arXiv:2105.14295v2 [cs.AR] 5 Oct 2021

ECMO: Peripheral Transplantation to Rehost Embedded Linux Kernels

A Preprint

Muhui Jiang[1,2], Lin Ma[1], Yajin Zhou*[1], Qiang Liu[1], Cen Zhang[3], Zhi Wang[4], Xiapu Luo[2], Lei Wu[1], and Kui Ren[1]

[1]*Zhejiang University*
[2]*The Hong Kong Polytechnic University*
[3]*Nanyang Technological University*
[4]*Florida State University*

October 6, 2021

Abstract

Dynamic analysis based on the full-system emulator QEMU is widely used for various purposes. However, it is challenging to run firmware images of embedded devices in QEMU, especially the process to boot the Linux kernel (we call this process rehosting the Linux kernel in this paper). That's because embedded devices usually use different system-on-chips (SoCs) from multiple vendors and only a limited number of SoCs are currently supported in QEMU.

In this work, we propose a technique called *peripheral transplantation*. The main idea is to transplant the device drivers of designated peripherals into the Linux kernel binary. By doing so, it can replace the peripherals in the kernel that are currently unsupported in QEMU with supported ones, thus making the Linux kernel *rehostable*. After that, various applications can be built.

We implemented this technique inside a prototype system called ECMO and applied it to 815 firmware images, which consist of 20 kernel versions and 37 device models. The result shows that ECMO can successfully transplant peripherals for all the 815 Linux kernels. Among them, 710 kernels can be successfully rehosted, i.e., launching a user-space shell (87.1% success rate). The failed cases are mainly because the root file system format (*ramfs*) is not supported by the kernel. Meanwhile, we are able to inject rather complex drivers (i.e., NIC driver) for all the rehosted Linux kernels by installing kernel modules. We further build three applications, i.e., kernel crash analysis, rootkit forensic analysis, and kernel fuzzing, based on the rehosted kernels to demonstrate the usage scenarios of ECMO.

1 Introduction

IoT devices (or embedded devices) are becoming popular [7], many of which run Linux-based operating systems [30]. At the same time, hundreds of vulnerabilities are discovered every year for the Linux kernel [22]. Once the devices are compromised, attackers can control them to launch further attacks. As such, the security of embedded devices, especially the kernel, deserves a thorough analysis.

Dynamic analysis has been widely used for various purposes [41, 31, 44, 51, 70, 47]. It can monitor the runtime behavior of the target system, complementing the static analysis [55, 61, 30, 45]. Rehosting, also known as emulation, is used to run a target system inside an emulated environment, e.g., QEMU, and provides the capability to introspect the runtime state. Based on this capability, different applications, e.g., kernel crash analysis, rootkit forensic analysis, and kernel fuzzing, can be built upon. Running the Linux kernel in QEMU for the desktop system is a solved problem. However, embedded systems usually have different system-on-chips (SoCs) with customized hardware peripherals from multiple vendors. Due to the diverse peripherals in the wild, it is not practical for QEMU to support all kinds of

*Corresponding author (yajin_zhou@zju.edu.cn).

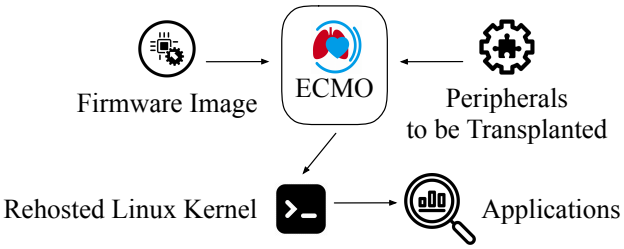


Figure 1: The overview of our system (ECMO)

peripherals in any SoC, which is tedious and error-prone. In this case, embedded Linux kernels, which depend on the unsupported peripherals, may stuck, halt, or crash during the rehosting process. Thus, how to rehost the embedded Linux kernels in QEMU is still an open research question.

Previous research [26, 50] provides the capability of rehosting user-space programs by running a customized Linux kernel for a single type of SoC that is already supported in QEMU. This works well because user-space programs mainly depend on standard system calls that are provided by the underlying Linux kernel. Different from user-space programs, the OS kernel (e.g., Linux kernel) interact with hardware peripherals that are usually different in different SoCs.

Some researchers have proposed to use real devices to perform the dynamic analysis [69, 54, 43, 62]. Such solutions do not scale since there exist a large number of embedded devices. Other researchers work towards the bare-metal systems [37, 52, 28], i.e., embedded systems without an OS kernel or having a thin layer of abstraction. However, the methodology cannot be directly used to rehost the Linux kernel as the Linux kernel is far more complicated than the bare-metal ones.

Our Approach In this work, we propose a solution called *peripheral transplantation. It is device-independent, and works towards the Linux kernel binary without the need of the source code of the target system*. The main idea is, instead of manually adding emulation support of various peripherals in QEMU, we can transplant the device drivers of designated peripherals into the target Linux kernel binary. It replaces the peripherals in the target Linux kernel that are currently unsupported in QEMU with supported ones, thus making the Linux kernel *rehostable*. In particular, given a Linux kernel retrieved from the firmware image of an embedded device, our system turns it into a rehostable one that can be successfully booted in QEMU. After that, various applications can be built to analyze the rehosted kernel.

Specifically, our system transplants two components, i.e., the emulated models of peripheral into QEMU and their device drivers into the Linux kernel. Transplanting a peripheral model requires us to build the hardware emulation code for the specified (or simplified) peripheral and integrate it into QEMU. This is straightforward Since QEMU has provided us with APIs to add new peripheral models.

However, transplanting a driver into the Linux kernel binary is non-trivial since we need to achieve three design goals. *First*, we need to substitute the original (unsupported) device driver with the transplanted one. Since the peripheral driver is initialized with indirect calls in Linux kernel, we need to locate functions and rewrite the function pointers in a stripped binary on the fly, which is challenging. *Second*, the transplanted driver should not affect the memory view of the original kernel. Otherwise, the memory holding the transplanted driver can be overwritten since the Linux kernel is not aware of the existence of that memory region. *Third*, the transplanted driver needs to invoke APIs in the Linux kernel. Otherwise, the transplanted driver cannot function as desired.

To address these challenges, we implement and integrate the *peripheral transplantation* technique into QEMU to create a prototype called ECMO. We detail the system design and implementation in Section 4. Figure 1 shows the overview of ECMO. It receives the firmware image and the peripherals to be transplanted. Then it transplants the peripherals to the Linux kernel binary to make it rehostable in QEMU and launch a shell. Then we can build various applications on it.

We apply our system on 815 Linux kernel binaries extracted from firmware images, including 20 different kernel versions, 37 device models, and 24 vendors. Our experiment shows that ECMO can successfully transplant peripherals for all 815 Linux kernels. Among them, 710 are able to boot and launch a shell. The failed cases are due to the unsupported root file system format (*ramfs*) in the rehosted kernel. To demonstrate the functionality and usefulness of our system, we build and port three applications, including kernel crash analysis, rootkit forensic analysis, and kernel fuzzing. Note that, the applications themselves are not the contribution of our work. They are used to demonstrate the usage scenarios of our system. Other applications that can be built on QEMU can also be ported.

```
MACHINE_START(VERSATILE_AB, "ARM-Versatile␣AB")
    .atag_offset = 0x100,
    .map_io = versatile_map_io,
    .init_early = versatile_init_early,
    .init_irq = versatile_init_irq,
    .init_time = versatile_timer_init,
    .init_machine = versatile_init,
    .restart = versatile_restart,
MACHINE_END
```

Figure 2: The machine description for *ARM-Versatile AB*.

```
//UART read call back
static uint64_t serial_mm_read(void *opaque,
    hwaddr addr, unsigned size) {
  SerialMM *s = SERIAL_MM(opaque);
  return serial_ioport_read(&s->serial,
                            addr >> s->regshift, 1);
}
//register read/write call back functions
static const MemoryRegionOps serial_mm_ops = {
  .read = serial_mm_read,
  .write = serial_mm_write,
  ...
};
```

Figure 3: The callback functions for UART emulation in QEMU

In summary, this work makes the following main contributions.

Novel technique We propose a *device-independent* technique called *peripheral transplantation* that can rehost Linux kernels of embedded devices without the availability of the source code.

New system We implement and integrate the *peripheral transplantation* technique into QEMU, to create a prototype system called ECMO.

Comprehensive evaluation We apply ECMO to Linux kernels from 815 different images. It can transplant peripherals for all the Linux kernels and successfully launch the shell for 710 ones.

To engage with the community, we will release the source code of our system and the dataset of firmware images. We have provided a docker image [6] that contains the ECMO system for testing.

2 Background

2.1 Linux Kernel

Linux kernel source code can be categorized into three types according to their functionalities. The first type is the *architecture independent code*, which contains the core functionality used by all CPU architectures. The second type is *architecture dependent code*. For instance, the sub-directories under the *arch/* directory contain the code for multiple CPU architectures. The third type is *board-specific code*, which is used by specific board (machine). For instance, the directory *arch/arm/versatile/* contains the code used by the machine named *versatile*. The kernel compiled for one machine usually cannot be directly booted on other machines (or QEMU instances that emulate different machines.)

