FirmGuide: Boosting the Capability of Rehosting Embedded Linux Kernels through Model-Guided Kernel Execution

Qiang Liu*●, Cen Zhang†● , Lin Ma*, Muhui Jiang‡*, Yajin Zhou*, Lei Wu** ,
Wenbo Shen*, Xiapu Luo‡, Yang Liu†, Kui Ren*
* Zhejiang University
† Nanyang Technological University
‡ The Hong Kong Polytechnic University

***Abstract*—Linux kernel is widely used in embedded systems. To understand practical threats to the Linux kernel, we need to perform dynamic analysis with a full-system emulator, e.g., QEMU. However, due to hardware fragmentation, e.g., various types of peripherals, most embedded systems are not currently supported by QEMU. Though some progress has been made on rehosting firmware, it mainly focuses on user space programs or simple real-time operating systems.**

The goal of this work is to boost the capability of rehosting the embedded Linux kernels in QEMU. By doing so, dynamic analysis systems can be firstly applied on embedded Linux kernels by leveraging off-the-shelf tools upon QEMU. Accordingly, we proposed a new technique called *model-guided kernel execution*. It combines the peripheral abstractions in the Linux kernel and kernel-peripheral interactions to *semi-automatically* generate peripheral models that are then used to synthesize new QEMU virtual machines to start the dynamic analysis.

We have implemented a prototype called `FirmGuide`. It generates 9 peripheral models with full functionality and 64 with minimum functionality covering 26 SoCs. Our evaluation with $6,188$ firmware images shows that it can successfully rehost more than 95% of Linux kernels in 2 architectures and 22 versions. None of them can be rehosted in the vanilla QEMU. The result of the LTP benchmark shows the reliability and robustness of the rehosted Linux kernels. We further conduct two security applications, i.e., vulnerability analysis and fuzzing, on the rehosted Linux kernels to demonstrate the usage scenarios.

I. Introduction

With the proliferation of IoT (or embedded) devices in recent years, the firmware (the software stack) of these devices has become one of the most common attack targets [1–3]. Many of them integrate the Linux kernel [4,5] in which vulnerabilities are still discovered every year [6]. What's worse, the device vendors do not timely apply or backport security patches from the mainstream to the firmware [7–9], exposing vulnerable devices in the wild. Once being exploited, these devices can be fully controlled by attackers.

Rehosting, also known as emulation, is used to load and run the analysis target, e.g., the Linux kernel of the firmware, inside an emulator (e.g., QEMU) and provides the capability to introspect the runtime states of the target. After that, various security analysis applications can be applied, e.g., vulnerability analysis and fuzzing. Running the Linux kernel of the desktop system in QEMU is not an issue. However, embedded devices tend to use different types of system-on-chips (SoCs) that are currently unsupported by QEMU. Hence, it's still an underdeveloped research problem to dynamically run the embedded Linux kernels inside an emulated environment.

In this work, we focus on rehosting embedded Linux kernels to lay the foundation of the dynamic analysis for them. Though researchers have made progress in firmware rehosting, these tools cannot serve our purpose. First, they target user-space programs of the firmware, instead of the Linux kernels [5,10–12]. For instance, FIRMADYNE [5] rehosts user-space programs by using a customized Linux kernel that can be loaded in QEMU. Since the loaded kernel is different from the real one, it cannot be used to analyze the original Linux kernel. Second, they pay attention to non-Linux kernels of bare-metal systems [13–15], or leverage real hardware [16] for analysis, which is not scalable in the context of the Linux kernel. To the best of our knowledge, no system can rehost the embedded Linux kernel on a large scale yet.

Rehosting the embedded Linux kernel is challenging. First, the booting process depends on multiple peripherals. For instance, after analyzing $1,639$ device tree blobs in the Linux kernel, we found that each device has 32 peripherals on average. Second, the same type of peripherals usually has different hardware designs. There are thousands of peripherals supported by the Linux kernel and it is impractical to implement them one by one. Third, peripheral interfaces have semantic diversity. A peripheral usually exposes several interfaces (hardware registers) to the Linux kernel to control its inner states and each register has its specific semantics. A successful rehosting requires an understanding of these interfaces' semantics. These challenges make the manual rehosting of these peripherals a tedious and error-prone task.

Our approach *In this paper, we aim to boost the capability of rehosting embedded Linux kernels in QEMU. By doing so, existing dynamic analysis tools built upon QEMU can be easily applied to analyze the embedded Linux kernels.* We have three observations to semi-automatically generate peripheral models.

- ❶ Rehosting the Linux kernel only requires the full emulation

● The first two authors contributed equally to this work.
* Corresponding author, lei_wu@zju.edu.cn.

of a few peripherals (called Type-I peripherals in this paper), e.g., *Interrupt Controller* and *Timer* are two Type-I peripherals that require full functionality emulation. For other peripherals (called Type-II peripherals in this paper), we only need to emulate their minimum functionalities with properly initialized values, such as Network Card and Flash.

- ❷ The Linux kernel has well-defined abstractions for different types of peripherals. For instance, the Linux kernel interrupt subsystem abstracts common actions of the *Interrupt Controller* and defines these actions as callback functions for low-level device drivers to register. There is a similar design for the Linux kernel time subsystem and the Timer peripherals.
- ❸ The well-defined abstractions and kernel-peripheral interactions together can describe the peripheral interfaces. For instance, the interrupt subsystem draws the inner states of the *Interrupt Controller*. If we can use kernel-peripheral interactions to identify which callback function is executed, we can then transit the inner states properly.

Subsequently, we propose a new technique called *model-guided kernel execution*. It *semi-automatically* builds the peripheral models that QEMU does not support yet by analyzing the Linux kernel source code. The peripheral model consists of two parts, the *general model template* and the *specific model parameters*. The model template is manually built based on the Linux kernel's abstraction layers for that type of peripherals, e.g., the interrupt subsystem, which is a one-time effort. These abstractions help us construct state machines that can cooperate with the Linux kernel. The state machine defines all the states and the state transition table but leaves the transition conditions (events) as blanks. We then extract model parameters from the peripheral's driver code to fill these blanks to generate transition conditions. The parameters are automatically generated by symbolic execution. The peripheral models then can guide the kernel's execution in a synthesized QEMU virtual machine.

We have developed a prototype called `FirmGuide` that has two components. One is *offline model generation* analyzing the Linux kernel source code to *semi-automatically* generate peripheral models. The other is *online kernel booting* checking the hardware dependency of the Linux kernel with its device tree and rehosting the Linux kernel with a synthesized QEMU virtual machine. Note that `FirmGuide` requires the Linux kernel source code during the first component to generate peripheral models, but does not need the source code in the second component to rehost the Linux kernel inside the firmware.

Evaluation To evaluate the effectiveness of our method, we manage to generate 9 Type-I peripheral models and 64 Type-II peripheral models (Section VI-B) and synthesize corresponding QEMU virtual machines for each embedded Linux kernel. With the virtual machines, we've rehosted the Linux kernels in $6,188$ firmware images downloaded from the Internet [17,18]. On one hand, $5,947$ (96.11%) Linux kernels are successfully rehosted (entering into the user space), and none of them can be rehosted in the vanilla QEMU (Section VI-C). On the other hand, the rehosted Linux kernels cover 26 SoCs, 2 architectures, and 22 kernel versions (Section VI-D). Note that the rehosted Linux kernel version is not necessarily as same as the version to generate peripheral models. This result, together with the number of rehosted Linux kernels, demonstrated the scalability of our system. Furthermore, we used the Linux Testing Projects (LTP) [19] to test the functionality of the rehosted Linux kernels, showing the feasibility to build security applications (Section VI-E).

We further present two applications to show the usage scenarios with the support of the synthesized QEMU virtual machines. First, we analyzed 6 Linux kernel CVEs. With the help of the debugging capability of QEMU, we have successfully triggered 5 and exploited 4 of them, respectively. This shows the usage of the synthesized QEMU virtual machines to trigger, understand, and exploit vulnerabilities of the rehosted Linux kernels. Second, we have ported two fuzzing tools UnicoreFuzz [20] and TriforceAFL [21] to demonstrate the capability of supporting other dynamic analysis tools to the rehosted Linux kernels. The applications themselves are not our contribution but can demonstrate the usage scenarios of `FirmGuide`. Nevertheless, without the capability of `FirmGuide` to rehost the embedded Linux kernels, it's hard, if not impossible, to apply these security applications to them.

Contributions Our main contributions are as follows.

