ARMPatch: A Binary Patching Framework for ARM-based IoT Devices

Mingyi Huang* and Chengyu Song

Department of Computer Science and Engineering, University of California, Riverside, 900 University Ave, Riverside, CA 92521, United States
E-mail: tobyxdd@gmail.com; csong@cs.ucr.edu
**Corresponding Author*

Received 11 May 2021; Accepted 22 June 2021;
Publication 21 September 2021

Abstract

With the rapid advancement of hardware and internet technologies, we are surrounded by more and more Internet of Things (IoT) devices. Despite the convenience and boosted productivity that these devices have brought to our lives and industries, new security implications have arisen. IoT devices bring many new attack vectors, causing an increment of cyber-attacks that target these systems in the recent years. However, security vulnerabilities on numerous devices are often not fixed. This may due to providers not being informed in time, they have stopped maintaining these models, or they simply no longer exist. Even if an official fix for a security issue is finally released, it usually takes a long time. This gives hackers time to exploit vulnerabilities extensively, which in many cases requires customers to disconnect vulnerable devices, leading to outages. As the software is usually closed source, it is also unlikely that the community will review and modify the source code themselves and provide updates. In this study, we present ARMPatch, a flexible static binary patching framework for ARM-based IoT devices, with a focus on security fixes. After identified the unique challenges of performing binary patching on ARM platforms, we have provided novel features by replacing, modifying, and adding code to already compiled programs. Then, the viability and usefulness of our solution has been verified through demos and final programs on real devices. Finally, we have discussed the current limitations of our approach and future challenges.

Keywords: ARMPatch, ARM-based IoT devices, ARM platforms.

Introduction

During the last decades, the development of Internet technologies has caused a great increase in the number of Internet of Things (IoT) devices in our daily lives [1]. Furniture, tools, toys, sensors, doorbells, cameras, cars, etc. are no longer single-purpose hardwired objects. Instead, they share similarities with computers that have a considerable computational power, equipped with corresponding peripheral hardware. Nowadays, these devices have operating systems, run upgradeable software and are connected to the Internet to receive commands or even upload telemetry.

While this has brought great convenience to our lives and boosted productivity in various industries, the security implications should not be underestimated. IoT devices bring many new attack vectors, leading to a higher number of attacks that target or exploit them [2]. Moreover, these devices tend to lack software transparency in comparison with the traditional desktop and server sector. In the end, security vulnerabilities often go unfixed because providers are generally not informed in time, they have stopped maintaining these models, or they simply no longer exist. Although fixes of security issues are sometimes released, their development often takes a long time. This implies that usually vulnerabilities have been already exploited by hackers if customers have not disconnected their IoT devices. As their software is usually closed source, it is difficult for the community to review and modify it to provide updates.

In this context, a binary patching framework that can modify compiled binaries is useful. Such a framework would help security researchers freely patch the firmware of IoT devices when a vulnerability is discovered, avoiding the potentially lengthy process of contacting the developer and waiting for an updated firmware version. Additionally, non-security researchers may also use this framework to customize the devices to meet their own business needs. Even though there are other approaches in the literature, we believe their solutions are not suitable for fixing vulnerabilities in IoT devices owing to a variety of reasons.

In this paper, ARMPatch, a flexible static binary patching framework for ARM-based IoT devices focused on security, is presented. The unique

Table 1 Comparison of current binary rewriting/instrumentation solutions

	Platform(s)	Type	Usage	Overhead
PEBIL [4]	x86, x86_64	Static	Debugging	Moderate
Dyninst [5]	x86, x86_64	Static/Dynamic	Debugging	Low
E9Patch [6]	x86_64	Static	Multi-purpose	Low
Multiverse [7]	x86, x86_64	Static	Debugging	Moderate
DynamoRIO [8]	x86, x86_64, ARM, AArch64	Dynamic	Debugging	High
Valgrind [9]	x86, x86_64, ARM, AArch64, PPC32, PPC64, MIPS32, MIPS64	Dynamic	Debugging	Very High
Frida [10]	x86, x86_64, ARM, AArch64	Dynamic	Reverse engineering	Moderate

challenges of performing binary patching on ARM platforms have been identified. Then, functions of compiled binaries (both executables and libraries) have been modified by adding code, replacing, or modifying it to provide new features. The viability and usefulness of our solution has been verified through demos and final programs on real devices. Finally, current limitations of our approach are discussed.

Background and motivation, ARM is the most popular platform on mobile and embedded devices today. Currently, 90% of these devices use chips based on the ARM architecture [3]. However, despite its dominance in the mobile and embedded world, there is a lack of binary rewriting instrumentation solutions for ARM-based devices. Most established and mature binary rewriting projects only support platforms such as x86, PowerPC, or SPARC. Projects that do support ARM are primarily dynamic instrumentation tools, designed for debugging purposes and reverse engineering, and not for patching security vulnerabilities.

Nevertheless, dynamic instrumentation tools are not applied to fixing security vulnerabilities in the firmware of IoT devices because they are usually large standalone frameworks that need to run concurrently with the target program in order to inject into the target process and alter instructions and states at runtime. This may be ultimately possible or even preferable if the goal is, for instance, to reverse engineer an application on a rooted Android phone. However, it is impractical to use the same approach in IoT

Table 2 Comparison between x86 and ARM instruction sets

	x86	ARM
Instruction Type	CISC, variable lengths	RISC, fixed length
Instruction Mode	Single mode, x86	2 modes, 3 types (ARM, Thumb 16/32)
Program Counter	Not accessible	Readable and writable, like a standard register
Position Dependency [11]	Poor PIC support. PIEs are rare	Good PIC support. PIEs are common
Inline Data [12]	Rare	Common

environments where both software and hardware are designed to run only the necessary routines to operate the device. Thus, running these frameworks on IoT devices would imply additional complexity, which will also lead to excessive overhead.