2.2 ARM Machines

Embedded systems usually use SoCs from multiple vendors with different designs. For instance, they contain different peripherals. Each SoC is expressed as a machine in the Linux kernel. Manufacturers develop the *board support package* (BSP) (e.g., drivers of peripherals) so that Linux kernel can use these peripherals.

Linux kernel introduces the structure *machine_desc* for ARM to describe different machines. The structure *machine_desc* provides interfaces to implement BSPs. For example, Figure 2 shows an example of one machine *ARM-Versatile AB* in the Linux kernel (Version 3.18.20). It initializes function pointers and data pointers with its implementation. Specifically, in line 5, the function pointer *init_irq* is assigned the value as *versatile_init_irq*. During the booting process, the Linux kernel will invoke the function *machine_desc→init_irq* to initialize the IC (interrupt controller). The same logic applies to the function pointer *init_time*. Linux kernel invokes the function *machine_desc→init_time* to initialize the timer.

2.3 QEMU

QEMU [16] is one of the most popular full-system emulators. It emulates different machines by providing different machine models. A machine model consists of CPU, memory, and different kinds of peripheral models. To emulate a peripheral, QEMU registers the read/write callback functions for the MMIO (memory-mapped I/O) address space of the peripheral. Once the Linux kernel running inside QEMU reads from or writes into the address inside the MMIO range, the registered callback functions inside QEMU will be invoked to emulate the peripheral. Basically, it maintains an internal state machine to implement the peripheral's functionality. Figure 3 shows an example of the registered callback functions for UART emulation. Specifically, when the Linux kernel reads from the MMIO space of the emulated UART device (e.g., `0x01C42000`), the `serial_mm_read` function will be invoked by QEMU to emulate the read access.

3 Challenges and Our Solution

The main goal of our work is to rehost Linux kernel binaries that are originally running on embedded systems in QEMU. This lays the foundation of applications that rely on the capability to introspect runtime states of the Linux kernel, e.g., kernel crash and vulnerability analysis [41, 31], rootkit forensic analysis [64, 56], and kernel fuzzing [51, 59].

3.1 Challenges

Rehosting the Linux kernel on QEMU faces the following challenges.

Peripheral dependency Rehosting the Linux kernel requires QEMU to emulate the peripherals, e.g., the interrupt controller, that the Linux kernel depends on. During the booting process, Linux kernel will read from or write into the peripheral registers and execute the code according to the state specified by the value of peripheral registers. Without the emulation of these peripherals, the rehosted kernel will halt or crash during the booting process.

Peripheral diversity SoCs vary widely [19] and different vendors, e.g., Broadcom, Marvell may design and develop different SoCs. These new SoCs introduce many new peripherals that are not currently supported in QEMU and the open-sourced mainstream of the Linux kernel. Due to the diversity of peripherals, there are still a large number of devices that are not supported. Meanwhile, manually developing peripheral emulation routine is tedious and error-prone, especially due to the diversity of peripherals. Thus, the diversity of peripherals brings significant challenge to build a general emulator, which can re-host various Linux kernels of embedded devices.

Lack of public information The information (e.g., specifications, datasheets, and source code) of SoCs and firmware images are usually not public. This is because vendors may not release the detailed hardware specification. Furthermore, vendors may not release the source code immediately after releasing the image and not all vendors strictly follow the GPL license [42, 35]. Meanwhile, the binary of the Linux kernel is stripped and has no particular headers (i.e., ELF section headers) or debugging information. These obstruct the diagnosis of failures when adding emulation support of new SoCs in QEMU.

3.2 Our Solution: Peripheral Transplantation

In this work, we propose a technique called *peripheral transplantation*. The main idea is, instead of manually adding emulation support of various peripherals in QEMU, we can *replace the peripherals that are used in target Linux kernels with existing peripherals in QEMU*. By doing so, we can rehost the Linux kernel and the kernel functionality is intact (Section 5.4).

Figure 4 shows the overview of peripheral transplantation. This involves the injection of peripheral models into QEMU and the *ECMO Driver* into the Linux kernel. To distinguish them from original ones of the (emulated) machine, we call the transplanted peripheral models *ECMO Peripheral*. To let the kernel use the transplanted *ECMO Driver*, our system identities the functions that are used to initialize device drivers (*ECMO Forward Pointers*) and redirects them to the functions inside the *ECMO Driver* (Fig. 4 ❶). Moreover, our system identifies the APIs that are responsible for interacting with peripheral models. These APIs are used by the *ECMO Driver* to communicate with the transplanted peripheral models (Fig. 4 ❷). The addresses of these functions are called *ECMO Backward Pointers* in this paper. We will elaborate how to identify the ECMO Pointers in Section 4.2.

Note that, to ensure the *ECMO Driver* does not affect the memory view of the rehosted Linux kernel, we propose the concept of the *opaque memory*. This memory region is available on the emulated machine but cannot be seen by

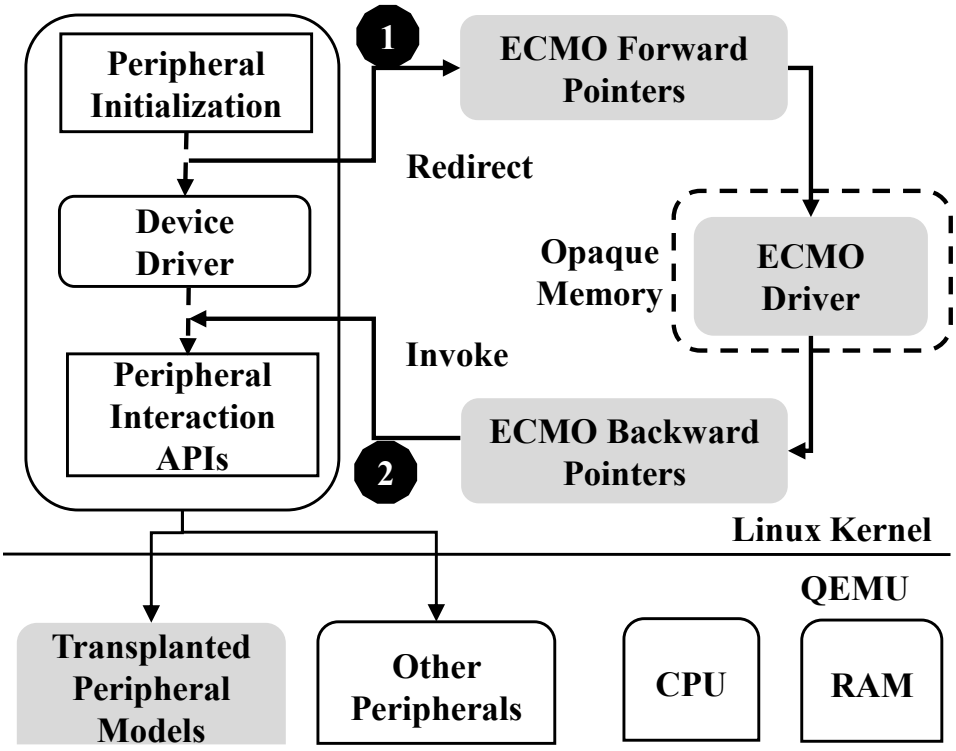


Figure 4: The overview of peripheral transplantation.

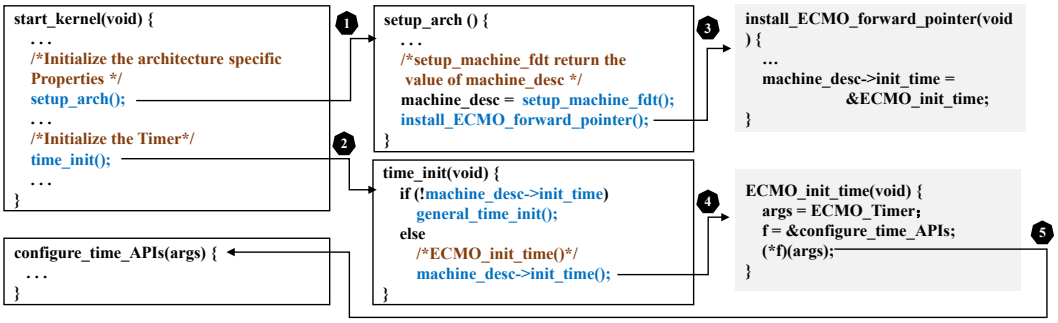


Figure 5: A concrete example of peripheral transplantation.

the Linux kernel. As such, we can prevent the kernel from allocating memory pages that are reserved for the *ECMO Driver*. We will elaborate this in Section 4.3.

3.3 An Illustration Example of Peripheral Transplantation

Fig. 5 shows a concrete example of transplanting one peripheral (i.e., timer) into the Linux kernel. In particular, the function `start_kernel` is responsible for initializing the Linux kernel. It will invoke several different functions, including `setup_arch` and and `time_init`.

The function `setup_arch` will setup architecture-related configurations and initialize the `machine_desc` structure (Fig. 5 ❶). This structure contains multiple function pointers (*ECMO Forward Pointers*) that will be used to initialize corresponding drivers. Our system first locates the function `setup_arch` and then injects a function (`install_ECMO_forward_pointers`) to change the pointers to our own ones (Fig. 5 ❸).