- We summarized the problem of rehosting embedded Linux kernels and proposed a new technique called *model-guided kernel execution* to *semi-automatically* generate stateful peripheral models that are not supported in QEMU.
- We implemented a prototype called `FirmGuide` to boost the capability of rehosting embedded Linux kernels. Evaluated with $6,188$ firmware, the result shows that we can successfully rehost more than 95% Linux kernels covering 26 SoCs, 2 architectures, and 22 versions.
- `FirmGuide` lays a foundation to perform the dynamic analysis of embedded Linux kernels. We applied `FirmGuide` to vulnerability analysis and fuzzing to show the usage scenarios.

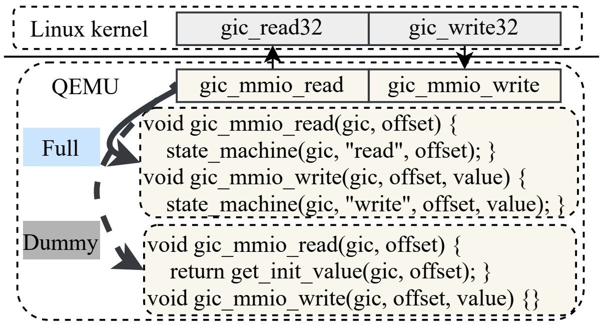
We have released the Docker image to dry run our system [22]. To engage the community, we will release the *source code* of `FirmGuide` [23].

II. Background

Embedded Systems Advanced embedded systems, e.g., routers, are often based on the SoCs that have integrated CPU, memory, and basic peripherals. Among those peripherals, Interrupt Controller and Timer are two of the most important peripherals. Peripherals use interrupts to inform the processor of what is happening and an Interrupt Controller attached to a processor is responsible for delivering the interrupts. After the Interrupt Controller notifies the processor that an interrupt is fired, the processor then retrieves the interrupt request number (IRQn) from it and jumps to the corresponding interrupt service routine (ISR). An embedded system usually requires two kinds of Timers. One connected to the Interrupt Controller is used for periodically generating interrupts for task scheduling. The other one which never raises an interrupt is used as the source

```
compatible= "plxtech,nas7820";
cpu@0; // processor
memory; // memory
ic@47001000 { // peripheral 1
  compatible="arm,arm11mp-gic";
  // MMIO memory space <start, size>
  reg = <0x47001000 0x1000>;
};
ethernet@41000000; // peripheral 2
```

Fig. 1: An example of the device tree of *OX820 NAS7820* SoC.



```
void gic_mmio_read(gic, offset) {
  state_machine(gic, "read", offset); }
void gic_mmio_write(gic, offset, value) {
  state_machine(gic, "write", offset, value); }
```

```
void gic_mmio_read(gic, offset) {
  return get_init_value(gic, offset); }
void gic_mmio_write(gic, offset, value) {}
```

Fig. 2: An example of QEMU peripheral emulation. `FirmGuide` supports both fully functional and dummy peripheral models.

of time counting. The Linux kernel maintains the timeline by periodically reading this Timer's count register.

The device tree describes the hardware information of an SoC. It is passed to the Linux kernel at the beginning of the booting process. The Linux kernel then uses it to correctly initialize peripherals. It has a property named *compatible* which is the model number. For example, as shown in Figure 1, the top-level *compatible* is the model number of the SoC and the second-level is the model number of the specific peripheral (i.e, Interrupt Controller). Besides, the device tree has a rich description of a peripheral, including its MMIO ranges, IRQn, input clock frequency, etc. We retrieve the description from the device tree for further analysis.

Memory Mapped I/O Memory Mapped I/O (MMIO) is one kind of communication method between software and hardware. With the help of MMIO, the Linux kernel can map physical registers of a peripheral to the physical address space. Therefore, from the Linux kernel's perspective, it can interact with the peripheral (reading or writing registers) as it interacts with the memory (reading or writing memory). The Port I/O (PIO) is not considered in this paper due to the following consideration: ❶ our key method also works for PIO cases; ❷ PIO is rarely used in our target firmware.

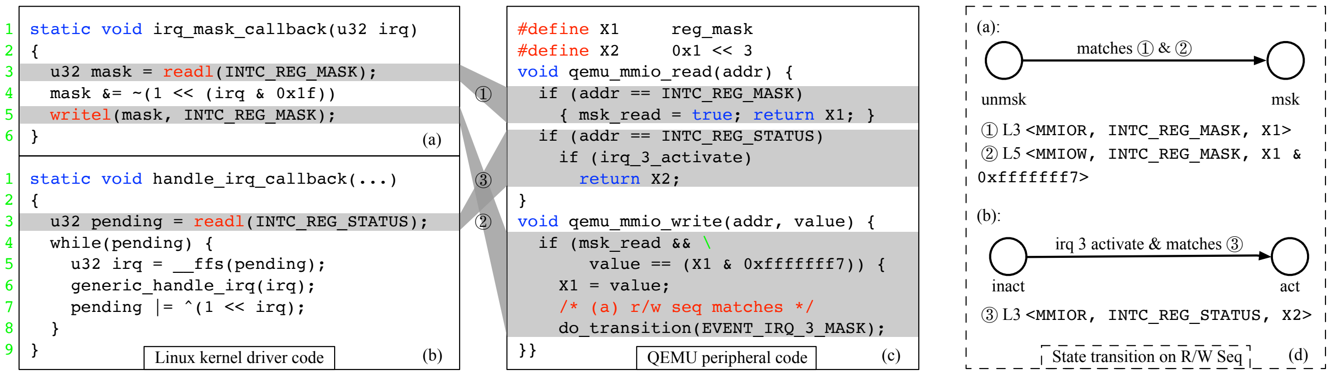
QEMU QEMU [24] is one of the most popular full system emulators that consists of peripheral models. In Figure 2, the peripheral model implements read and write callback functions for its registers. When the embedded Linux kernel reads from an MMIO address, the read callback function will be invoked. The write callback function works similarly. A fully functional peripheral model has a state machine that emulates the peripheral's functionality. If there is no such state machine, we call this a dummy peripheral model. Leaving the real functionality unimplemented, we usually need to initialize its MMIO address space with proper initialized values.

III. Problem Statement and Solution

Our goal is to boost the capability of rehosting embedded Linux kernels in QEMU. To rehost these embedded Linux kernels, we first discussed with a well-experienced embedded system engineer what he normally did to support a new device. According to his experience, we were surprised to know that, given the source code of a Linux-based firmware, extending it to support a new IoT device (till kernel booted successfully, e.g., spawn a userspace shell) only requires the adaption of the driver code of certain peripherals. This fact implies that the task for emulating the Linux-based firmware may not be as complex as it looks (may only require several key peripherals). To better understand this, we systematically analyzed the booting process of the Linux kernel and the emulation mechanism of a peripheral inside QEMU. We summarized the identified key challenges and observations for the rehosting as the following.

Challenges ❶ A real physical embedded device usually has multiple peripherals, and it is tedious and error-prone to emulate them manually. The statistics of 1,639 device tree blobs of OpenWRT firmware shows that the number of peripherals per device ranges from 5 to 76, around 32 on average. ❷ The embedded Linux kernels share common types of peripherals for different SoCs, e.g., Interrupt Controller or Timer, but these peripherals follow different hardware specifications provided by different SoC vendors, which is impractical to manually implement all of them. ❸ Every peripheral's interfaces (hardware registers) have diverse semantics, thus making the construction of a peripheral model more challenging. The diversity resides in two aspects. On one hand, through the peripheral interfaces, the Linux kernel can retrieve the status and control the behavior of the peripheral. The peripheral maintains its inner states and reacts according to different interfaces. On the other hand, a peripheral interface itself has semantics as well. For example, an Interrupt Controller maintains the status (masked, unmasked, etc.) of each interrupt source inside a register. A single bit flip of this register can lead to the change of the hardware's inner states. Building a successful peripheral model usually requires the understanding of its inner states, state transitions, and hardware register semantics.