A more practical approach would be to use a static binary patching tool that supports ARM platforms, especially designed to provide the required framework to fix security vulnerabilities. Noteworthy, we have also born in mind the possibility of adapting existing binary rewriting projects to support ARM; however, they are often deeply coupled to the x86 instruction set and the supported systems, making it difficult to provide ARM support. This is to be expected, as problems and solutions usually vary for different instruction sets and platforms. Moreover, the aforementioned interfaces are also especially designed for debugging purposes and reverse engineering and thus, they are not suitable for security fixes.

Compared to other architectures that have been well studied in this area, ARM poses some unique challenges. To begin with, it essentially consists of two different sets of instructions: ARM and Thumb. While ARM instructions have a fixed length of 4 bytes, Thumb provide a separate set designed to reduce footprint of code that uses 2- or 4-byte instructions. Binaries often contain both ARM and Thumb instructions at the same time, switching between them as needed. ARM also has a readable and writable program counter, which allows devices to use any instruction to change the flow of control. Compared to earlier architectures, ARM was designed considering the PIC (position-independent code), so many binaries instructions operate relative to the program counter. As we will see below, this is both a benefit and a drawback when it comes to binary patches. It is also worthy to mention that inline data (i.e., interleaved in the middle of instructions) is common in

ARM binaries, leading to additional factors that must be considered when disassembling and rewriting.

Since our focus is on security, a necessary step in deciding what capabilities our framework should have is understanding what common security fixes look like, what they entail, and how extensive the changes would be. Therefore, we have collected 66 high risk vulnerability fixes (i.e., identified as "High" or more by the Common Vulnerability Scoring System) released during 2010–2020 from various IoT related open-source projects (e.g., libraries such as *openssl* or *libcurl*, common command-line utilities, and operating systems such as "OpenWrt") in the National Vulnerability Database (NVD). After evaluating these fixes, we have categorized the following types of changes:

- Function changes (FC): involve changes in the internal logic of certain functions.
- Function additions (FA): involve the creation of completely new functions.
- Data changes (DC): involve data changes.
- Data additions (DA): involve adding new data.
- Structure changes (SC): involve changes in data structures.

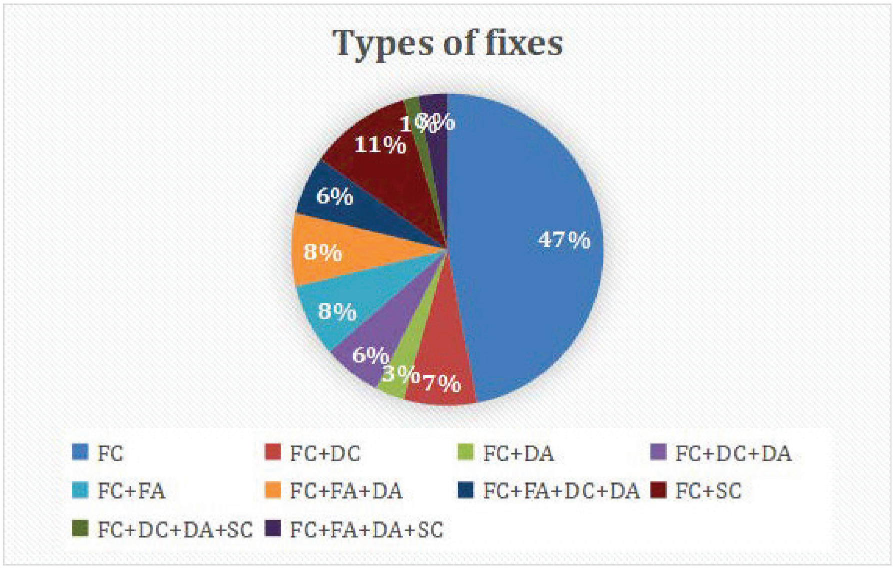


Figure 1 Type of fixes according to our categorization.

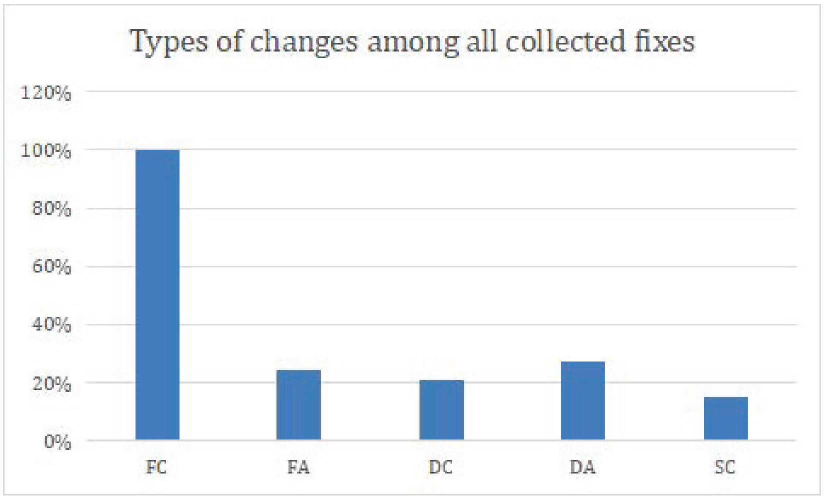


Figure 2 Types of changes among all collected fixes.