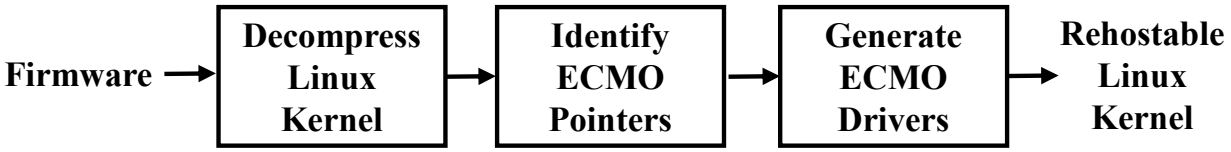


Figure 6: The work flow of our system.

```
Assembly code:
  mov r0, #0
  str r0, [r2], #4
  str r0, [r2], #4
  str r0, [r2], #4
  str r0, [r2], #4
  cmp r2, r3
  blo 1b
  tst r4, #1
  bic r4, r4, #1
  blne  cache_on
  mov r0, r4 //r0 stores the value of output_start
  mov r1, sp
  add r2, sp, #0x10000
  mov r3, r7
  bl  decompress_kernel
  // we can dump the decompressed Linux kernel after
  // function decompress_kernel returns

Simplified C code:
  void decompress_kernel(uint32 output_start, args)
```

Figure 7: The assembly code that invokes function *decompress_kernel*, which is in *arch/arm/boot/compressed/head.S*.

When the function `init_time` is invoked to initialize the timer (Fig. 5 ❷), the `ECMO_init_time`, which is pointed by `machine_desc->init_time`, will be invoked to initialize the injected timer driver (*ECMO Driver*) in QEMU (Fig. 5 ❹) (through *ECMO Forward Pointers*), instead of the original one. Accordingly, this function will invoke APIs (through *ECMO Backward Pointers*) in the Linux kernel to interact with the *ECMO Peripheral* (Fig. 5 ❺).

Note that, the code snippets in Fig. 5 are for the illustration purpose. *Our system does not rely on the availability of the source code. It directly works towards the Linux kernel binary that is retrieved from a firmware image.*

4 System Design and Implementation

In order to rehost Linux kernels, our system first extracts and decompresses the Linux kernel from the given firmware image (Section 4.1). We then apply multiple strategies to identify both *ECMO Forward and Backward Pointers* (Section 4.2). These pointers are essential for *ECMO Drivers*. At last, we semi-automatically generate *ECMO Drivers* and load them at runtime to boot the kernels (Section 4.3). Fig. 6 shows the overall workflow.

4.1 Decompress Linux Kernel

Firmware image usually consists of the OS, which is the Linux kernel, and user applications. However, the Linux kernel inside the firmware images is usually in the compressed zImage format. To identify ECMO Pointers inside the Linux kernel, we need to first extract the Linux kernel and decompress it. With the decompressed Linux kernel, we can utilize different strategies to locate the ECMO Pointers.

Specifically, we feed the firmware image to firmware extraction tool (i.e., Binwalk) to extract the kernel image. Then we directly feed the extracted kernel image (with added u-boot information) to QEMU. Since the code inside the zImage to decompress the Linux kernel does not operate on the peripherals (except the UART to show the message of decompressing Linux kernel), it can be successfully executed in vanilla QEMU.

As shown in Fig. 7, function `decompress_kernel` is used to decompress the kernel. Its first parameter (i.e., `output_start`) indicates the start address of the decompressed kernel. Thus, if we can identify when `decompress_kernel` is invoked, we can get the first parameter by checking the machine register (`R0` in ARM) and dump the decompressed Linux kernel.

We notice that the function *decompress_kernel* is invoked by the assembly code in *arch/arm/boot/compressed/head.S*. We observe that this snippet of assembly code remains unchanged in different kernel versions. With this observation, we identify the address of instruction `BL decompress_kernel` by strictly comparing the execution trace of QEMU and the hard coded assembly code. After finding the instruction, we can obtain the address of the function *decompress_kernel* and the value of *output_start* according to the execution trace. With this information, we can dump the decompressed Linux kernel after the function *decompress_kernel* returns.

By doing so, we can automatically retrieve decompressed Linux kernels from firmware images.

4.2 Identity ECMO Pointers

Our system needs to obtain the addresses of two essential types of functions in the Linux kernel. Specifically, the *ECMO Forward Pointers* contain the functions that are used by the Linux kernel to initialize device drivers. We dynamically hook and redirect them to *ECMO Drivers* at runtime in QEMU. The *ECMO Backward Pointers* contain the APIs that are used by the *ECMO Driver* to invoke functions provided by the Linux kernel to interact with emulated peripherals in QEMU.

However, precisely identifying ECMO Pointers is not easy. The main challenge is the decompressed Linux kernel is stripped and only contains the binary data. It has neither meaningful headers nor debugging symbols and contains thousands of functions. Furthermore, the Linux kernel is compiled with different compilers and compiling options, which can result in different binaries. Thus, we cannot have any assumption on the compiling options or compilers. We also cannot rely on run-time symbol tables like `/proc/kallsym` because they are only available *after* booting. To address these problems, we propose following strategies that can automatically identify ECMO Pointers.

The basic idea to identify ECMO Pointers is leveraging the source code of the mainline Linux kernel, which provides valuable information for functions that need to be located. Note ECMO Pointers are functions in *architecture independent code* or *architecture dependent code* (Section 2.1), which is in mainline Linux kernel and open-source. For instance, if we find that a function uses a specific string by reading the source code, then we can easily locate this function inside the binary by locating the function that has references to the same string. Of course, this simple strategy may not always work, since some functions do not have such obvious patterns or multiple functions can refer to the same string. In our work, we take three different strategies to locate ECMO Pointers (Section 4.2.2). We illustrate each step to identify ECMO Pointers in the following.

4.2.1 Disassemble the Linux Kernel

The first step is to disassemble the Linux kernel for further analysis, including constructing the control flow graph and identifying function boundaries. Accurately disassembling the ARM binaries is still challenging, especially when the binary is stripped [46]. In our work, we choose to ensure that this step does not introduce false negatives, i.e., all the code sections should be dissembled. Otherwise, we cannot identify the functions if they are not correctly disassembled. However, we can tolerate the false positives, i.e., the inline data may be wrongly disassembled as code. The strategies described in Section 4.2.2 can help us to filter out these false positives.

After disassembling the Linux kernel and constructing the control flow graphs, we further locate function boundaries by combining the algorithm introduced in Nucleus [23] and angr [1]. Necleus can identify the functions indirectly called while angr locates the function according to the prologue. These two tools can help to reduce the false negatives and guarantee that the required function addresses (ECMO Pointers) will be located during the disassembly process. Finally, we build a mapping for each function and various types of information, e.g., number of basic blocks, string references, number of called functions and etc. This mapping describes the signature (or portrait) of each function. Note that, our system does not require that the constructed control flow graph is sound or complete, as long as they can provide enough information for further analysis (Section 4.2.2).

4.2.2 Locate Pointer Addresses

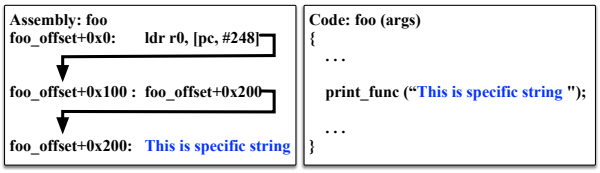
Algorithm 1 describes the process to locate pointer addresses of ECMO Pointers in the decompressed Linux kernel binary, i.e., *LKB*. Note that, we first need to get the source code of the functions, i.e., *SC*, inside the mainline Linux kernel. The outputs of this algorithm are the addresses of ECMO Pointers in the kernel binary, i.e., *FA* (line 12).

First, we disassemble the decompressed Linux kernel, construct the control flow graph (line 2) and generate function boundaries (line 3). Then for the source code function of each ECMO Pointer (line 4), we loop through the generated functions (line 5) and apply different filtering strategies (line 6). If one filtering strategy can identify one address as a candidate address of the ECMO Pointer (line 7), this address will be appended to the candidate list (line 8). Finally,

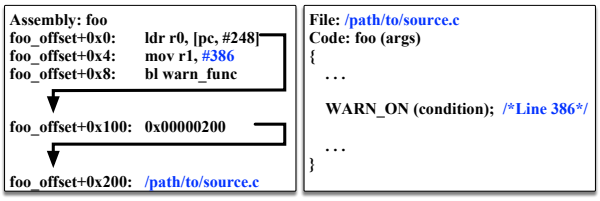
Algorithm 1: The algorithm to identify the addresses of ECMO pointers from the Linux kernel binary.

Input: The decompressed Linux kernel LKB;
The source code of ECMO Pointers SC (architecture independent code or architecture dependent code);
Output: The addresses of ECMO Pointers FA;

```
1  Function Identify(LKB,SC):
2      CFG = Disassembly(LKB)
3      Generated_Functions = GenerateFunctions(CFG)
4      for S_F in SC do
5          for G_F in Generated_Functions do
6              for Filtering_Strategy in Filtering_strategies do
7                  if Filtering_Strategy(S_F,G_F) then
8                      Append G_F to S_F.Candidates
9      for S_F in SC do
10         if Length(S_F.Candidates) == 1 then
11             FA[S_F] = S_F.Candidates
12     return FA
13
```



(a) Specific constant string: the constant string is referenced by a data pointer (i.e., foo_offset+0x200).