Observations ❶ **Achieving our goal requires the full functionality emulation of only a few peripherals.** We define a successful rehosting of the Linux kernel when the CPU has entered the user mode to execute the initialization process (`run_init_process`). Though a successful rehosting process involves dozens of different peripherals (32 on average), they can be divided into two types. Specifically, Type-I peripherals include the ones which have complicated interactions with the Linux kernel and need to be emulated with their full functionality. This type of peripherals includes Interrupt Controller, Timer, and UART. A popular NS16550A UART has been well supported in QEMU and can be reused directly. However, the rest of Type-I peripherals, i.e., Interrupt Controller and Timer, require fully functional emulation. We define Type-II peripherals as the ones which only require minimum emulation



```
1 static void irq_mask_callback(u32 irq)
2 {
3   u32 mask = readl(INTC_REG_MASK);
4   mask &= ~(1 << (irq & 0x1f))
5   writel(mask, INTC_REG_MASK);
6 }
```

```
1 static void handle_irq_callback(...)
2 {
3   u32 pending = readl(INTC_REG_STATUS);
4   while(pending) {
5     u32 irq = __ffs(pending);
6     generic_handle_irq(irq);
7     pending |= ^(1 << irq);
8   }
9 }
```

```
#define X1      reg_mask
#define X2      0x1 << 3
void qemu_mmio_read(addr) {
  if (addr == INTC_REG_MASK)
    { msk_read = true; return X1; }
  if (addr == INTC_REG_STATUS)
    if (irq_3_activate)
      return X2;
}
void qemu_mmio_write(addr, value) {
  if (msk_read && \
       value == (X1 & 0xfffffff7)) {
    X1 = value;
    /* (a) r/w seq matches */
    do_transition(EVENT_IRQ_3_MASK);
}}
```

① L3 <MMIOR, INTC_REG_MASK, X1>
② L5 <MMIOW, INTC_REG_MASK, X1 & 0xfffffff7>
③ L3 <MMIOR, INTC_REG_STATUS, X2>

Fig. 3: Examples for using R/W Seq to guide kernel's execution: (a), (b) show recognition and control, respectively, (c) shows the R/W Seq matching in the peripheral model, (d) details the state transition by R/W Seq. The MMIO operations in (a), (b) are marked in grey, macro `INTC_REG_XXX` represents the MMIO register addresses, and `__ffs` is short for "find first set".

(as a dummy MMIO memory region with proper initial values.) They only need to provide suitable values when the Linux kernel reads specific peripheral registers (MMIO read) during the booting process. No inner states of the peripherals need to be emulated. Such peripherals include Network Card, Flash, etc.

❷ **The Linux kernel has well-defined abstraction models for different types of peripherals.** The Linux kernel core (upper layer) abstracts a type of peripherals as one device with a set of common actions. Each device driver (lower layer) has to implement these actions. For the Type-I peripherals (Interrupt Controller and Timer), the upper abstraction layers are the interrupt and the time subsystems, respectively. The lower implementation layers are the low-level device drivers of the peripherals. For instance, in the interrupt subsystem, each Interrupt Controller is an instance of `struct irq_domain`, and the common actions are callback functions like `irq_domain->irq_mask`, `irq_domain->irq_unmask`, `irq_domain->irq_ack`, etc. A low-level device driver provides the functions' implementation. In the rest of the paper, we use *critical functions* to represent these callback functions.

❸ **Well-defined abstraction models and kernel-peripheral interactions can be combined to describe the peripheral interfaces.** The above-mentioned kernel abstractions inspire us to build a peripheral model in QEMU that combines kernel-peripheral interactions to guide the kernel's execution to a successful rehosting (hence the name *model-guided kernel execution*). Specifically, the peripheral model is a state machine summarized from the upper layer (subsystem). It defines the states and available state transitions. This state machine can be manually extracted from the Linux kernel source code, which is a one-time effort. Besides, the peripheral model needs to recognize executed *critical functions* and then react accordingly. A key intuition is that MMIO read/write sequences (R/W Seq) from the Linux kernel to the peripherals are signatures to represent the execution paths of a *critical function*. They can be used in *recognizing* the paths of *critical functions* that the Linux kernel has executed, and *controlling* the Linux kernel's following execution by returning specific values to the Linux kernel in an MMIO read request. A path's R/W Seq can be automatically inferred via symbolic execution.

A Motivating Example Figure 3 shows an example of using R/W Seq to guide kernel's execution. The MMIO operation is marked as `<MMIOR/MMIOW, addr, expr>`, where `MMIOR/MMIOW` is read/write operation, `addr` is read/write address, and `expr` is the value the Linux kernel reads from/writes to the peripheral model. For simplicity, we only consider IRQn 3.

Figure 3(a) shows a simplified `irq_mask` callback function to demonstrate how to recognize its execution from the peripheral's perspective. The Linux kernel calls it to mask a specific interrupt source (the `irq` argument). Given a concrete `irq` value, e.g., 3, the R/W Seq is `<MMIOR, INTC_REG_MASK, X1>`, `<MMIOW, INTC_REG_MASK, F(X1)>`, where `X1` is a symbol representing the MMIO read value and `F(X1) = X1 & 0xfffffff7`. As shown in Figure 3(c), by matching the R/W Seq, the peripheral recognizes that the Linux kernel is calling `irq_mask`. Specifically, the peripheral can figure out the IRQn 3 by checking whether `X1` (returned to the Linux kernel) and `value` (written to the peripheral) satisfies the equation `F(X1)`. After matching the R/W Seq, the peripheral recognizes the event *the Linux kernel masks the interrupt source whose number is 3* and shifts its state correspondingly.

Figure 3(b) is an implementation of `handle_irq` callback to demonstrate how to control its execution from the peripheral's perspective. Once the Linux kernel receives an interrupt request from the Interrupt Controller, it calls `handle_irq` to dispatch the request of the IRQn obtained from the Interrupt Controller. The R/W Seq is `<MMIOR, INTC_REG_STATUS, X2>`, where `X2` stands for the MMIO read value. The peripheral can control the execution times and the argument of `generic_handle_irq` by providing a crafted `X2`. In Figure 3(c), after matching the R/W Seq, the peripheral returns `0x1 << 3` and tells the Linux kernel that *the interrupt source 3 should be triggered* after reading `INTC_REG_STATUS`.

Lastly, as shown in Figure 3(d), we found that R/W Seq can behave as the state transition condition in the peripheral model by guiding the kernel's execution towards a specific code path. Intuitively, we can create a full state machine of the peripheral by manually building the general peripheral model of the states and transitions (using the Linux kernel's abstraction) and extracting R/W Seqs as the transition condition (use symbolic execution to analyze the *critical functions*).

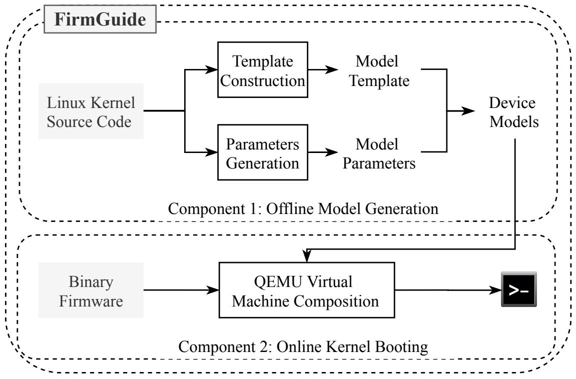


Fig. 4: FirmGuide architecture.

Model-Guided Kernel Execution Based on the above observations, we proposed a new method called *model-guided kernel execution* to emulate the Type-I peripherals. We split the peripheral model into two parts: the model template (peripheral independent part) and the model parameters (peripheral dependent part). Specifically, for each type of peripherals, we manually build its model template from the Linux kernel subsystems (Section IV-A), and the model parameters are automatically generated from the low-level driver code (Section IV-B). Therefore, our model generation for each specific peripheral works like a fill-in-the-blanks process. Finally, the generated models can guide (recognize and control) the kernel's execution to a successful rehosting.

We have developed a prototype named `FirmGuide`. Figure 4 shows its architecture. It consists of two components: *offline model generation* and *online kernel booting*. The first component analyzes the Linux kernel source code to generate peripheral models, while the second rehosts embedded Linux kernels using synthesized QEMU virtual machines equipped with the generated peripheral models. Note that, though our system requires the Linux kernel source code in the offline component, there is no need for source code in the online kernel booting. Section IV and V detail these two parts respectively.

IV. Offline Model Generation

In the offline model generation, we use model-guided kernel execution to *semi-automatically generate the peripheral models for Interrupt Controllers and Timers* with the Linux kernel source code. The generated peripheral models (C code) can be compiled with QEMU to provide the full functionality of the peripherals. Figure 5 shows its procedure. The model generation process has two parts: the manual model template construction and the automatic model parameters generation.