As expected, all fixes involved at least some kind of changes to the existing functions. In fact, for almost the half of the fixes (i.e., 47%), only function modifications were required. We have observed that many of these fixes added some additional validations, fixed some conditional expressions, and modified the parameters of some function calls. After these modifications, the machine code generated for those functions, as well as their lengths, are obviously altered. Changes to data structure are particularly difficult to manage, since the concept of data structures does not exist in complied binaries, which simply define how memory regions are accessed and manipulated. Thus, data structure changes often lead to variations in the generated machine code of the functions that use the data structure, even if the code is completely untouched.

Design, Our implementation and evaluation target “OpenWrt” devices running on ARM processors. The “OpenWrt” project is a Linux-based open-source operating system built for embedded devices, currently used in many routers, access points and other network equipment [13]. Note that routers are typically gateways for traffic on home and business networks, a common and lucrative target for hackers. Although we used Linux-based “OpenWrt” for the evaluation, all concepts and techniques covered in this manuscript apply to all ARM platforms.

As shown in the figure above, ARMPatch takes 3 inputs: (i) the original binary to be patched, (2) a configuration file that defines how the patch

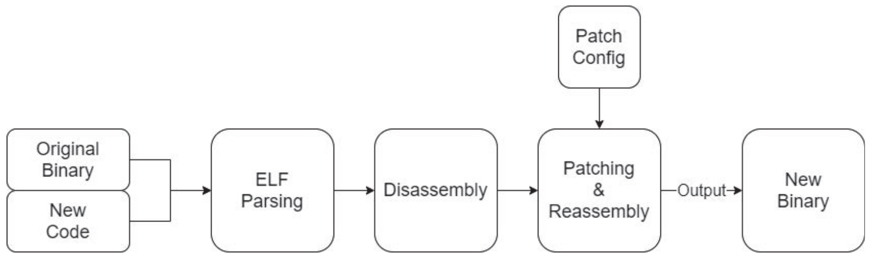


Figure 3 Processing flow.

must be performed, and (iii) one or more binary files containing the code that will be injected. The pipeline is then composed by 3 stages: parsing, disassembling, and patching. Each of these processing stages are detailed in the following subsections.

ELF Parsing, Executable and Linkable Format (ELF) is the standard file format for executables and shared libraries on Linux. Immediately after receiving the original binary, ARMPatch parses its ELF header and performs basic sanity checks to ensure the input is valid and supported. Then, the information needed by the disassembly and patching stages is extracted. For instance, a common ELF parsing stage would check if the binary targets ARM architectures, if it is an executable or a dynamic link library, and if it uses position-independent code (PIC). It would also parse section and segment definitions, keeping track of imported and exported symbols, identifying code and data areas, and creating and internal mapping. Such information is essential for disassembly, patching and the final reassembly stages.

Configuration, A configuration file given by the user controls the processing of the original binaries and corresponding new code to inject. Each configuration file is essentially a collection of descriptions operations supported by our implementation. Among them, we found:

– Function Replacement: replaces an original function in the binary with a new externally provided implementation.
– Data Replacement: replaces data in the original binary, including but not limited to strings, numeric constants, etcetera.
– Function Diff Patch: modifies parts of an original function in the binary based on an externally provided patch file.
– Dependency Injection: adds a new dynamic link library dependency to the binary. Optionally, it also allows redirecting original binary functions to this library.

Users can freely combine these operations to compose a configuration file, and eventually create a complete patch when testing it. The implementation of the operations will be detailed in the following sections.

Disassembly, In the last stage, ARMPatch disassembles both the original binary and the provided new code according to the results of the previous stages. The following pseudo-code state lays the background for the patching stages as it allows decoding instructions and retrieving instruction addressing information and dependencies.

Algorithm: Pseudo-code to analyze the input assembly of a function

```
F: Function, B: Binary File, I: Instruction List, R: Reference List
function AnalyzeFunction(F)
    I = empty
    Ir = empty // PC-relative instructions
    Ri = empty // Internal references
    Re = empty // External references
    for each f in F
        i = DecodeInstruction(f)
        I.add(i)
        if i.PC_relative
            Ir.add(i)
        end if
    end for
    for each i in I
        rs = DecodeReferences(i)
        for each r in rs
            if r.target >= F.begin && r.target <= F.end
                Ri.add(r)
            else
                Re.add(r)
            end if
        end for
    end for
     return I, Ir, Ri, Re
end function
```

Function Replacement, Function replacement allows the user to replace a function in the original binary with a new implementation. The replacing of binary instruction is not trivial as the length of the new function (Ln) is probably not the same as the original one (Lo). If Ln = Lo, the process is simple; however, if Ln > Lo, the new code will not fit into the original position without overwriting other instructions or data that is stored below. We have identified three possible approaches to solve this problem, identifying their advantages and disadvantages:

(1) Insert instructions directly, then move the code that follows it. Even though it may be considered the most intuitive approach, it is in fact the least feasible. Compiled code is a tight structure and cannot be easily changed in length or position. If we change the length of one or more of the functions in the middle, we would inadvertently change the addresses of all subsequent content, completely breaking hardcoded address references of position-dependent code. Even for position-independent code, we would also alter the distance between the content before and after the new code, causing relative offsets to change and point to incorrect addresses. To fix this issue, we would have to analyze and correct all relevant instructions throughout the entire binary. However, rewriting the entire binary is a slow and hard (if not impossible) process. In fact, correctly identifying functions and distinguishing between code and data is still a challenge on ARM [14].

(2) Place the new function as a new segment at the end. It is possible to add new segments at the end of a binary and use them to encapsulate new code or data by modifying the ELF header. The flexibility of the ELF format allows segment definitions to be placed anywhere in the file, making it possible to move new segments to the end of the binary without worrying about changing the length of the header. In this way, we can place the new function without breaking references. Therefore, to replace the original function with the new one, one can simply insert jump instructions at the beginning of the header. Note that this procedure turns the original function instructions into dead code, though. It is also noteworthy to mention that a lot of space is wasted if the function being replaced is large, which could be a problem for IoT devices where storage is severely limited.