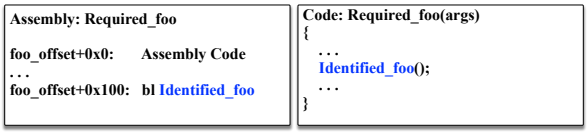
(b) Warning information: line number (i.e., 386) is the operand of assembly code; file name (i.e., /path/to/source.c) is a constant string.

Figure 8: Strategy-I: Lexical information

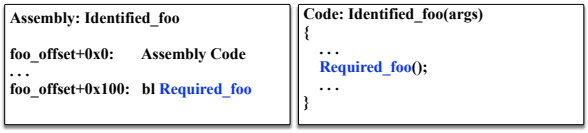
we check the candidates of each ECMO Pointer (line 9). If there is only one candidate (line 10), it means the address of this ECMO Pointer is successfully identified in the kernel binary (line 11).

Strategy-I: Lexical information The first strategy uses the lexical information inside a function as its signature, e.g., a specific constant string and the warning information. If the function we want to identify has such strings, we can then lookup the disassembly code to find the functions that have data references to the same string. The line number and file name in the warning information can further help to locate the function.

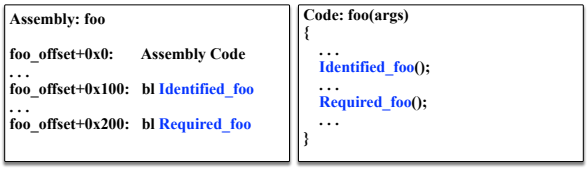
Fig. 8(a) shows a pair of the disassembled code and the source code in the mainline Linux kernel. In the source code, the function `foo` contains a specific constant string "*This is a specific string*". In the assembly code, the instruction at `foo_offset+0x0` will load the data pointers (i.e., `foo_offset+0x100`) using the `LDR` instruction. The data pointer refers to another pointer (i.e., `foo_offset+0x200`), which contains the same constant string. Based on this, we can locate function `foo` in the disassembled kernel. Fig. 8(b) shows a similar example with the warning information. The WARN_ON will call function *warn_func*. The first parameter is the filename, which is a specific constant string. The second parameter is the line number of WARN_ON. Usually, the line number is hard coded as an operand of instruction after compilation. Thus, functions containing specific constant string or warning information can be easily identified.



(a) Caller relationship: Required_foo is the caller of Identified_foo



(b) Callee relationship: Required_foo is the callee of Identified_foo



(c) Sibling relationship: Required_foo and Identified_foo are both called by foo

Figure 9: Strategy-II: Function relationship

Strategy-II: Function relationship The second strategy uses the relationship between functions. That's because functions that do not contain specific strings cannot be identified by the strategy-I. However, we can use the relationship between the functions we want to identify and the ones that have been identified using the previous strategy. For instance, if we have identified the function (`Identified_foo`) and this function is *only* invoked by the function `Required_foo`, then we can easily locate the `Required_foo` by finding the caller of the `Identified_foo` function (Figure 9(a)). Similar strategy can be applied to the callee and sibling relationship, as shown in Figure 9(b) and Figure 9(c), respectively. Thus, we can identify the functions indirectly with the help of function relationship.

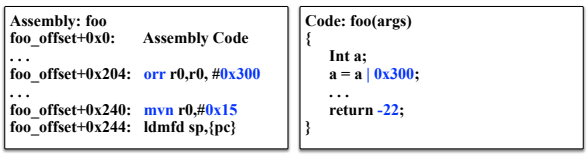
Strategy-III: : Function structure If one function has more than one caller, callee or sibling, it cannot be located solely using the function relationship. The third strategy takes the function structure, including logic or arithmetic operations, return value, the number of basic blocks, and the number of callee functions. Fig. 10(a) shows the example that the function performs the logic operation on some specific values (i.e., `a = a|0x300`) and return a specific value (i.e., -22) , the compiler will generate the instructions that contain the specific values (e.g., `orr r0,r0, #0x300`, `mvn r0,#0x15`). Besides, the callee number and basic block number will also be considered to filter out the candidate. Fig. 10(b) shows that function *foo* has two callees (i.e., *callee_foo_one* and *callee_foo_two*), which map to two instructions at foo_offset+0x18 and foo_offset+0x1c. Basic block number works with the same rule.

Summary With the above three strategies, we can automatically and successfully identify ECMO Pointers for all the Linux kernels (815 ones in 20 kernel versions) used in the evaluation (Section 5.2).

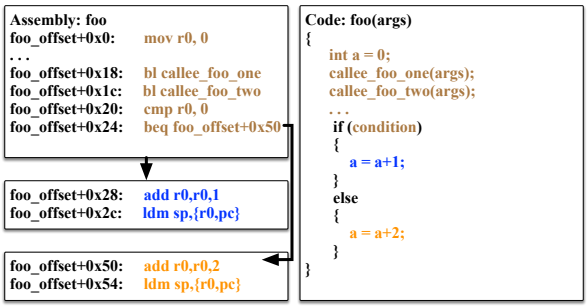
4.3 Generate ECMO Drivers

The process to generate *ECMO Drivers* is similar with developing a kernel module. However, we need to make the driver self-contained as much as possible and invoke the APIs in the Linux kernel through *ECMO Backward Pointers*. In particular, we compile the source code into an object file (i.e., `ECMO_Driver.o`). To make this driver work, we need to setup the base address and fix up the function calls to *ECMO Backward Pointers*. Moreover, we need to ensure that this driver does not occupy the physical memory region that the kernel can perceive, which is achieved by allocating the opaque memory.

Fixup the driver Note that the compiled object file's base address is `0x0`. Given a new load address at runtime, our system calculates new values of the data pointers and function pointers and automatically rewrite the corresponding values in the driver.



(a) Logic operation: The constants (i.e., 0x300, -22) of logic operation or return value in source code map to the operands in assembly code.



(b) Callee Number: The two callee functions (i.e., callee_foo_one, callee_foo_two) map to the two bl instruction at offset foo_offset+0x18 and foo_offset+0x1c. Basic Block Number: The three basic blocks in source code maps to three basic blocks in assembly code.

Figure 10: Strategy-III: Function structure

```
0x10000: ldr r3, [pc, #72]
0x10004: blx r3
0x10050: // Pointer value of called function
```

Figure 11: `ECMO Driver` indirectly invokes functions in Linux kernel. In offset 0x10000, the memory address pointed by [pc, #72] is 0x10000 + 8 + 72 = 0x10050. In this case, functions with arbitrary address can be invoked.

Furthermore, due to the limitation of the jump range for the `BL Label` instruction, the driver may not be able to invoke the functions (*ECMO Backward Pointers*) in the original Linux kernel with direct calls, if the offset between them is far from the range of the `BL` instruction. To make it work, we rewrite the direct calls with indirect calls. For example, Fig. 11 shows a code snippet of the assembly code. At the offset `0x10000`, it loads the value stored at the offset `0x10050` into the register `R3`, which is the jump target. We can rewrite the value in the offset `0x10050` to invoke arbitrary function (*ECMO Backward Pointers*) in the Linux kernel, without being limited by the direct call.

Allocate the opaque memory The *ECMO Driver* is loaded into the memory for execution. As a result, if we directly inject the driver into the physical memory pages that are treated as free page by the rehosted Linux kernel, the pages could be allocated for other purposes. That's because the rehosted kernel does not explicitly know the existence of the *ECMO Driver* code in the memory. Thus, we need to guarantee that the driver should reside inside a memory region that cannot be affected by the Linux kernel.

To solve this problem, we propose the concept of the *opaque memory*, a memory region that is not perceived by the Linux kernel but can be used at runtime. We implement the opaque memory by hooking into the emulated MMU in QEMU. Fig. 12 shows how opaque memory works.

Specifically, the emulated MMU walks through the page table to translate virtual addresses to physical addresses. ECMO changes the MMU module in QEMU to check whether the virtual address being translated is in the region of the opaque memory. If so, it will walk through our hijacked page table for the opaque memory to get the physical address. Otherwise, the original kernel page tables will be used. We ensure that the virtual address in the opaque memory always has a valid entry in the page table. By doing so, the *ECMO Driver* can be loaded and executed in the opaque memory, without affecting the memory view of the rehosted Linux kernel.

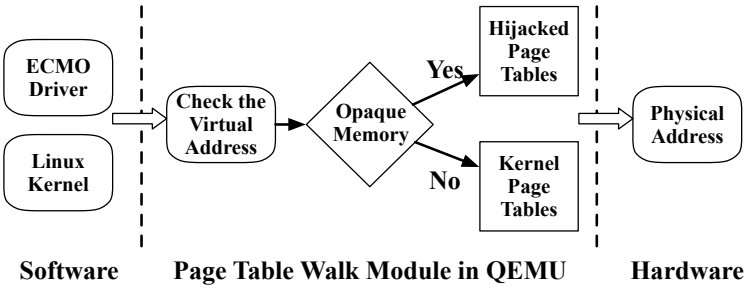


Figure 12: The overall design of opaque memory.