As shown in Figure 5, the model templates are inducted from the Linux kernel subsystems. The template for each type of peripheral contains a state machine that communicates with the Linux kernel subsystems via *critical functions*. In other words, if the model template knows exactly which *critical function* the Linux kernel has executed, it will react correctly by shifting its inner states and taking the following actions. The model template implements the nodes (states) and the unidirectional edges (state transition table) of a state machine but leaves the transition conditions as blanks (triggering of events). Note that the model template construction is a one-time effort for each type of peripherals.

The model parameters, generated by analyzing the low-level driver code, are designed to describe the transition conditions. Note that the conditions for triggering these events are the executions of specific paths of the *critical functions*. For example, in Figure 3(a), the event of masking interrupt source `irq` means the calling of `irq_mask_callback` with argument `irq`. To describe these events, we combine three methods to extract all necessary information: Basic R/W Seq Extraction (Section IV-B), CFSV Handling (Section IV-C), and Value's Semantics Inference (Section IV-D). The first one generates the basic R/W Seq of an execution path of a *critical function*. The last two complement the first method and extract additional properties of the execution path for the cases where only using R/W Seq is not enough.

A. Template Construction

The model template construction is to build a state machine containing all actions of that type of peripherals. To do so, we first studied the abstraction of the Interrupt Controller and Timer subsystems in Linux. Specifically, there are three kinds of events that can be triggered for a given peripheral, which are the Linux kernel, other connected peripherals, and the peripheral itself. For example, an Interrupt Controller has events like masking an interrupt (from the Linux kernel), requesting for firing an interrupt (from other peripherals), and selecting an interrupt to fire (from itself). Based on the defined events, we design the states and the state transition table but leave the transition condition (triggering of the events) as blanks. Note that only transition conditions from the Linux kernel are handled as blanks, while others have simple conditions like raising or lowering the interrupt signal. We manually designed eight to nine events and four to five states for the edge- and level-triggered Interrupt Controllers. Similarly, we have two models for `clkevt` Timers with nine events and four states, and `clksrc` Timers with four and two. You can find more detail about these manually built model templates in [22].

B. Basic R/W Seq Extraction

A basic R/W Seq is a sequence of MMIO read or write operations which used to represent a specific execution path of the *critical function*. Indeed, more than half of the cases only use the basic R/W Seq to represent the transition conditions. We apply symbolic execution to the *critical functions* to find out the MMIO operation sequences of the paths which can represent specific functionality (e.g., mask interrupt source whose number is 3). These operation sequences are called the basic R/W Seq. Figure 6 shows its general workflow.

Boot Context Preparation Applying symbolic execution directly to a *critical function* needs an execution context. We create a simplified booting process to get the initial context of the related kernel subsystems. Specifically, the booting process is simplified from the Linux kernel `start_kernel` but only does necessary initialization of kernel subsystems (i.e., only keep

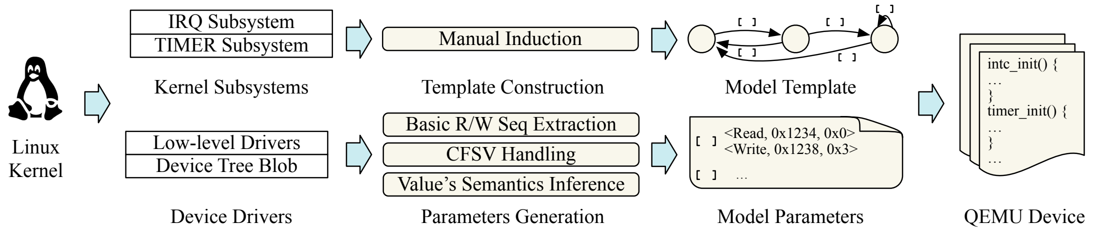


Fig. 5: Overview of *model-guided kernel execution*.

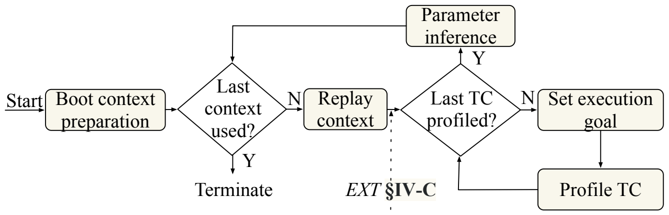


Fig. 6: General workflow of basic R/W Seq extraction. TC stands for a transition condition. The dotted arrow marks the extension mentioned in Section IV-C.

```
1 #define MASK_ADDR 0xFFFF0014
2 void irq_mask(u32 irq) {
3   u32 mask = 0x1 << irq;
4   /* The mask_cache is a CFSV, i.e.
5   * a non-local variable and a symbolic value. */
6   *mask_cache &= ~mask;
7   writel(*mask_cache, MASK_ADDR);
8 }
```

Fig. 7: A simplified CFSV example

interrupt and time subsystem initialization code). Applying the symbolic execution to this booting process, we may get multiple paths that can successfully finish the whole booting process. Each path and its memory status (initial values) is one valid boot context because multiple hardware configurations can correctly boot the Linux kernel in practice. The initial values are obtained after the constraint solving. Moreover, the context preparation also solves the problem of inferring initial values for Type-II peripherals. Note that we introduce a new symbol for each MMIO read operation in this and the following symbolic execution since it is indeed a volatile operation.

Replay Context After the boot context preparation, the symbolic execution engine (KLEE) has maintained several booting contexts in memory. Before executing *critical functions*, we additionally implement a context replay technique to recreate these booting contexts by rerunning the boot process with one valid boot context. As KLEE only uses one host CPU to perform the state exploration, for each booting context, we analyze all transition conditions independently, such that the replay technique can also make full use of the host CPU resources to speed up the analysis.

Obtain Transition Conditions The next step is to obtain the transition conditions based on the model template. For instance, the model template of Interrupt Controllers requires us to collect the transition conditions for critical functions including `irq_mask`, `irq_ack`, `irq_unmask`, etc., for each interrupt source (we obtain the list of the interrupt sources from the device tree). Each transition condition is noted as the pair `<execution goal, critical function>`. We first prepare the list of the pairs from the state machine and then obtain each transition condition by symbolically executing (profiling) the *critical function* with the execution goal. We use *a pass of analysis* to represent the enumeration of this list.

Set Execution Goal: The execution goal is to describe the desired path that implies the transition condition. Each *critical function* may have multiple paths, and the desired path is the one to finish its real functionality. For example, in Figure 3(b), to find the execution path of *the firing interrupt source whose number is 3*, the execution goal should be in the following: ❶ it must call `generic_handle_irq` once and only once; ❷ it must call `generic_handle_irq` and pass 3 as the first argument; ❸ it must return from the function without error.

Profile Transition Condition: The MMIO operations in the path that satisfy the execution goal are traced as the profile.

Parameter Inference The following rules are used to infer the basic R/W Seq from the profile. ❶ For an MMIO read, if the symbol value has constraints, we append a new node `<MMIOR, addr, value>` to the basic R/W Seq list, where the `value` is the concrete value provided by the constraint solver. This node tells the peripheral model to return a `value` when the Linux kernel reads from this MMIO address. ❷ For an MMIO read, if the symbol has no constraints, we append a new node `<MMIOR, addr, USE_LAST_VALUE>` to the basic R/W Seq list. This node tells the peripheral model to provide the last value written to the same MMIO address when reading from it. ❸ For an MMIO write, we append a new node `<MMIOW, addr, match(expr)>`, where the `match` is a function used by the model to check whether the value written by the Linux kernel satisfies the `expr`.

C. CFSV Handling

Basic R/W Seq works well in cases where there is no data exchange between two *critical functions* in a pass of analysis. From our experience, this assumption is tenable for major cases. However, the data exchange can happen between two executions of the *critical function*. Specifically, it happens through the read and the write operations to a non-local variable in two executions. A non-local variable for a function is the variable whose life cycle is longer than that function, e.g., global variables and heap variables. Figure 7 shows an example. The `mask_cache` in line 6 is a heap variable. It records the current mask status of all interrupt sources. It is updated whenever the Linux kernel invokes the `irq_mask` or `irq_unmask` and is also used in line 7. This leads to a problem that the profiled `expr` in line 7 cannot match the real scenario because the value

```
Algorithm 1: CFSV Detection
input  : $ctxt$, symbolic execution context
output : $CFSVs$, recognized CFSV set
1 init empty set $CFSVs$, $nonlocal\_rw$;
2 do
3     state_merge_begin();
4     label known $CFSVs$ in $ctxt$;
5     $records \leftarrow$ do a pass of analysis using $ctxt$;
6     $nonlocal\_rw \leftarrow$ extract memory r/w info from $records$;
7     state_merge_end();
8     update $CFSVs$ based on merged $nonlocal\_rw$;
9 while $CFSVs$ has increased;
```

written in line 7 can be changed. Therefore, the mismatch will happen, and the peripheral model cannot work properly.