(3) Hybrid approach: overwrite and add new segment. A hybrid approach would be overwriting when $Ln = Lo$ and putting the function in a new segment when $Ln > Lo$. In the latter, it is also possible to overwrite part of the instructions and put the rest in a new segment, connected with jumps; albeit this process would require careful instruction disassembly and rewriting for the new function.

Our current implementation uses the overwriting and new segment hybrid approach. However, we did not employ the partial function overwriting

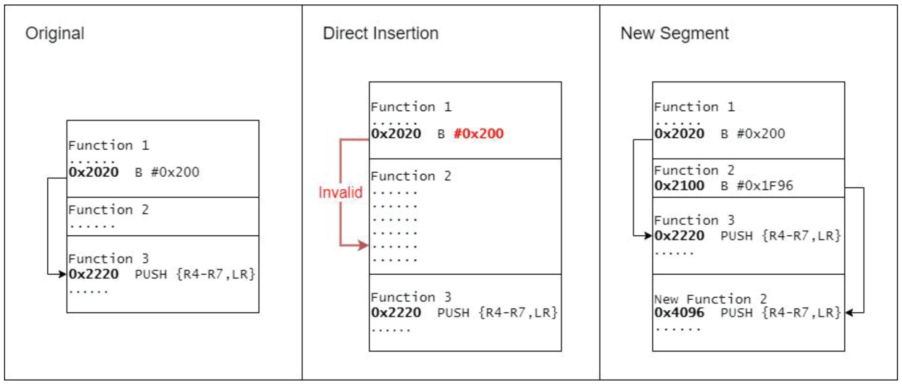


Figure 4 Direct Insertion vs New Segment procedures.

```
00 C0 9F E5    ldr   ip, [pc]
1C FF 2F E1    bx    ip
40 8A F7 01    .word 0x1F78A40
```

Figure 5 An example of “veneer” code.

because of the non-trivial details involved. Hence, if the new function exceeds the size of the original, it will be entirely moved to a new segment. ARM’s unconditional PC-relative branch instruction B is used to jump from the original function to the new replacement. Nonetheless, care must be taken as the supported range is limited to $\pm$32MB due to design [15]. When very large binaries are required to be patched (rare in IoT environments), the distance from the original function to the end of the new segment may exceed this limit. In this case, we use a “veneer” to load the range into the R12 register, taking the length of three instructions up instead of only one. Although it occurs rarely, a potential problem with this approach could be the shortage of space to insert the veneer when the original function is composed of less than 3 instructions.

ARMPatch also accepts compiled executables and dynamic link libraries as the source of new functions. Users need to specify, in the configuration file, the locations of new functions in the files using symbols or offsets. Note that these inputs go through the same parsing and disassembly stages as the source binaries.

Owing to the coupling of binary files, it is not possible to extract the instructions of a function from a binary, copy them into another binary

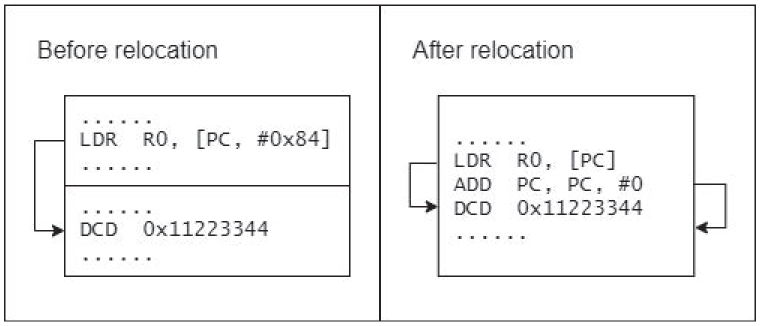


Figure 6 An example of value-based relocation.

and expect it to work. In order to assure a proper function operation, it is required to:

(1) Correct external references. Besides very basic pure algorithmic functions, the code often depends on external references. They may be constants or global variables of data segments, other functions in the same binary, or third-party library functions previously imported through dynamic linking. These references must be corrected, considering that they may use PC relative offsets or fixed addresses.

(2) Correct internal references. The code may contain branches to internal basic blocks due to the presence of control flows. If the entire function is moved without changing any instructions, internal references are not an issue. However, if external references must be corrected and thus, the length of the function changes because of the replacing of instructions; internal references must be fixed as well.

In addition to the aforementioned correction procedures, it is required to perform an “instruction relocation” (do not confuse with linker relocation). The process in detailed in the next section.

Instruction Relocation, During the disassembly stage, we have analyzed each instruction in the function, finding the places where external references are involved. They may be branch instructions such as “BL #0x70d08”, PC-relative loads/stores like “LDR R1, [PC, #0x2b4]”; or any other instructions that contain PC as operands such as “ADD R1, PC”. First, we determine whether they are code or data by identifying the type of the instruction and the segment in which it is included. Here, three cases are possible:

(1) Data. By default, ARMPatch extracts the value from the referenced location and places it under the instruction. Then, it skips the data to

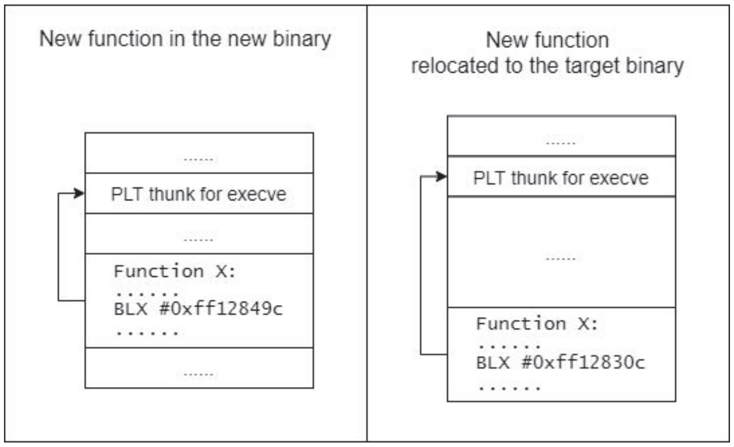


Figure 7 An example of imported function relocation.