4.4 Implementation Details

We implement ECMO based on LuaQEMU [11]. LuaQEMU is a dynamic analysis framework based on QEMU that exposes several QEMU-internal APIs to LuaJIT [10] core, which is injected into QEMU. We port LuaQEMU based on old QEMU (version 2.9.50) to support the QEMU in new version (4.0.0) and exposes more designated APIs for initializing the peripheral models. With LuaQEMU, we are able to hijack the execution process of rehosted Linux kernel at runtime and manipulate the machine states, e.g., accessing registers and memory regions, through Lua scripts, at specified breakpoints. For example, we can specify a breakpoint at any particular addresses. Inside the breakpoint, we can execute our own Lua script for different purpose. This eases the implementation of the opaque memory, dump the decompressed Linux kernel, and install the ECMO Pointers.

The module to identify ECMO Pointers (Section 4.2) is implemented in Python. We utilize Capstone [3] to disassemble the decompressed Linux kernel. For the function identification, we re-implement the algorithm described in Nucleus [23] and angr [1] with Python. We further extract the required function information, which is the function signature based on the generated functions and their control flow graphs. Finally, we integrate all these code with our strategies for identifying ECMO Pointers, which takes 2290 lines of Python code. All the above mentioned procedures can be done automatically except that the *ECMO Driver* is developed using the C language manually, which takes 600 lines of code, and cross-compiled by GCC. Note that it is one-time effort to develop the *ECMO Driver*. One *ECMO Driver* can be used by different Linux kernel versions if the related functions and structures are not changed.

5 Evaluation

In this section, we present the evaluation result of our system. Note that, the main purpose of our work is to rehost Linux kernels in QEMU so that we can build different dynamic analysis applications and install drivers for more peripherals. In the following, we first introduce the dataset of firmware images used in the evaluation and then answer the following research questions.

- **RQ1:** Is ECMO able to identify ECMO Pointers?
- **RQ2:** Is ECMO able to rehost the Linux kernels of embedded devices with different kernel versions and device models?
- **RQ3:** Are the rehosted Linux kernels stable and reliable?
- **RQ4:** Can ECMO support more peripherals and be used to develop dynamic analysis applications?

5.1 Dataset

As our system targets embedded Linux kernels, we have collected the firmware images from both third-party projects (i.e., OpenWRT [13]) and device vendors (i.e., Netgear [12]). Our evaluation targets Linux kernels in ARM devices, since they are the popular CPU architectures in embedded devices [17]. However, the overall methodology can also be applied to other architectures (e.g., MIPS).

During the experiment, we focuses on transplanting three early-boot peripherals, i.e., interrupt controller (IC), timer, and UART, which are required to boot a Linux kernel. Once the Linux kernel is rehosted, we can install different peripheral drivers to support other peripherals with kernel modules. Specifically, we use the PrimeCell Vectored Interrupt Controller (PL190) [14] and ARM Dual-Timer Module (SP804) [2]. We use the ns16550 UART device in our system. In total, we evaluate 815 (720 in OpenWRT and 95 in Netgear) firmware images that contain Linux kernels.

Table 1: The ECMO Pointers, identification strategy, and the Linux kernel versions that the ECMO pointers used by.

Forward Pointers	Strategy	Kernel Version
init_irq	I	ALL
init_time	I	ALL
Backward Pointers	**Strategy**	**Kernel Version**
irq_set_chip_and_handler_name	III	3.18.x/4.4.x/4.14.x
irq_set_chip_data	III	ALL
handle_level_irq	II	ALL
__handle_domain_irq	III	3.18.x/4.4.x/4.14.x
setup_machine_fdt	I	3.18.x/4.4.x/4.14.x
set_handle_irq	III	3.18.x/4.4.x/4.14.x
irq_domain_add_simple	III	3.18.x/4.4.x/4.14.x
irq_create_mapping	I	3.18.x/4.4.x/4.14.x
of_find_node_by_path	II	3.18.x/4.4.x/4.14.x
setup_irq	I	ALL
clockevents_config_and_register	III	3.18.x/4.4.x/4.14.x
irq_domain_xlate_onetwocell	I	3.18.x/4.4.x/4.14.x
clockevent_delta2ns	I	2.6.x
clockevents_register_device	II	2.6.x
set_irq_flags	I	2.6.x/3.18.x
set_irq_chip	I	2.6.x
irq_to_desc	II	2.6.x
__do_div64	II	2.6.x
platform_device_register	I	ALL
lookup_machine_type	I	2.6.x
_set_irq_handler	I	2.6.x
irq_modify_status	III	4.4.x/4.14.x

Table 2: The decompressed Linux kernel size and the disassembled function numbers for our dataset.

	Maximum	Minimum	Mean	Median
Size (Bytes)	8,526,240	4,134,392	7,297,977	8,478,848
Functions (#)	48,412	18,455	29,910	23,872

5.2 Identify ECMO Pointers (RQ1)

ECMO Pointers are important to peripheral transplantation. In this section, we evaluate the success rate of identifying ECMO Pointers. Among all the 815 Linux kernels, there are 20 different kernel versions.

Table 1 lists the required ECMO Pointers, the strategies we used, and the Linux kernel versions that these ECMO Pointers are used. In total, we need to identify 24 different ECMO Pointers for all the 20 Linux kernel versions. Among them, two (i.e., `mach_desc->init_time`, and `mach_desc->init_irq`) are data pointers. Identifying the data pointers is rather more difficult than the function pointers as we need to identify symbols in each function and infer the right ones. Fortunately, these two data pointers are the return values of `setup_machine_fdt` and `lookup_machine_type`, respectively. According to the ARM calling convention, the return value is saved in register `R0`. In this case, we can identify these two data pointers by identifying function pointers `setup_machine_fdt` and `lookup_machine_type`.

Identifying ECMO Pointers requires us to disassemble the decompressed Linux kernel. Table 2 lists the information of these kernels. The decompressed Linux kernel is about 730k bytes on average, with thousands of functions. Among these functions, we successfully identify the required ECMO Pointers for all Linux kernels.

Answer to RQ1: ECMO can identify all the required ECMO Pointers from thousands of functions inside decompressed Linux kernel.

5.3 Rehost Linux Kernels (RQ2)

In this section, we evaluate the capabilities of ECMO on rehosting the Linux kernels. During this process, we use our system to boot the kernel and provide a root file system (rootfs) in the format of ramfs. We use our own rootfs because we can include different benchmark applications into the rootfs to conduct security analysis. For example, we

Table 3: The overall result of ECMO on rehosting the Linux kernel of OpenWRT. "Downloaded Images" represents the number of downloaded images. "Format Supported" represents the number of images whose formats are supported by firmware extraction tool (i.e., Binwalk). "Kernel Extracted" represents the number of images extracted from the downloaded image, which are rehosted by ECMO. "Peripherals Transplanted" represents the number of the images that peripheral can be transplanted successfully (e.g., IC can handler the interrupt well). "Ramfs are not Mounted" represents the number of images that cannot mount the given ramfs. "Shell" represents the images that we can rehost and spawn a shell. Success Rate of Transplantation = (Peripherals Transplanted)/(Images); Success Rate of Rehosting = (Shell)/(Images).

Kernel Version	Downloaded Images	Format Supported	Kernel Extracted	Peripherals Transplanted	Success Rate of Transplantation	Ramfs are not Mounted	Shell	Success Rate of Rehosting
3.18.20	23	23	21	21	100%	8	13	61.9%
3.18.23	29	29	29	29	100%	8	21	72.4%
4.4.42	37	37	37	37	100%	8	29	78.4%
4.4.47	37	37	37	37	100%	8	29	78.4%
4.4.50	45	45	45	45	100%	16	29	64.4%
4.4.61	39	39	37	37	100%	8	29	78.4%
4.4.71	40	40	38	38	100%	8	30	78.9%
4.4.89	40	40	38	38	100%	8	30	78.9%
4.4.92	41	41	38	38	100%	8	30	78.9%
4.4.140	41	41	38	38	100%	8	30	78.9%
4.4.153	40	38	38	38	100%	8	30	78.9%
4.4.182	40	38	38	38	100%	8	30	78.9%
4.14.54	54	54	42	42	100%	0	42	100%
4.14.63	66	66	42	42	100%	0	42	100%
4.14.95	66	66	42	42	100%	0	42	100%
4.14.128	66	66	42	42	100%	0	42	100%
4.14.131	66	66	42	42	100%	0	42	100%
4.14.151	66	66	42	42	100%	0	42	100%
4.14.162	66	66	42	42	100%	0	42	100%
Overall	902	898	720	720	100%	96	624	86.7%

include PoCs of kernel exploits to conduct the root cause analysis (Section 5.5). Furthermore, we can include different peripheral drivers to support more peripherals. The rootfs extracted from the firmware image can also be used.

5.3.1 Firmware Images from Third Party Projects

Table 3 shows the overall result and the success rate of peripheral transplantation and kernel rehosting for OpenWRT. We define the success of peripheral transplantation as that the transplanted IC, timer and UART devices function well in the kernel. If the rehosted kernel enters into the user-space and spawns a shell, we treat it as a successful kernel rehosting. In total, we download 902 firmware images from OpenWRT. However, four images' formats are not supported by Binwalk and the Linux kernel cannot be extracted (if there is). For the left 898 firmware images, 720 of them contain Linux kernels while the left ones contain only user-level applications. The 720 ones will be evaluated by ECMO.