CFSV Definition We first introduce a concept called *Cross Function Symbolic Variable* (CFSV). CFSV is a non-local variable that is both read and written after a pass of analysis. Since the value of a CFSV can be updated in one function and be used in another function, its value depends on the execution amount and execution order of these *critical functions*. If a *critical function* is CFSV-free, the basic R/W Seq is enough to describe the transition condition. Conversely, if not CFSV-free (read or write CFSV), we need to record CFSV read/write operation, extend the basic R/W Seq by adding the CFSV information, then emulate the CFSV in the peripheral model.

CFSV Detection Algorithm 1 depicts an algorithm that finds the CFSVs incrementally. There are several assumptions to guarantee that the algorithm can finally converge: ❶ no new allocated memory among all *critical functions*; ❷ no symbolic pointers; ❸ the *critical function* is at a reasonable size so that the symbolic execution can finish its exploration. These assumptions ensure the amount of the CFSVs is finite and fixed. Therefore the $CFSVs$ in Algorithm 1 is a monotonically increasing set with an upper bound. Consequently, the algorithm can reach a fixed point at last.

Note that line 3 and line 7 use a technique in KLEE called state merging [25]. We adjust it to support additional merges of CFSV related information. Merging state helps us aggregate the CFSV read/write operations in a pass of analysis while keeping the number of contexts unchanged. Line 4 labels known $CFSVs$ in the context and all known $CFSVs$ are regarded as fake MMIO registers, i.e., every read of CFSV introduces a new symbol. The reason is that $CFSV$ can be changed during the execution. Thus we symbolize it as any possible value (an over-approximation) to maximize the path exploration. In line 6, we collect all non-local variables' read/write information in each state and then merge the information. After that, we use the information to update the $CFSVs$. The CFSV detection is added at the dotted arrow in Figure 6.

To handle the CFSV, we emulate its data propagation in our model. Similar to the CFSV detection, we treat the CFSV as a fake MMIO register in the pass of analysis. The introduced symbols are marked with CFSV labels in the profile. Then we record its data propagation in parameter inference by extending basic R/W Seq with two new types of nodes called `<CFSVR, addr, val>` and `<CFSVW, addr, expr>`. Finally, we allocate global variables and emulate the data propagation of CFSV along with the basic R/W Seq in the peripheral model.

D. MMIO Value's Semantics Inference

The above two methods solve the problems of describing a path of *critical function*. However, to emulate a Timer, the model needs to understand the semantics of the MMIO register's value for correct reaction to some *critical functions*. One case is that our Timer model needs to understand when the Linux kernel wants to fire the next Timer interrupt. Firstly, the Linux kernel determines a value in `cycle_t` (a tick for the Timer device, the smallest unit). Then it aligns the unit with the specific Timer device. The formula of the unit conversion can be collected during the symbolic execution.

However, from the emulated Timer's perspective, understanding the time period represented by the Linux kernel written value requires us to convert back the value's unit to the common one (`cycle_t`). That means we need to perform the reverse operation for the formula. Mostly, the conversion formula is as simple as $y = k * x$ where k is a constant, and x, y are the time periods represented in two units. We can directly detect these cases and handle them in a lightweight way (hardcode their inverse functions). If we meet a complicated conversion formula, we need a constraint solver in the peripheral models. To the best of our knowledge, we did not meet a complicated formula in real-world cases.

We again extend our basic R/W Seq by adding unit conversion information. This extension helps our emulation to provide accurate time emulation for Timer peripherals.

E. Implementation Detail

In the offline model generation, the model template has $2,812$ lines of C code ($1,712$ for the Interrupt Controller, $1,100$ for Timer). The parameter generation part is based on KLEE, including $4,869$ lines of C code for static analysis, $1,902$ lines of C++ code for patching KLEE, and $1,110$ lines of Python code for gluing.

Preprocessing of Source Code `FirmGuide` leverages LLVM and KLEE [26] to perform static analysis on the Linux kernel source code. We preprocess the code for three reasons. ❶ We replace the inline assembly with the equivalent C functions because LLVM cannot analyze assembly code. ❷ We simplify some libraries' functions and change static variables to be visible to other modules to make the analysis easier. Note that this is a one-time effort since we only change common header files ❸ Moreover, these changes, targeting small functions and limited variables, do not affect the static analysis results since we do not change the code semantics.

Analysis Based on Symbolic Execution Our symbolic analysis is developed upon KLEE. The input is the target Linux kernel source code as well as its device tree. The outputs are generated model parameters. These parameters will be rendered into the model template to generate peripheral models that can be directly compiled with QEMU. In detail, we compile the target Linux kernel source code as a linked LLVM IR file and run the symbolic execution on the IR

TABLE I: Overview of representative OpenWrt subtargets.

subtarget	Source Code		Firmware			
	Rev. of OpenWrt	Ver. of LinuxKern	Architecture	# of Firmware	# of SoCs	# of Vendors
ramips/rt305x	15.05	3.18.20	mipseb	5249	4	55
ath79/generic	19.07.1	4.14.167	mipsel	613	15	24
kirkwood/generic	15.05	3.18.20	armel	482	3	6
bcm53xx/generic	15.05	3.18.20	armel	388	3	1
oxnas/generic	15.05	3.18.20	armel	176	1	4
summary	$\times 2$	$\times 2$	$\times 3$	$6,908$	26	90

file. The following strategies are applied. ❶ Control the symbolic execution's flow by feeding the bogus entry point of the linked IR. We add our `main` function to the LLVM IR file and therefore the symbolic execution will start there. Inside the `main` function, firstly the simplified kernel booting process (e.g., `time_init`, `init_IRQ`) will be executed to get the booting context, then the analysis code is launched to generate the template parameters (Figure 6). ❷ Hook an entire library of the Linux kernel by linking our customized implementation rather than the original code. Our customized libraries cover the Linux kernel's interrupt subsystem, time subsystem, device tree libraries, memory management, and `stdlib` facilities. This simplifies the symbolic execution and avoids the possible path exploration. ❸ Add auxiliary features facilitating the analysis mainly by extending the KLEE's `SpecialFunctionHandler` interface.

V. Online Kernel Booting

The online kernel booting component is implemented with several existing tools. Given a firmware image, we first retrieve the Linux kernel and the device tree from the image using `Binwalk`. We then extract the peripheral list from the device tree. For each peripheral in the list, we use its `compatible` property in the device tree entry to match the generated peripheral models in offline model generation. This is feasible because we organize the generated peripheral models by the same `compatible` property. The peripheral models (C code) will be compiled together with the QEMU. To add a new virtual machine in QEMU, we need to write a machine file in C language that initializes all peripherals. We automatically generate a generic machine file and finally use the generated QEMU virtual machine to rehost the Linux kernel along with a prepared ram file system (ramfs) that is generated by Buildroot. This component is written in Python with $5,519$ lines of code.

VI. Evaluation

A. Experiment Setup

To evaluate the effectiveness of our system, we conduct experiments using firmware images downloaded from OpenWrt covering multiple SoCs. The source code for a family of SoCs is organized in a subtarget. We select the top five subtargets covering 26 SoCs and 90 vendors following three criteria: they support device tree; they cover different architectures and endianness; they are popular due to a number of released firmware images [17,18]. Table I shows the details of our dataset. The number of *ramips/rt305x* firmware images is larger than others because OpenWrt has more firmware images released for *ramips/rt305x*. Note that we conduct experiments on OpenWrt because this dataset has covered diverse SoCs, vendors, architectures and kernel versions, making the dataset representative.

We conducted all experiments on a server with two Intel Xeon Silver 4114 processors, 128GB RAM, Ubuntu 16.04.6 LTS system. We have released the Docker image of the generated QEMU virtual machines and part of the firmware images used in our evaluation [22].

B. Offline Model Generation

Model Parameters Generation As shown in Table II, `FirmGuide` generates model parameters for 9 Type-I peripherals. Note that we do not generate parameters for Timers (marked as `not necessary`) in two MIPS subtargets. Because the QEMU MIPS processors have implemented these timers. Specifically, the 4th column shows the number of paths that the symbolic execution engine explores during the model generation. The 5th column shows the final solutions our system finds for a successful rehosting. The longest time to find the first solution is still within one hour showing that *our approach is faster than developing the peripheral models manually*. The 6th column details the time to get the first solution and all solutions. The 7th column gives the existence of CFSV. The 8th column presents the unit conversion formula used in the Timer for that subtarget. The x represents a time interval that aligns with the specific Timer's unit, y is the value in terms of `cycle_t`. We use x_1, x_2 to represent two registers that the peripheral uses to present the time. The last column lists the lines of the code for the generated peripheral models.

