. Wei, Y. Chen, and W. Zou, "Intpatch: Automatically fix integer-overflow-to-buffer-overflow vulnerability at compile-time," in *European Symposium on Research in Computer Security*. Springer, 2010, pp. 71–86.

[33] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan *et al.*, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 87–102.

[34] F. Thung, D. X. B. Le, D. Lo, and J. Lawall, "Recommending code changes for automatic backporting of linux device drivers," in *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016.

[35] S. Ma, D. Lo, T. Li, and R. H. Deng, "Cdrep: Automatic repair of cryptographic misuses in android applications," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 711–722.

[36] "kpatch," https://github.com/dynup/kpatch.

[37] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 2009, pp. 187–198.

[38] "livepatch," http://lxr.free-electrons.com/source/include/linux/livepatch.h.