Linux Kernel Versions The kernels in the 720 OpenWRT firmware images consist of 19 different kernel versions. Our evaluation shows that we can transplant the peripherals for all the 720 Linux kernels. However, some Linux kernels cannot be booted. This is because they cannot recognize our pre-built root file system (in the ramfs file format) as the support of ramfs is not enabled when being built. Without the root file system, we cannot launch the shell. However, all of them enter into the function (i.e., `init_post`) to execute the `init` program. In summary, among 720 kernels, our system can rehost 624 of them, which is shown in Table 3.

Vendors and Device Models As the OpenWRT project supports devices from multiple vendors, we calculate the supported vendors and there are 24 different vendors. Figure 13 shows the result of the top five vendors, i.e., `Netgear`, `Asus`, `Pogoplug`, `Buffalo`, and `Linksys`, in the OpenWRT dataset. Among them, `Pogoplug` has a relatively low success rate of rehosting. That's because most kernels from that vendor cannot recognize our pre-built root file system. We also count the number of device models for the successfully rehosted Linux kernels. In total, 32 device models are identified.

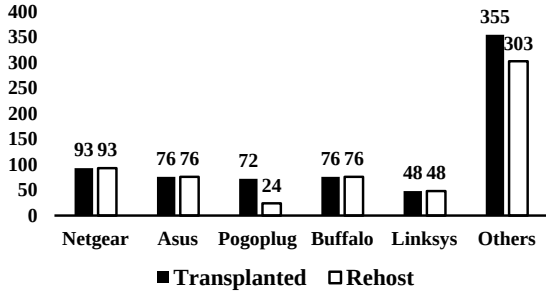


Figure 13: Supported Vendors of OpenWRT Linux Kernels.

Table 4: The overall result of ECMO on rehosting the Linux kernel of Netgear Devices.

Device Name	Kernel Version	Images	# of Peripherals Transplanted	Shell
R6250	2.6.36	21	21	15
R6300v2	2.6.36	22	22	19
R6400	2.6.36	20	20	20
R6700	2.6.36	16	16	16
R6900	2.6.36	16	16	16
Overall	-	95	95	86

5.3.2 Firmware Images from Official Vendors

Besides third-party firmware images, we also apply ECMO on the official images released by `Netgear`. We collect the firmware images for five popular devices, including R6250, R6300v2, R6400, R6700, R6900, from the vendor's website [12]. In total, we manage to collect 95 firmware images, and the latest one is released on 2020-09-30. Table 4 shows the result. We noticed that all the Linux kernels of these devices are in the version 2.6.36. We can successfully transplant the peripherals to all the 95 different firmware images. Among them, we can launch the shell for 86 images while the left 9 cannot be rehosted due to the same root file system problem.

Answer to RQ2: ECMO can rehost the Linux kernel of embedded devices from 20 kernel versions and 37 (32 in OpenWRT and 5 in Netgear) device models. Peripherals can be transplanted to all the Linux kernels while 87.1% (710/815) Linux kernels can be successfully rehosted (i.e., launch the shell).

5.4 Reliability and Stability (RQ3)

We use the LTP (Linux Test Project [8]) testsuite to evaluate the reliability and stability of the rehosted kernel. In total, there are $1,257$ test cases for system calls. Among them, 148 are skipped as the testing environment (e.g., the CPU architecture and the build configuration) does not meet the requirement. For the left $1,109$ test cases, $1,043$ passed while the left 66 ones failed.

We further analyze the reason for the failed test cases. Table 5 lists the category of the reason. Among them, 15 cases are due to the lack of network devices. This is expected since our system does not add the support of network device initially. However, all the 15 test cases are passed after installing the Ethernet device driver with kernel modules on the rehosted Linux kernel (Section 5.5.1). Also, 16 cases aim to test whether the Linux kernel fixes a bug or vulnerability. For instance, the test case (timer_create03 [9]) is to check whether CVE-2017-18344 [5] is fixed. If the vulnerability is not fixed, the test case will fail. They are also expected since the testing kernel does not fix these vulnerabilities. The other 25 cases return back the `ENOSYS` error number, which means the functionalities are not implemented. For the remaining 10 cases, the reason is adhoc, such as the kernel version is old and timeout.

In summary, 94% of the system call test cases passed. This evaluation shows the rehosted kernel is reliable and stable. We further demonstrate the usage scenarios of the rehosted Linux kernel in Section 5.5.

Answer to RQ3: The rehosted Linux kernel can pass 94% system call test cases in LTP, which demonstrates its reliability and stability.

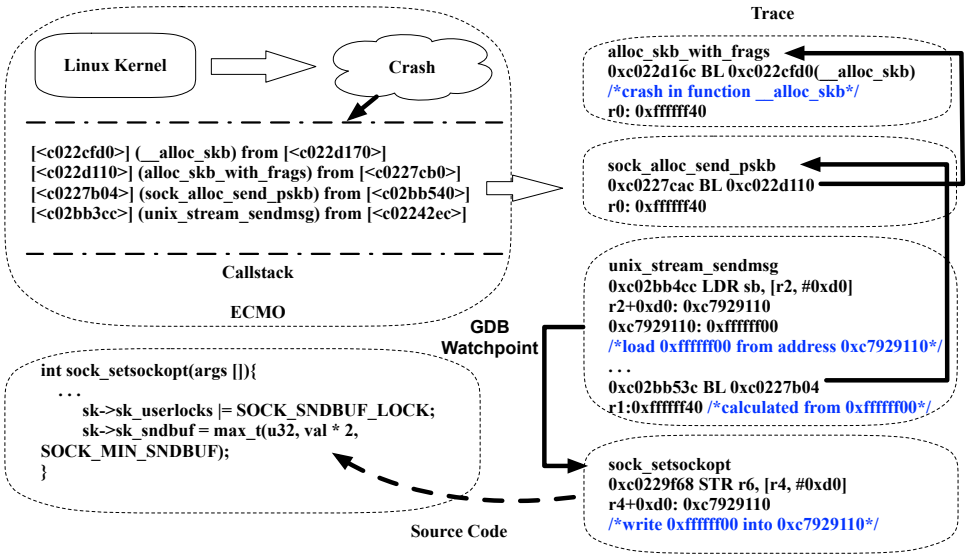


Figure 14: Root cause analysis of CVE-2016-9793.

Table 5: The category of the failed syscall test cases.

Category of Failed cases	Number
Testing the bug or vulnerability of Linux kernel	16
Network is not enabled	15
The function is not implemented	25
Others	10
Total	66

5.5 Applications and Other Peripherals (RQ4)

Our system can rehost Linux kernels, which provides the capability to install different peripheral drivers with kernel modules to support more peripherals. Furthermore, the rehosted Linux kernel lays the foundation of applications relying on the capability to introspect the runtime states of the target system. In this section, we successfully install the Ethernet device driver (i.e., smc91x) for all the rehosted Linux kernels. We also leverage our system to build three applications, including kernel crash analysis, rootkit forensic analysis, and kernel fuzzing, to demonstrate the usage scenarios of ECMO. Other applications that rely on QEMU can be ported. Note that, we only use these applications to demonstrate the usage of our system. The applications are not the main contribution of this work.

5.5.1 Other Peripherals

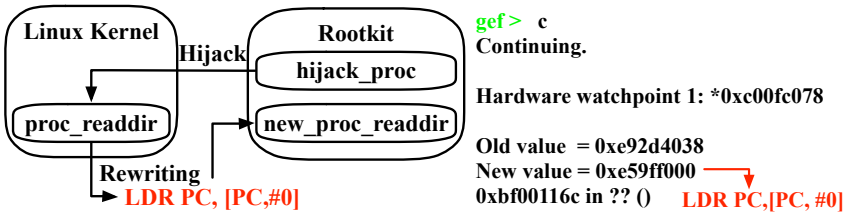
Linux kernel module is an object file that can be loaded during the runtime to extend the functionality of the Linux kernel. In this case, peripheral drivers can be built as kernel modules and loaded into the kernel dynamically. To demonstrate that our rehosted Linux kernel is able to support more peripherals. we select one rather complex peripheral (i.e., smc91x [18]) and build the driver code into kernel module (i.e., smc91x.ko). We then inject this kernel module into the ramfs that is fed to rehosted Linux kernel. After the embedded Linux kernel is rehosted by ECMO, we use the command *insmod smc91x.ko* to install the peripheral driver for smc91x. Meanwhile, QEMU has already provided the peripheral model for smc91x and we can integrate this model into the machine model directly. Finally, we successfully install the peripheral driver of smc91x for all the 710 rehosted Linux kernels, which demonstrate the capability of ECMO to support the other peripherals.

5.5.2 Crash Analysis

In the following, we show the process to utilize ECMO to understand the root cause of the crash on rehosted kernels. To this end, we collect the PoCs that can trigger the crash for six reported bugs and vulnerabilities (as shown in Table 6). We then boot the Linux kernel and run the PoCs to crash the kernel. During this process, we use the QEMU to collect the runtime trace. We also leverage the remote GDB in QEMU to debug the rehosted kernel. We detail the

Table 6: CVEs that can be triggered on the rehosted Linux kernel by ECMO.