Type-II Peripheral For Type-II peripherals, we automatically generate their peripheral models without state machines but with the proper initial values of their hardware registers. The values are calculated in the boot context preparation (Section IV-B). Table III shows the number of Type-II peripherals emulated and of those having non-zero initial values. In total, 10 Type-II peripherals out of 64 have non-zero initial values.

C. Online Kernel Booting

Besides offline model generation, we conduct experiments to rehost a number of firmware images. The results show that our generated QEMU machines can successfully rehost more than 95% of the Linux kernels. During the evaluation, we check whether the Linux kernel image has switched to user mode by parsing the CPU register file traces provided by QEMU debug option `-d cpu`. We also check whether the ramfs has spawned a shell by detecting the booting messages, i.e., `Welcome to Buildroot`. The overall result is shown in Table IV. In total, for $6,908$ firmware images, $6,192$ of them are successfully unpacked. We retrieved $6,188$ Linux kernels. Among them, $5,947$ (96.11%) enter the user space, and $5,469$ (88.38%) successfully spawn shells. We manually analyzed the reasons for the failed cases.

Triggering of a Double-free Bug† Some Linux kernels in kirkwood/generic suffer from a double-free bug in the function `orion_nand_probe` [27], where `clk_put` will be called twice

TABLE II: Results of offline machine model generation.

Subtarget	Interrupt Controller	Timer	# of Paths	# of Solutions	First/All Solution (s)	Exists CFSV (y/n)	Timer Semantics	LoC
ramips/rt305x	ralink-rt2880-intc	`not necessary`	262	4	1/2	n	-	3,366
ath79/generic	qca,ar7240-intc	`not necessary`	110,083	1,134	5/943	n	-	4,138
kirkwood/generic	marvell,orion-intc marvell,orion-bridge-intc	marvell,orion-timer	132	2	2/3	y	$y = \sim x$	4,790
bcm53xx/generic	arm,cortex-a9-gic	arm,cortex-a9-global-timer arm,cortex-a9-twd-timer	150,336	2,592	2,027/24,070	y	$y = x_1 << 32 + x_2$	3,537
oxnas/generic	arm,arm11mp-gic	arm,arm11mp-twd-timer plxtech,nas782x-rps-timer	52,332	1,246	914/16,184	y	$y = x$	3,366

TABLE III: The fraction of Type-II peripherals with non-zero initial values of all Type-II peripherals emulated in each subtarget.

Subtarget	ramips/ rt305x	ath79/ generic	kirkwood/ generic	bcm53xx/ generic	oxnas/ generic
count	1/10	2/15	3/26	2/4	2/9

if the Flash device does not exist. This bug exists in the Linux kernel before 4.9. Interestingly, this bug will not be triggered on a physical router device, since it usually uses the Flash device as the external storage. However, it is triggered in `FirmGuide` since we only have a dummy Flash device in the emulator.

Unsupport of the Root Filesystem✦ Some Linux kernels in oxnas/generic do not support the ramfs that is supported by the Linux kernel by default. We suspect that the support of this file system is removed to reduce the image size.

D. Firmware Diversity

We further study the scalability of `FirmGuide` by exploring the diversity of the rehosted Linux kernels. Figure 8 shows that `FirmGuide` can rehost diverse embedded Linux kernels.

Architecture `FirmGuide` supports Linux kernel images in ARM32 with little-endian, MIPS32 with both big- and little-endian (Figure 8a). Our system is architecture-independent.

Kernel Version `FirmGuide` can rehost 4 major and 22 distinct minor versions of embedded Linux kernels, no matter when they were released, showing that the generated model is not binding to the particular kernel version (Figure 8b).

Firmware Format We extended Binwalk to support seven firmware formats listed in Figure 8c. Among them, the legacy uImage is the most popular firmware format.

Firmware Size The size of supported firmware varies from 3 MB to 16 MB and the average is about 3.6MB (Figure 8d).

SoC and Vendor As shown in Table IV, `FirmGuide` has supported 26 SoCs. In Figure 8e, we list the number of firmware images for the top ten vendors.

E. Functionality

We conducted two experiments to demonstrate the functionality of the firmware booted by `FirmGuide`. First, we use the syscall testings of LTP (Linux Testing Project) to these booted firmware images. LTP contains testings of file systems, IO, memory management, scheduler, etc. In total, for all $1,259$ system call tests, $1,049$ of them have been passed, 164 of them were skipped since these tested system calls have not been introduced at the kernel version of these booted firmware images, and 46 of the tests failed. We analyzed the failed tests and summarized the failure reason (Table VI). Most failures are caused by the dummy network devices or non-implemented

TABLE IV: Results of online kernel booting. Unpack: Number of unpacked firmware images. Kernel: Number of the Linux kernels detected. User Space: Number of the Linux kernels entering into user space. Shell: Number of the Linux kernels spawning shells.

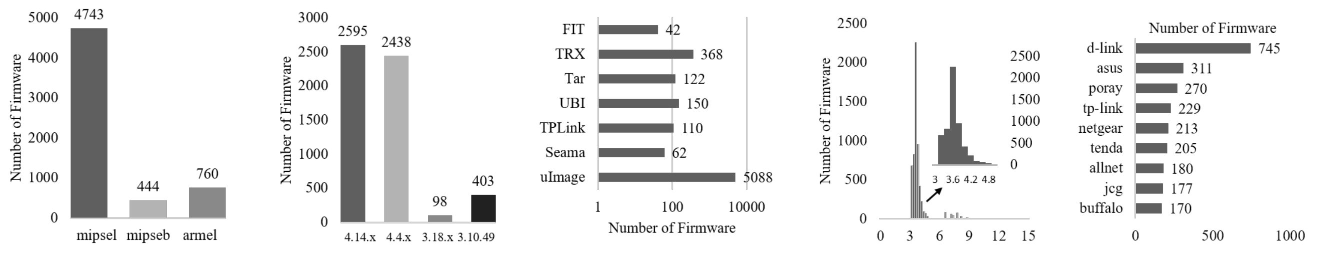
SoC	Unpack	Kernel	Booting Validation	
			User Space	Shell
Ralink RT3050	1164	1164	1144 (98.28%)	1052 (90.38%)
Ralink RT3052	1815	1815	1815 (100.00%)	1661 (91.52%)
Ralink RT3352	173	173	173 (100.00%)	157 (90.75%)
Ralink RT5350	1632	1632	1611 (98.71%)	1475 (90.38%)
subtarget: ramips/rt305x	4784	4784	4743 (99.14%)	4345 (90.82%)
Atheros AR7161	36	36	20 (55.56%)	20 (55.56%)
Atheros AR7241	20	20	12 (60.00%)	12 (60.00%)
Atheros AR7242	24	24	24 (100.00%)	24 (100.00%)
Atheros AR9330	4	4	4 (100.00%)	4 (100.00%)
Atheros AR9331	24	24	12 (50.00%)	12 (50.00%)
Atheros AR9341	10	10	4 (40.00%)	4 (40.00%)
Atheros AR9342	24	24	24 100.00%)	24 (100.00%)
Atheros AR9344	70	70	64 (91.43%)	64 (91.43%)
Qualcomm Atheros QCA9531	22	22	16 (72.73%)	16 (72.73%)
Qualcomm Atheros QCA9533	41	41	14 (34.15%)	14 (34.15%)
Qualcomm Atheros QCA9557	64	64	64 (100.00%)	64 (100.00%)
Qualcomm Atheros QCA9558	54	54	50 (92.59%)	50 (92.59%)
Qualcomm Atheros QCA9560	16	16	16 (100.00%)	16 (100.00%)
Qualcomm Atheros QCA9561	18	18	14 (77.78%)	14 (77.78%)
Qualcomm Atheros QCA9563	114	114	106 (92.98%)	106 (92.98%)
subtarget: ath79/generic	541	541	444 (82.07%)	444 (82.07%)
Broadcom BCM4708A0	241	241	241 (100.00%)	241 (100.00%)
Broadcom BCM4709A0	128	128	128 (100.00%)	128 (100.00%)
Broadcom BCM47189	19	19	19 (100.00%)	19 (100.00%)
subtarget: bcm53xx/generic	388	388	388 (100%)	388 (100%)
Marvell 88F6192	20	20	20 (100.00%)	20 (100.00%)
Marvell 88F6281	208	204	204 (100.00%)	144 (70.59%)
Marvell 88F6282	102	102	100 (98.04%)	80 (78.43%)
subtarget: kirkwood/generic	330	326	324 (99.39%)	244 (74.85%) †
PLX NAS7820	149	149	48 (32.21%)	48 (32.21%)
subtarget: oxnas/generic	149	149	48 (32.21%)	48 (32.21%) ✦
Overall	6,192	6,188	5947 (96.11%)	5469 (88.38%)

TABLE V: Results of system call tests.