prevent it from being falsely executed as an instruction by including "ADD PC, PC, #n". Although this can work for any bytes, half-words, words, and double words; it cannot cope with larger data structures, strings, etc. In such cases, it is also possible to configure ARMPatch to append an entire data segment from the new binary to the original binary. The relocation will adjust the instructions to point to the correct new locations in the data segment.

(2) Imported function. Calls to functions imported from external dynamic link libraries are recognizable because they are usually branches of the PLT section. By using the relocation entries of the ELF header, it is possible to figure out what are the symbols of the imported functions. Then, it is just needed to retrieve the locations of the PLT entries of the same functions in the original binary to complete the relocation. A limitation of our current implementation is that it cannot handle the cases where the new function refers to external symbols that were not imported in the original binary. Nevertheless, it is theoretically possible, although complicated, to add new relocation entries, GOT, and PLT to the original binary.

(3) Other functions in the binary. If the new function calls other functions within its binary, the user has two options: (1) provide those functions alongside it, or (2) let ARMPatch heuristically search for their equivalents in the original binary. If these functions have the same names for their symbols in both the original and new binaries, ARMPatch will associate them accordingly. On the other hand, if symbols are not available (e.g., stripped binaries), our current implementation generates byte patterns based on the instructions in the referenced functions and looks for them into the original binary. Although we are aware that this is not a perfect solution, optimizing this solution would be out of the scope of this manuscript. As a future research line, we consider integrating other state-of-the-art approaches such as DeepBinDiff [16].

Since the above steps involve replacing the original instructions with their relocated equivalents, the length of the function will not be likely the same. In this sense, ARMPatch keeps track of each instruction as it is rewritten, creates a mapping between the original and the new offsets, and finally fixes internal references after the relocation of the external ones is completed.

Algorithm: Pseudo-code to relocate external references & fix internal references

```
F: Function, M: Map, ID: Index List
function RelocateFunction(F)
    Fnew = empty // Relocated function
    Moff = empty // Mapping between original & new offsets
    IDitn = empty // Indexes of internal reference instructions
    offOrig = 0
    offNew = 0
    for each i in F
        if IsExternalRef(i)
            is = RelocateInstructionExternal(i)
            Fnew.add(is)
            Moff[offOrig] = offNew
            offOrig + = len(i)
            offNew + = len(is)
        else
            Fnew.add(i)
            Moff[offOrig] = offNew
            if IsInternalRef(i)
                IDitn.add(offNew)
            end if
            offOrig + = len(i)
            offNew + = len(i)
        end if
    end for
    for each id in IDitn
        i = Fnew[id]
        Fnew[id] = RelocateInstructionInternal(Moff, i)
    end for
     return Fnew
end function
```

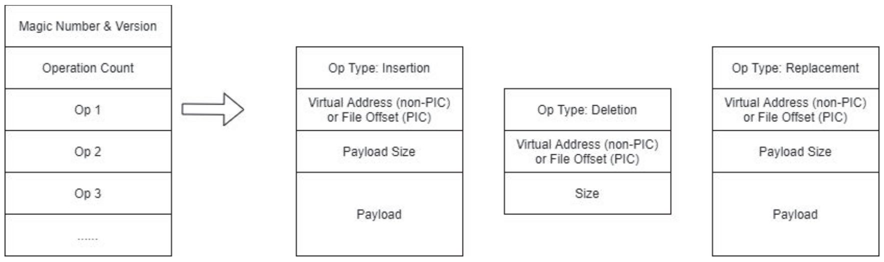


Figure 8 Function diff patch file format.

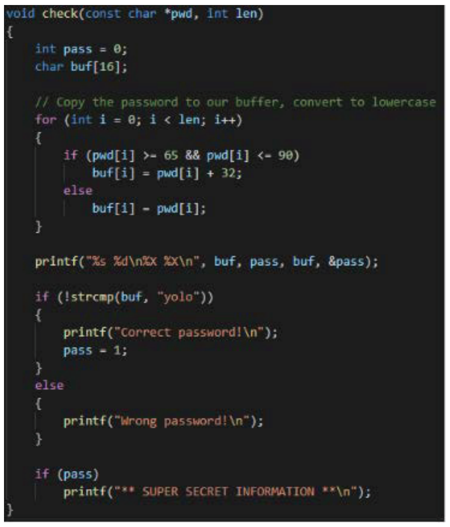
Data Replacement: Constants defined in the source code (i.e., numbers, strings, structures) become data in the segments of a binary after compilation. Sometimes they can be the cause of bugs or security vulnerabilities. ARMPatch provides a functionality to overwrite the original data (located by symbol or offset) with the supplied data, supporting common primitive data types (such as signed and unsigned char, short, integer, long, float, double), or data structures composed with these primitive types. Of course, the length of the data cannot be changed, since it would either change the location of the data that follows, breaking the references; or overwrite it. In fact, it generally does not make sense to vary the length of the data without also changing the way the code loads it.

Function Diff Patch, ARMPatch also allows applying patches to functions in the original binary by adding, removing, or altering instructions. The function "diff patch file format" is a collection of operations that are sequentially performed on the target function; as opposed to the patch config, which modifies the entire binary. In this case, it can be manually crafted or generated programmatically via "binary diff" algorithms.