CVE ID	CVE Score	CVE Type	Fix Version
CVE-2018-5333	5.5	Null Pointer Dereference	4.14.13
CVE-2016-4557	7.8	Double Free	4.5.5
CVE-2017-10661	7.0	Race Condition	4.10.15
CVE-2016-0728	7.8	Interger Overflow	4.4.1
CVE-2016-9793	7.8	Type Confusion	4.8.14
CVE-2017-12193	5.5	Null Pointer Dereference	4.13.11



(a) Workflow of rootkit *Suterusu*

(b) ECMO observes how the rootkit *Suterusu* works.

Figure 15: The workflow of rootkit *Suterusu* and how ECMO analyzes the behavior

procedures on how to conduct the crash analysis for one case (CVE-2016-9793 [4]) with the collected runtime trace. Figure 14 shows the whole procedure.

Specifically, when the rehosted Linux kernel crashes, the detailed call stack will be printed out. The call stack includes the function name and the addresses of these functions. With the runtime trace provided by QEMU, we can get the information including the register values and the execution path. By analyzing the trace, we noticed that a negative value (i.e., `0xffffff40`) is the first parameter of the function `__alloc_skb`. This negative value results in the crash.

We then analyze the propagation of this negative value within the trace. This value is propagated by the first parameter of the function `sock_alloc_send_pskb`. Finally, we notice that the negative value `0xffffff40` is calculated from `0xffffff00`, which is loaded by the function `unix_stream_sendmsg` from the address `0xc7929110`. We then use the `GDB` to set a watchpoint at this memory address and capture that the instruction at the address `0xc0229f68` was writing the negative value (i.e., `0xffffff00`) into this memory location.

We further analyze the function that contains the instruction at the address `0xc0229f68`. It turns out that the root cause of the crash is because of the type confusion. In the function `sock_setsockopt`, the variable `sk→sk_sndbuf` will be set by the return value of `max_t` (maximum value between two values in the same type). However, due to the wrong type `u32`, the return value can be a negative value, which triggers the crash.

This analysis shows the usage of ECMO by providing the capability introspect the runtime states of the rehosted kernel.

5.5.3 Rootkit Forensic Analysis

Rootkit forensic analysis requires the ability to monitor the runtime states of the kernel [40, 48]. We demonstrate this ability by conducting the rootkit forensic analysis with one (i.e., *Suterusu* [20]) popular rootkit in the wild.

Specifically, `Suterusu` is able to hide specific processes by hijacking the kernel function `proc_readdir`, which is used to get the process information. As shown in Figure 15(a), it hijacks the function `proc_readdir` by rewriting the function's first instruction to `LDR PC,[PC,#0]`. As a result, it redirects the execution to the function *new_proc_readdir* inside the rootkit. With ECMO, we can monitor the changes to the kernel code sections (a suspicious behavior) by setting up memory watchpoints to the Linux code section (Figure 15(b)).

5.5.4 Fuzzing

Fuzzing has been widely used to detect software vulnerabilities. We ported one of the most popular kernel fuzzers (i.e.,`UnicornFuzz` [51]) into ECMO and fuzzed the example kernel modules provided by `UnicornFuzz`. `UnicornFuzz` can work under ECMO and the fuzzing speed can reach to 396 instances per second. This demonstrates the usage of ECMO for kernel fuzzing.

Answer to RQ4: Applications, e.g., crash analysis, forensic analysis, kernel fuzzing, can be built upon the rehosted Linux kernel by our system. Furthermore, rehosted Linux kernel can install peripheral drivers with kernel modules to support more peripherals.

6 Discussion

Manual efforts ECMO provides mostly automated approach and only developing the *ECMO Driver* requires manual efforts. However, this is a one-time effort. Furthermore, one *ECMO Driver* can be transplanted to different kernel versions if the related functions and structures are not changed. Even if the functions are changed, we just need to change a few APIs and compile it again to create a new *ECMO Driver*. For example, the 815 Linux kernels consist of 20 different kernel versions. For the kernel in version 2.6.36, it takes 385 lines of C code. This driver can be used for all the kernel images of Netgear (Table 4). For the kernel in version 3.18.20 and 3.18.23, it takes 534 lines of C code while 180 lines of new code are added. For kernels in all the left 17 versions, they share the same driver code. 60 lines of new code are added compared with the one used in 3.18.20. Note that the driver code for the transplanted peripherals does not need to be developed. Instead, we reuse the existing code. For example, the driver code for VIC (PL190) is open source [15]. Thus, we just reuse the existing driver code, merge the driver code into one file, and finally compile it to generate the ECMO driver. In total, it takes less than one person-hour to build a new customized driver.

Functionalities of peripherals We successfully boot the Linux kernel by transplanting designated peripherals (e.g., IC, Timer, and UART). We admit that the original peripherals may not work property as they are not emulated (or transplanted) in QEMU. However, the functionalities of the transplanted peripherals are guaranteed. With the transplanted peripherals, ECMO can provide the capability to introspect the runtime states of the Linux kernel that dynamic analysis applications can be built upon. Without our system, it's impossible to build such applications since the target Linux kernel cannot be booted in QEMU. The three applications used in the evaluation have demonstrated the usage scenarios of our system. We may build or port more complicated applications, e.g., dynamic taint analysis [60], to further evaluate our system.

Other peripherals Currently, ECMO is evaluated based on transplanting three early-boot peripherals (i.e., IC, timer, and UART) as they are required to boot a Linux kernel. In general, peripheral transplantation works on all kinds of peripherals. The transplanting process depends on the identification of ECMO pointers. Fortunately, to support the other peripherals, users can install the kernel modules directly on the rehosted Linux kernel, which does not need to identify pointers. In this case, all kinds of peripherals can be supported. Our experiments show that the driver of Ethernet device, which is rather complex, can be successfully installed and the network functionality can be guaranteed.

Other architectures Currently, ECMO only supports ARM architecture, which is the most popular one in embedded systems [17]. However, the technique peripheral transplantation can be easily extended to the other architecture as it does not rely on any particular architecture feature. Specifically, developers need to implement the module for identifying ECMO Pointers for the new architecture. This requires additional engineering efforts and algorithm 1 is provided.

7 Related Work

Static Firmware Analysis Researchers apply the static analysis technique to analyze the embedded firmware. For instance, Costin et al. [30] conduct a large-scale analysis towards the embedded firmware. By analyzing 32 thousand firmware images, many new vulnerabilities are discovered, influencing 123 products.

Code similarity is widely used to study the security issue of embedded devices. Feng et al. propose Genius [38], a new bug search system to address the scalability issues by translating binary control flow graph to high-level numeric feature vectors. The experiments show that Genius can identify many vulnerabilities in a short time. Considering the inaccuracy of approximate graph-matching algorithm, Xu et al. utilize neural network-based approach to abstract the control flow graph of binary function and build a prototype named Gemini [65]. The result shows Gemini can identify more vulnerable firmware images compared with Genius. Yaniv et al. introduce a precise and scalable tool named Firmup [32] by considering the relationship between procedures. The result show Firmup has a relatively low false positive and effective on discovering vulnerabilities. In the case that firmware images are not available, Xueqiang et al. [63] applies cross analysis of mobile apps to detect the vulnerable devices. Finally, 324 devices from 73 different

vendors are discovered. Our system is used to analyze the firmware images of embedded systems with dynamic analysis. Application building upon ECMO can complement the static analysis ones.

Dynamic Firmware Analysis Besides static analysis, researchers propose several methods to support the dynamic firmware analysis. Avatar [69] is proposed to support complex dynamic analysis of embedded devices by orchestrating the execution of an emulator and real hardware. Charm [62] applies a similar strategy. It introduces the technique named remote device driver execution by forwarding the MMIO operation to a real mobile. Avatar2 [54] extends Avatar to support replay without real devices. However, they both suffer from the problem of scalability. Inception [29] applies symbolic execution based on KLEE [25] and a custom JTAG to improve testing embedded software. However, it assumes that the source code is available. IoTFuzzer [27] aims to fuzz the firmware from the mobile side. However, the code coverage of firmware and the coverage of attack surface are limited. Pretender [43] is able to conduct automatically rehosting tasks. However, it replies on the debug interface of specific devices. Jetset [49] utilizes the symbolic execution to infer the return values of device registers. However, the functionality of the peripherals cannot be guaranteed. Furthermore, the shell may not be obtained for further development of different applications.

Besides, many researchers utilize the fuzzing technique to detect the security issues of embedded firmware. P2IM [37] is proposed to learn the model of peripherals automatically. DICE [52] focused on the DMA controller and can extend the P2IM's analysis coverage. Halucinator [28] proposed a new methodology to rehost the firmware by abstracting the HAL functions. ECMO are different from them in the aspects to transplant peripherals into the target kernel, instead of inferring the peripherals models. Besides, all these systems focus on bare-metal system, which is less complicated than the Linux kernel. Firmadyne [26] and FirmAE [50] target on Linux-based firmware. However, they focus on the user-space program, instead of the Linux kernel.