Models	Pass	Skipped	Failed	Total
Fully generated	1049	164	46	1259
Ground Truth	1049	164	46	1259

system calls (the Linux kernel contained in that firmware does not support that syscall).

Second, we compared the functionality of the peripherals generated by `FirmGuide` with the manually written peripherals. *plxtech,nas7820* device is used for the comparison. To manually write the emulation code of peripherals, we first learn the driver source code of the Type-I peripherals, then write the QEMU emulation based on human understanding. It costs around 1 week/person to write the emulation code of the Type-I peripherals for *plextec,nas7820* device (assuming has already learned the driver development background, and the cost time



(a) Architecture (b) Kernel Version (c) Firmware Format (d) Firmware Size (e) Top-10 Vendors

Fig. 8: We can rehost Linux kernels in different architectures, Linux kernel versions, firmware formats, firmware sizes (MB), and firmware vendors. In Figure 8b, 4.14.x has 8 sub-versions, 4.4.x has 11 and 3.18.x has 2 sub-versions; there are 22 distinct versions in total.

TABLE VI: Reasons of failed system call tests.

Reason	Count
Bugs or vulnerabilities of the Linux kernel are not patched	6
Network device is not available	14
Some syscalls are not implemented	20
Other	6
Total	46

TABLE VII: Results of testing 6 CVEs for the rehosted kernel. ✓ indicates a successful trigger and † indicates a successful exploit, while ✗ indicates a failure. The other three symbols (□ ◊ △) represent different versions of the Linux kernel on which we triggered or exploited the vulnerabilities (□ 3.10.49, ◊ 3.18.20, △ 4.4.42).

CVE ID	CVE Type	Status	Version
CVE-2016-5195	Race Condition	✗	N/A
CVE-2016-8655	Race Condition	✓ †	□
CVE-2016-9793	Integer Overflow	✓	□ ◊
CVE-2017-7038	Integer Overflow	✓ †	◊
CVE-2017-1000112	Buffer Overflow	✓ †	△
CVE-2018-5333	NULL Pointer Dereference	✓ †	□ ◊

includes both the code development and debugging). Table V shows the results where Ground Truth represents the manually written emulation code. The results show that the generated board has identical functionality as the manually written one.

VII. Applications

We applied two security applications on `FirmGuide` to demonstrate its usage scenarios. Note that these applications are demonstrative and not our main contribution. Other tools [28–31] that build upon QEMU, e.g., S2E [32], can also be applied.

Linux Kernel Vulnerability Analysis We first collected 6 Linux kernel vulnerabilities (shown in Table VII) and then analyzed them on several Linux kernels targeting a *plxtech,nas7820* device with the *OX820 NAS7820* SoC (ARM). We then used `FirmGuide` to rehost the embedded Linux kernels. Next, with the debugging capability of QEMU, we managed to trigger 5 of them, develop 4 Linux kernel exploits, and show the reasons for failed cases.

Vulnerability Triggering After inspecting this new execution environment with the debugging capabilities of QEMU, 5 vulnerabilities have been successfully triggered, laying a foundation for further exploitation. The reason that CVE-2016-5195 (a.k.a DirtyCow) cannot be triggered is the limitation of the target firmware itself. The Linux kernels inside these firmware images do not support *madvise* system call which is necessary to trigger the race condition.

Vulnerability Understanding and Exploiting For the rehosted vulnerable Linux kernels, we use the debugging capabilities of QEMU to understand the vulnerabilities, search the exploitable function pointers and successfully develop 4 exploits. For example, CVE-2017-1000112 [33], a buffer overflow vulnerability, is triggered when the packet processing routine switches from the UFO (UDP Fragment Offload) path to the non-UFO one. To exploit it, the $skb\_prev \rightarrow len$ and the offset of the hi-jacked pointer have to be carefully calculated. We used the QEMU remote GDB to find an appropriate overflow size. The detailed process of the exploit and the usage of the GDB are shown in [22].

Fuzzing We also ported two fuzzing tools, TriforceAFL [21] and UnicoreFuzz [20] to demonstrate the usage scenario of `FirmGuide`. Due to the page limitation, the detailed running status of these fuzzing tools is shown in [22]. Note that `FirmGuide` doesn't focus on any specific fuzzing technique but provides the infrastructure of dynamic analysis platforms towards embedded Linux kernels.

VIII. Discussion

Threats to Validity The threats to validity comes from three aspects. ❶ In our method, the model templates should be manually built by human experts. This manual process possibly can be the bottleneck if the state machines of some complex Type-II peripherals are too complex. ❷ To migrate the method to the same or other peripherals in the same or other OSes, our method requires a peripheral interface at a suitable layer in the target OS. The layer is suitable when it is a common interface and its functions have clear semantics of the peripheral; otherwise, the R/W Seq cannot effectively guide (recognize and control) the kernel's execution. ❸ We use symbolic execution to find usable booting context and R/W Seq. If the *critical function* is complex, the parameter generation process may fail due to the path explosion. However, from our experience in supporting Linux, path explosion is not a fundamental threat as the *critical function* tends to be simple. Besides, our context replay in Section IV can make the generation process running in parallel (KLEE itself doesn't support running in parallel). Furthermore, we don't need to find all solutions for a given source code as finding one satisfying path for the boot process and one R/W Seq for each state transition condition is enough.

Full Emulation of Type-II Peripherals Full emulation of Type-II peripherals can enable more interesting functionalities of the rehosted Linux kernel like the network facility and make the rehosted Linux kernel complete (pass all syscall tests in LTP in evaluation). Since dummy Type-II peripherals do not affect the successful rehosting of the Linux kernel, full emulation of them is out of the current work's scope. However, the lack of Type-II peripherals limits the ability for conducting dynamic analysis on the code that relies on the functionality of these peripherals, e.g., fuzzing on these drivers. Therefore, we leave the support of Type-II peripherals as future work. From our perspective, fully emulating more Type-II peripherals requires more engineering efforts and perhaps the current emulation methods should be adjusted or further improved according to the target peripheral's specification. Note that using dummy Type-II peripherals doesn't mean their drivers in the rehosted Linux kernel cannot be dynamically analyzed. Specifically, the hardware-independent code in these drivers can still be analyzed as long as their driver is successfully initialized (as shown in Ex-vivo [34]). Using the shell provided by `FirmGuide`, these parts of code can be directly tested through their user-space interfaces, e.g, related `ioctl` system calls.

IX. Related Work

Firmware Rehosting Firmware rehosting is to dynamically run the firmware on a virtual execution environment. One type of firmware re-hosting focuses on transferring code execution between the virtual execution engine like QEMU and the physical device. In this way, Avatar [16], Prospect [35], Surrogates [36], and Kammerstetter et al. [37] boosted the capability of the dynamic analysis of embedded system firmware. These systems are precise but suffer from hardware availability and debug interface availability.

The other approaches aim at full system emulation. For bare-metal devices, Gustafson et al. [38] managed to replace real devices by modeling the code execution between QEMU and the physical hardware traced by Avatar. Later, P2IM [13] collected the instruction trace and then instantiated predefined models. HALucinator [14] identified HAL APIs in firmware and replaced them without hardware emulation. P2IM and HALucinator successfully performed dynamic analysis on the firmware of bare-mental systems without physical devices. Our work instead focuses on the Linux kernel.

For Linux-based devices, Firmadyne [5] provides a utility to dynamically analyze the user-space programs. Other similar systems [11,12,39,40] also focus on user-space programs. `FirmGuide` is different from them because we can rehost and analyze the original Linux kernels inside the firmware, while others cannot. LuaQEMU [41] manipulated the code execution by a Lua script [41], while `FirmGuide` would not change it. Partemu [42] extends QEMU to support the analysis of TrustZone OSes rather than the Linux kernel. Simics [43] is a full-system clock-accurate simulator that is popular in hardware/software co-design. Theoretically, `FirmGuide` can be used to further facilitate Simics by semi-automatically generating the device emulation code using Simics's own device modeling language.