Since the diff patch can change the length of the function and thus, the internal layout of the instructions, the modified function may need to be moved to a new segment whether the new length exceeds the available space in the original location, just like than the function replacement operation mentioned above. Similarly, it is subject to the relocation process to fix internal and external references.

A typical use case for this functionality is to add parameter validations to buggy or vulnerable functions. Consider the code as follows:

As shown, the original code contains a classic buffer overflow vulnerability that is trivially exploitable by passing any password longer than 16 bytes, so the flag on the stack can be overwritten with a non-zero value, exposing the "super-secret information" without a correct password. In order to fix it, it



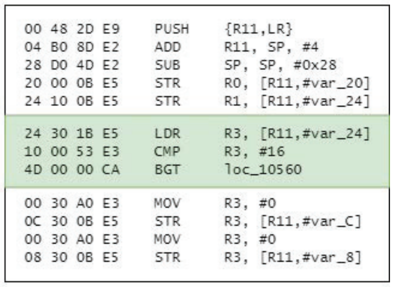
```c
void check(const char *pwd, int len)
{
    int pass = 0;
    char buf[16];

    // Copy the password to our buffer, convert to lowercase
    for (int i = 0; i < len; i++)
    {
        if (pwd[i] >= 65 && pwd[i] <= 90)
            buf[i] = pwd[i] + 32;
        else
            buf[i] = pwd[i];
    }

    printf("%s %d\n%X %X\n", buf, pass, buf, &pass);

    if (!strcmp(buf, "yolo"))
    {
        printf("Correct password!\n");
        pass = 1;
    }
    else
    {
        printf("Wrong password!\n");
    }

    if (pass)
        printf("** SUPER SECRET INFORMATION **\n");
}
```



```
00 48 2D E9    PUSH    {R11,LR}
04 B0 8D E2    ADD     R11, SP, #4
28 D0 4D E2    SUB     SP, SP, #0x28
20 00 0B E5    STR     R0, [R11,#var_20]
24 10 0B E5    STR     R1, [R11,#var_24]

24 30 1B E5    LDR     R3, [R11,#var_24]
10 00 53 E3    CMP     R3, #16
4D 00 00 CA    BGT     loc_10560

00 30 A0 E3    MOV     R3, #0
0C 30 0B E5    STR     R3, [R11,#var_C]
00 30 A0 E3    MOV     R3, #0
08 30 0B E5    STR     R3, [R11,#var_8]
```

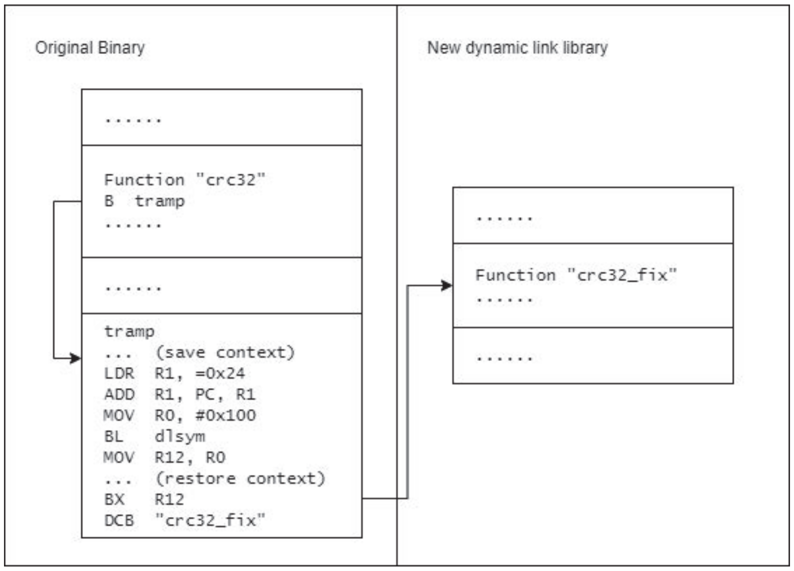
Figure 9 An example of the function “diff patch”.

would be necessary to insert three instructions at the beginning of the function to check the length and jump directly to the failed case if it exceeds the size of our buffer.

Although the function “diff patch” fulfill its task, it is somewhat limited in that it only provides shellcode-like instruction snippets, not complete functions within other binaries. As a result, they lack context and it is impossible for ARMPatch to resolve external references as in function replacement. Therefore, introducing complex logic with this functionality is not practical.

Dependency Injection: When the original code is too complex or runtime modifications/instrumentations are desired, the new code would depend on a lot of new data and functions. In these cases, it is preferable to compile the code into a dynamic link library to be loaded and tunned together with the program. ARMPatch can do this by inserting new DT_NEEDED entries into the binary header. Again, if the size exceeds, the same technique used in function replacement operations is applied to add segments (i.e., moving the header to the end of the binary).

For this new dynamic link library to be useful, we still need some way to redirect some original functions to the new implementations. A possible solution is to create runtime hooks on the original functions during the initialization paths of the library. Nevertheless, this solution is not convenient because it would require additional code to make runtime changes



```
......
Function "crc32"
B  tramp
......
......
tramp
...  (save context)
LDR  R1, =0x24
ADD  R1, PC, R1
MOV  R0, #0x100
BL   dlsym
MOV  R12, R0
...  (restore context)
BX   R12
DCB  "crc32_fix"
```

```
......
Function "crc32_fix"
......
......
```

Figure 10 “dlsym” workaround.

to the instructions. Moreover, since PIC is very popular in dynamic link libraries [17], it is also impossible to statically patch instructions from the original functions to jump to fixed addresses of the dynamic link library, as they are not known before being loaded by the linker.