Applications based on QEMU There are many applications based on QEMU. For example, researchers have developed new fuzzing systems [51, 70, 21] based on QEMU. KVM leverages the device emulation provided by QEMU or the virtio [58] framework for device virtualization. The idea of virtio is similar to ECMO. However, virtio requires to change the source code of guest while ECMO works towards the Linux kernel binary. Virtual machine introspection tools [40, 64, 33, 34, 39, 24], which are helpful for debugging or forensic analysis, utilize QEMU to introspect the system states. Furthermore, dynamic analysis frameworks use QEMU to analyze malware behavior [36, 53, 66, 57, 68, 67]. ECMO provides the capability to rehost Linux kernels, which lays the foundation for apply these applications on embedded Linux kernels.

8 Conclusion

In this work, we propose a novel technique named peripheral transplantation to rehost the Linux kernel of embedded devices in QEMU. This lays the foundation for applications that rely on the capability of runtime state introspection. We have implemented this technique inside a prototype system called ECMO and applied it to 815 firmware images, which consist of 20 kernel versions and 37 device models. ECMO can successfully transplant peripherals for Linux kernels in all images. Among them, 710 kernels can be successfully rehosted, i.e., launching the user-space shell (87.1% success rate). Furthermore, we successfully install one Ethernet device driver (i.e., smc91x) on all the rehosted Linux kernels to demonstrate the capability of ECMO to support more peripherals. We further build three applications to show the usage scenarios of ECMO.

References

[1] angr. `https://angr.io/`.

[2] Arm dual-timer module (sp804). `https://developer.arm.com/documentation/ddi0271/d/`.

[3] Capstone. `https://www.capstone-engine.org/`.

[4] CVE-2016-9793. `https://nvd.nist.gov/vuln/detail/CVE-2016-9793`.

[5] CVE-2017-18344. `https://nvd.nist.gov/vuln/detail/CVE-2017-18344`.

[6] Docker image that contains the ecmo system. `https://github.com/anonymous-researcher22/ecmo_docker`.

[7] Iot devices market. `https://www.zionmarketresearch.com/requestbrochure/iot-devices-market`.

[8] Linux Test Project. `http://linux-test-project.github.io/`.

[9] Linux test project test case timer_create03. `https://github.com/linux-test-project/ltp/blob/master/testcases/kernel/syscalls/timer_create/timer_create03.c`.

[10] Luajit. http://luajit.org/luajit.html.

[11] Luaqemu. https://github.com/Comsecuris/luaqemu.

[12] Netgear. https://www.netgear.com/.

[13] Openwrt. https://openwrt.org/.

[14] Primecell vectored interrupt controller (pl190). https://developer.arm.com/documentation/ddi0181/e/introduction/about-the-vic.

[15] Primecell vectored interrupt controller (pl190) source code. https://elixir.bootlin.com/linux/v3.18.20/source/drivers/irqchip/irq-vic.c#L445.

[16] Qemu. https://www.qemu.org/.

[17] The roadshow of arm. https://group.softbank/system/files/pdf/ir/presentations/2019/arm-roadshow-slides_q4fy2019_01_en.pdf.

[18] Smc91x source code. https://elixir.bootlin.com/linux/v3.18.20/source/drivers/irqchip/irq-vic.c#L445.

[19] Soc (system on a chip). https://openwrt.org/docs/techref/hardware/soc.

[20] suterusu. https://github.com/mncoppola/suterusu.

[21] Triforceafl. https://github.com/nccgroup/TriforceAFL.

[22] Vulnerability statistics of linux kernel. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html.

[23] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *Proceedings of the 2nd IEEE European Symposium on Security and Privacy*, 2017.

[24] Sina Bahram, Xuxian Jiang, Zhi Wang, Mike Grace, Jinku Li, Deepa Srinivasan, Junghwan Rhee, and Dongyan Xu. Dksm: Subverting virtual machine introspection for fun and profit. In *Proceedings of the 29th IEEE symposium on reliable distributed systems*, 2010.

[25] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008.

[26] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium*, 2016.

[27] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. IoTFuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, 2018.

[28] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. Halucinator: Firmware re-hosting through abstraction layer emulation. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[29] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. Inception: System-wide security testing of real-world embedded systems software. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[30] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A large scale analysis of the security of embedded firmwares. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[31] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. {REPT}: Reverse debugging of failures in deployed software. In *Proceedings of the 13th {USENIX} Symposium on Operating Systems Design and Implementation*, 2018.

[32] Yaniv David, Nimrod Partush, and Eran Yahav. FirmUp: Precise static detection of common vulnerabilities in firmware. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[33] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 32nd IEEE symposium on security and privacy*, 2011.

[34] Pavel Dovgalyuk, Natalia Fursova, Ivan Vasiliev, and Vladimir Makarov. Qemu-based framework for non-intrusive virtual machine instrumentation and introspection. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[35] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*, 2017.

[36] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *Proceedings of the 2007 USENIX Annual Technical Conference*, 2007.

[37] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *Proceedings of the 29th USENIX Security Symposium*, 2019.

[38] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[39] Yangchun Fu and Zhiqiang Lin. Bridging the semantic gap in virtual machine introspection via online kernel data redirection. *ACM Transactions on Information and System Security*, 2013.

[40] Tal Garfinkel, Mendel Rosenblum, et al. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the 2003 Annual Network and Distributed System Security Symposium*, 2003.

[41] Xinyang Ge, Ben Niu, and Weidong Cui. Reverse debugging of kernel failures in deployed systems. In *Proceedings of the 2020 USENIX Annual Technical Conference*, 2020.

[42] Daniel M German and Jesús M González-Barahona. An empirical study of the reuse of software licensed under the gnu general public license. In *IFIP International Conference on Open Source Systems*. Springer, 2009.

[43] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurélien Francillon, Yung Ryn Choe, Christophe Kruegel, and Giovanni Vigna. Toward the analysis of embedded firmware through automated re-hosting. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses*, 2019.

[44] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. PARTEMU: Enabling dynamic analysis of real-world trustzone software using emulation. In *Proceedings of the 29th USENIX Security Symposium*, 2020.

[45] Grant Hernandez, Farhaan Fowze, Dave Tian, Tuba Yavuz, and Kevin RB Butler. FirmUSB: Vetting usb device firmware using domain informed symbolic execution. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[46] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. An empirical study on arm disassembly tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[47] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based" out-of-the-box" semantic view reconstruction. In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[48] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection and monitoring through vmm-based "out-of-the-box" semantic view reconstruction. *ACM Transactions on Information and System Security*, 2010.

[49] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted firmware rehosting for embedded systems. In *Proceedings of the 30th {USENIX} Security Symposium*, 2021.

[50] Mingeun Kim, Dongkwan Kim, Eunsoo Kim, Suryeon Kim, Yeongjin Jang, and Yongdae Kim. Firmae: Towards large-scale emulation of iot firmware for dynamic analysis. In *Proceedings of the 2020 Annual Computer Security Applications Conference*, 2020.

[51] Dominik Maier, Benedikt Radtke, and Bastian Harren. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *Proceedings of the 13rd {USENIX} Workshop on Offensive Technologies ({WOOT} 19)*, 2019.

[52] Alejandro Mera, Bo Feng, Long Lu, Engin Kirda, and William Robertson. DICE: Automatic emulation of dma input channels for dynamic firmware analysis. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, 2021.

[53] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 28th IEEE Symposium on Security and Privacy*, 2007.

[54] Marius Muench, Dario Nisi, Aurelien Francillon, and Davide Balzarotti. Avatar2: A Multi-target Orchestration Platform. In *Workshop on Binary Analysis Research*, 2018.

[55] Nilo Redini, Aravind Machiry, Ruoyu Wang, Chad Spensky, Andrea Continella, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Karonte: Detecting insecure multi-binary interactions in embedded firmware. In *Proceedings of the 41st IEEE Symposium on Security & Privacy*, 2020.

[56] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the 11st International Workshop on Recent Advances in Intrusion Detection*, 2008.

[57] Ryan Riley, Xuxian Jiang, and Dongyan Xu. Multi-aspect profiling of kernel rootkit behavior. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 47–60, 2009.

[58] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 2008.

[59] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for {OS} kernels. In *Proceedings of the 26th {USENIX} Security Symposium*, 2017.

[60] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 31st IEEE symposium on Security and privacy*, 2010.

[61] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, 2015.

[62] Seyed Mohammadjavad Seyed Talebi, Hamid Tavakoli, Hang Zhang, Zheng Zhang, Ardalan Amiri Sani, and Zhiyun Qian. Charm: Facilitating dynamic analysis of device drivers of mobile systems. In *Proceedings of the 27th USENIX Security Symposium*, 2018.

[63] Xueqiang Wang, Yuqiong Sun, Susanta Nanda, and XiaoFeng Wang. Looking from the mirror: evaluating iot device security through mobile companion apps. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[64] Zhi Wang, Xuxian Jiang, Weidong Cui, and Xinyuan Wang. Countering persistent kernel rootkits through systematic hook discovery. In *Proceedings of the 11st International Workshop on Recent Advances in Intrusion Detection*, 2008.

[65] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[66] Lok-Kwong Yan, Manjukumar Jayachandra, Mu Zhang, and Heng Yin. V2e: combining hardware virtualization and softwareemulation for transparent and extensible malware analysis. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012.

[67] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st {USENIX} Security Symposium*, 2012.

[68] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[69] Jonas Zaddach, Luca Bruno, Aurélien Francillon, and Davide Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, 2014.

[70] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firmafl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *Proceedings of the 28th USENIX Security Symposium*, 2019.