To solve these issues, we provide a workaround using “dlsym”. ARMPatch inserts a piece of code that (i) saves the parameters, (ii) locate the new implementation in the library by its symbol using “dlsym”, and (iii) calls it; into each of the original functions that need to be replaced.

A limitation of this solution is that the original binary must have already imported “dlsym”, as we currently do not support adding new relocation entries, GOT, and PLT. Furthermore, the workaround needs to call “dlsym” whenever a symbol must be located, which can cause performance overhead if it is used frequently. As a future research line, we plan to use cache to avoid this issue.

Evaluation: In order to validate our approach and demonstrate the versatility of our framework, we tested ARMPatch on two typical ARM devices: (1) a wireless home router, and (2) an Android phone. The router is an ASUS RT-ACRH13 equipped with the Qualcomm Atheros IPQ4028 SoC, running a build of OpenWrt 19.07. It has a quad-core ARM Cortex A7 CPU clocked at 717MHz and a 128MB flash memory, which we consider representative of mid-to-high-end IoT devices. On the other hand, the Android phone is a OnePlus 7 Pro with Qualcomm Snapdragon 855 SoC technology, running the Android 10 stock ROM. It has an octa-core ARM Cortex A76 CPU clocked at 2.84 GHz and an 8GB RAM, which is a typical Android phone configuration.

Case Study #1: Vulnerability Patching and Bug Fixing with Function Replacement and Diff. First, we tested the patching of buggy and vulnerable functions both by writing our own examples and using real-world instances that have existed in real programs. Based on the statistics we showed above, many fixes only involved changes to specific functions. Thus, we selected cases where issues could be fixed using function replacements or diffs, programmed them as ARMPatch patches, and measured whether our framework was able to fix them successfully. Moreover, we measured the space overhead of our solutions.

As shown in the table above, ARMPatch successfully fixed all the cases we tested. Although the space overhead is relatively large when the original binary is small (because it needs to move the ELF header, insert new segments, and cope with memory alignment requirements), it becomes less significant as the input binary increases its size.

Case Study #2: Vulnerability Patching with multiple operations combined. In cases where a patch involves more changes and additions (e.g., adding new functions, constants, strings, etc.), a simple function replacement is not enough. For the second case study, we took some random samples of security patches from open-source projects and evaluated whether they could be implemented with ARMPatch. Again, measuring how much space overhead it would introduce if it succeeded, and what the obstacles would be if it did not.

As shown, ARMPatch was able to cope with two of the three problems we tested above. However, it had difficulties with a patch of “libcurl” that involved a change in data structures. We think this is one of the major limitations of ARMPatch at the moment. If a data structure that is repeatedly used in the code changes, many functions throughout the binary change automatically, so it is unrealistic to apply binary-level patches. A workaround for this issue would be to recompile the entire library as a dynamic link library, then use Dependency Injection to replace the one that is statically linked in the binary. This is still a very cumbersome and delicate process, which is rarely practical.

Limitations: Even though we believe that ARMPatch has demonstrated its ability to work as a comprehensive binary-patching framework, it presents some limitations that are discussed below. While some of the remaining

Table 3 List of cases tested with function replacement and diff

Project	Source	Description	Successful	Space Overhead
Demo: CRC32	N/A	A CRC32 calculation tool with an incorrect implementation of the CRC32 algorithm. We replaced the algorithm with the correct one.	Yes	12KB to 21KB (75% increase) (The original binary is too small to be representative)
Demo: Password Check	N/A	A program that verifies passwords from user input. Contains a buffer overflow vulnerability. We fixed the vulnerability by patching the check function.	Yes	11KB to 21KB (90% increase) (The original binary is too small to be representative)
uhttpd	CVE-2019-19945	Invalid data access can be triggered with an HTTP POST request, causing out-of-bounds memory reads.	Yes	41KB to 48KB (17% increase)
dnsmasq	CVE-2020-25681	A heap-based buffer overflow allows attackers to write arbitrary data in a heap memory segment, possibly executing code on the machine.	Yes	192KB to 202KB (5% increase)
wget	CVE-2019-5953	A buffer overflow allows remote attackers to cause a denial-of-service (DoS) attack or execute arbitrary code via unspecified vectors.	Yes	283KB to 290KB (2% increase)

Table 4 List of cases tested with multi-operation patches

Project	Source	Description	Changes Involved	Possible with ARMPatch	Space Overhead
aria2c	CVE-2019-3500	When –log is used, it stores an HTTP Basic Authentication username and password in a file, which might allow local users to steal sensitive information	FC, DA	Yes	1.20MB to 1.21MB (0.8% increase)
OpenSSH	CVE-2016-1908	Mishandles failed cookie generation for untrusted X11 forwarding and relies on the local X11 server for access-control decisions, which allows remote X11 clients to trigger a fallback and obtain trusted X11 forwarding privileges	FC, FA, DA	Yes	720KB to 728KB (1% increase)
libcurl	CVE-2018-1000007	Might accidentally leak authentication data to third parties	FC, DA, SC	No. This patch involves a change in the definition of a data structure, resulting in many function changes. We consider that it is not practical to use ARMPatch in this case.	N/A

issues are minor imperfections that can be easily enhanced in the future, we believe others are intrinsic to the currently available binary disassembly/analysis techniques.

Complex PC-based Pointer Arithmetic: ARMPatch currently uses simple static analysis to determine whether a PC-relative reference points to an external or internal target. Although we found this to be sufficient for ordinary binaries in our tests (compilers generally do not produce code that performs complex PC arithmetic), it is possible to have code that is difficult or impossible to perform correct static analysis. In such cases, ARMPatch will not be able to relocate the code correctly, resulting in errors executing relevant instructions.

Self-modifying obfuscated/virtualized code: As in the previous problem, ARMPatch it is not able to recognize the code that cannot be statically analyzed. Some proprietary software use techniques to obfuscate its code and protect it from being cracked or reverse engineered. Fortunately, this is rarely a problem in open-source projects, and IoT devices scarcely employ these techniques due to space and performance limitations.

Function recognition: As part of the steps to fix external references, ARMPatch looks for the equivalent external functions called by the new code in the original binary. In the current release, this is achieved by searching for the exact instructions. While this guarantees correctness, it often fails to match all in many cases. Depending on the compilers and parameters used, the generated instructions can be drastically different even if the source code is identical. This can be improved in the future by adopting more advanced function recognition techniques.

Conclusion

Despite the rapid growth of IoT devices, lack of transparency and timely updates cause many security risks. The absence of a static binary patching solution suitable for vulnerability mitigation on ARM platforms also makes it difficult for third parties to fix the issues themselves. In this study, we analyze the specific needs to correct vulnerabilities in firmware of IoT devices (without source code) using binary patching. We also explored and overcome the involved technical challenges, which resulted in the development of ARMPatch. Our proposal was validated with real-world cases, which successfully solved 7 of the 8 vulnerabilities presented. Therefore, we believe that ARMPatch has proven its effectiveness and versatility in fixing security vulnerabilities on ARM platforms, which we hope will be valuable for further research in the field.

Declarations

Availability of Data and Materials

All data and materials used in the current study are available from the corresponding author on reasonable request.

Ethics Approval and Consent to Participate

Not applicable

Consent for Publication

Not applicable

Competing Interests

The authors declare that they have no competing interests.

Authors' Contributions

Chengyu Song guided the whole work; Mingyi Huang and Chengyu Song developed all models; All authors read and approved the final manuscript.

References

[1] Mordor Intelligence, "internet of things (IoT) market – growth, trends, covid-19 impact, and forecasts (2021 – 2026)," 2020.

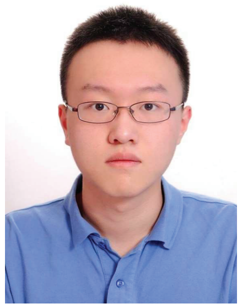
[2] Mahmoud, R., T. Yousuf, F. Aloul and I. Zualkernan, "Internet of things (IoT) security: Current status, challenges and prospective measures," in 2015 10th International Conference for Internet Technology and Secured Transactions (ICITST), London, UK, 2015.

[3] Statista, "Arm's market share and targets across key technology markets in 2019 and 2028 fiscal years," August 2020. https://www.statista.com/statistics/1132112/arm-market-share-targets/.

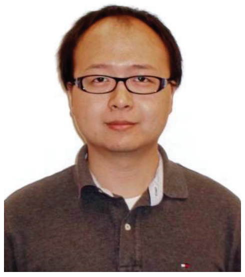
[4] M. Laurenzano, "Fast static binary instrumentation for linux/x86," https://github.com/mlaurenzano/PEBIL.

[5] D. Project, "DyninstAPI: Tools for binary instrumentation, analysis, and modification," https://github.com/dyninst/dyninst.
[6] GJDuck, "E9Patch – A Powerful Static Binary Rewriter," https://github.com/GJDuck/e9patch.
[7] utds3lab, "Multiverse, a static binary rewriter with an emphasis on simplicity and correctness," https://github.com/utds3lab/multiverse.
[8] Hewlett-Packard, "Dynamic Instrumentation Tool Platform". https://dynamorio.org/.
[9] Valgrind Developers "Valgrind: an instrumentation framework for building dynamic analysis tools," https://www.valgrind.org/.
[10] O. A. V. Ravnås, "Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers," https://frida.re/.
[11] Benjamin, S., D. Saumya and A. Gregory, "Disassembly of executable code revisited," in 9th Working Conference on Reverse Engineering, WCRE 2002, Richmond, 2002.
[12] Andriesse, D., X. Chen and V. v. d. Veen, "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries," in The 25th USENIX Security Symposium, Austin, 2016.
[13] "OpenWrt Project" https://openwrt.org/.
[14] Jiang, M., Y. Zhou, X. Luo, R. Wang, Y. Liu and K. Ren, "An empirical study on ARM disassembly tools," Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, 2020.
[15] ARM, ARM Compiler toolchain Assembler Reference, 2011.
[16] Duan, Y., X. Li, J. Wang and H. Yin, "DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing," *NDSS Symposium*, San Diego, 2020.
[17] Göktas, E., B. Kollenda, P. Koppe, E. Bosman, G. Portokalidis, T. Holz, H. Bos and C. Giuffrida, "Position-Independent Code Reuse: On the Effectiveness of ASLR in the Absence of Information Disclosure," *IEEE European Symposium on Security and Privacy (EuroS&P)*, London, UK, 2018.

Biographies



Mingyi Huang is a master's student in the Department of Computer Science and Engineering at University of California, Riverside. His research mainly focuses on network protocol & operating system security.



Chengyu Song is an Assistant Professor in the Department of Computer Science and Engineering at University of California, Riverside. He earned by Ph.D. in Computer Science from Georgia Tech, and was fortunate to be supervised by professor Wenke Lee and Taesoo Kim. His research interests include system security, program analysis and verification, and operating systems. His current research focuses on vulnerability related topics, including:

Advancing techniques for finding vulnerabilities in binaries, OS kernels, machine learning, and cyber-physical systems.
Eliminating vulnerabilities through automatic patch generation and verification.
New exploit techniques and automated exploit generation.
Runtime exploit prevention with software hardware co-design.