Firmware Analysis Researchers propose several static analysis tools to disclose vulnerabilities in the firmware [4,44–48]. Meanwhile, dynamic firmware analysis systems are also developed [12–14,39]. `FirmGuide` enables the dynamic analysis of the Linux kernel on embedded systems, which can be leveraged to develop dynamic security analysis frameworks.

Application with QEMU Besides firmware rehosting and analysis, QEMU is also applied in other scenarios, e.g., malware detection [28–31] and forensics [49–51]. However, previous QEMU-based applications mainly target desktop or Android. `FirmGuide` aims to support embedded Linux kernels, which enriches the security applications in this area.

X. Conclusion

In this paper, we proposed a new technique called *model-guided kernel execution* to rehost Linux kernels of embedded firmware. It leverages kernel's abstraction for peripherals and kernel-peripheral interactions to *semi-automatically* generate stateful peripheral models. Then the generated model can be used to synthesize QEMU virtual machines to rehost embedded Linux kernels. We have implemented a prototype system called `FirmGuide`. It generates 9 peripheral models with full functionality and 64 with minimum functionality covering 26 SoCs. Our evaluation with $6,188$ firmware images downloaded from the Internet shows that it can successfully rehost more than 95% Linux kernels covering 2 architectures and 22 versions. Two security applications have been applied on the rehosted kernels to demonstrate `FirmGuide` can build the foundation of dynamic analysis tools for embedded Linux kernels.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. Special thanks go to Mathias Payer, Zhi Wang for their constructive suggestions and comments. This work was partially supported by the National Natural Science Foundation of China (Grant No.61872438), the Fundamental Research Funds for the Central Universities (Zhejiang University NGICS Platform, K20200019), Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (No. 2018R01005), HK RGC Project (No. PolyU 152239/18E), the National Research Foundation, Singapore under its the AI Singapore Programme (AISG2-RP-2020-019), the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001. Any opinions, findings, and conclusions expressed in this paper are of the authors and do not necessarily reflect the views of funding agencies.

References

[1] M. Michael, "Attack landscape h1 2019: Iot, smb traffic abound," 2019, https://blog.f-secure.com/attack-landscape-h1-2019-iot-smb-traffic-abound/.

[2] J. Sattler, “Attack landscape h2 2019: An unprecedented year for cyber attacks,” 2020, https://blog.f-secure.com/attack-landscape-h2-2019-an-unprecedented-year-cyber-attacks/.

[3] Eclypsium, “Assessing enterprise firmware security risk in 2020,” 2020, https://eclypsium.com/2020/01/20/assessing-enterprise-firmware-security-risk/.

[4] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, “A large-scale analysis of the security of embedded firmwares,” in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014, pp. 95–110.

[5] D. D. Chen, M. Woo, D. Brumley, and M. Egele, “Towards automated dynamic analysis for linux-based embedded firmware,” in *23rd Annual Network and Distributed System Security Symposium (NDSS)*, vol. 16, 2016, pp. 1–16.

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

[7] Z. Zhang, H. Zhang, Z. Qian, and B. Lau, “An investigation of the android kernel patch ecosystem,” in *Proceedings of the 30th USENIX Security Symposium (USENIX Security)*, 2021.

[8] Z. Jiang, Y. Zhang, J. Xu, Q. Wen, Z. W. Wang, X. Zhaneg, X. Xiang, M. Yang, and Z. Yang, “Pdiff: Semantic-based patch presence testing for downstream kernels,” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020, pp. 1149–1163.

[9] J. Lawall, D. Palinski, L. Gnirke, and G. Muller, “Fast and precise retrieval of forward and back porting information for linux device drivers,” in *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017, pp. 15–26.

[10] S. M. S. Talebi, H. Tavakoli, H. Zhang, Z. Zhang, A. A. Sani, and Z. Qian, “Charm: Facilitating dynamic analysis of device drivers of mobile systems,” in *Proceedings of the 27th USENIX Security Symposium (USENIX Security)*, 2018, pp. 291–307.

[11] A. Costin, A. Zarras, and A. Francillon, “Automated dynamic firmware analysis at scale: a case study on embedded web interfaces,” in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016, pp. 437–448.

[12] Y. Zheng, A. Davanian, H. Yin, C. Song, H. Zhu, and L. Sun, “Firmafl: high-throughput greybox fuzzing of iot firmware via augmented process emulation,” in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019, pp. 1099–1114.

[13] B. Feng, A. Mera, and L. Lu, “P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling,” *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2019.

[14] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, “Halucinator: Firmware re-hosting through abstraction layer emulation,” in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[15] W. L. Li, L. Guan, J. Lin, J. Shi, and F. Li, “From library portability to para-rehosting: Natively executing microcontroller software on commodity hardware,” in *28th Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[16] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, “Avatar: A framework to support dynamic security analysis of embedded systems' firmwares,” in *21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[17] OpenWrt, “Openwrt downloads,” 2020, https://downloads.openwrt.org/.

[18] ——, “Openwrt archive,” 2020, https://archive.openwrt.org/.

[19] “Testing linux, one syscall at a time.” https://linux-test-project.github.io/.

[20] D. Maier, B. Radtke, and B. Harren, “Unicorefuzz: On the viability of emulation for kernelspace fuzzing,” in *Proceedings of the 9th USENIX Workshop on Offensive Technologies (WOOT)*, 2019.

[21] J. Hertz and T. Newsham, “Afl/qemu fuzzing with full-system emulation.” 2016, https://github.com/nccgroup/TriforceAFL.

[22] Q. Liu and C. Zhang, “cyruscyliu/firmguide-demo: Demo of firmguide for ase2021.” 2021, https://github.com/cyruscyliu/firmguide-demo.

[23] ——, “cyruscyliu/firmguide: Source code of firmguide.” 2021, https://github.com/cyruscyliu/firmguide.

[24] F. Bellard, “Qemu, a fast and portable dynamic translator,” in *2005 USENIX Annual Technical Conference (USENIX ATC)*, 2005, p. 46.

[25] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, “Efficient state merging in symbolic execution,” in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012, p. 193–204.

[26] C. Cadar, D. Dunbar, D. R. Engler *et al.*, “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 (OSDI)*, 2008, pp. 209–224.

[27] G. Kroah-Hartman, “mtd: nand: orion: fix clk handling,” 2017, https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg1403695.html.

[28] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, “Panorama: capturing system-wide information flow for malware detection and analysis,” in *Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS)*, 2007, pp. 116–127.

[29] L. K. Yan and H. Yin, “Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis,” in *Proceedings of the 21th USENIX Security Symposium (USENIX Security)*, 2012, pp. 569–584.

[30] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song, “Dynamic spyware analysis,” in *2007 USENIX Annual Technical Conference (USENIX ATC)*, 2007, pp. 233–246.

[31] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, “Scalable, behavior-based malware clustering,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, vol. 9, 2009, pp. 8–11.

[32] V. Chipounov, V. Kuznetsov, and G. Candea, “S2E: A platform for in-vivo multi-path analysis of software systems,” in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011, pp. 265–278. [Online]. Available: https://doi.org/10.1145/1950365.1950396

[33] A. Konovalov, “Cve-2017-1000112: Exploitable memory corruption due to ufo to non-ufo path switching,” 2017, https://www.openwall.com/lists/oss-security/2017/08/13/1.

[34] I. Pustogarov, Q. Wu, and D. Lie, “Ex-vivo dynamic analysis framework for android device drivers,” in *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 2020, pp. 1088–1105.

[35] M. Kammerstetter, C. Platzer, and W. Kastner, “Prospect: peripheral proxying supported embedded code testing,” in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2014, pp. 329–340.

[36] K. Koscher, T. Kohno, and D. Molnar, “Surrogates: Enabling near-real-time dynamic analyses of embedded systems,” in *Proceedings of the 9th USENIX Workshop on Offensive Technologies (WOOT)*, 2015.

[37] M. Kammerstetter, D. Burian, and W. Kastner, “Embedded security testing with peripheral device caching and runtime program state approximation,” in *10th International Conference on Emerging Security Information, Systems and Technologies (SECUWARE)*, 2016.

[38] E. Gustafson, M. Muench, C. Spensky, N. Redini, A. Machiry, Y. Fratantonio, D. Balzarotti, A. Francillon, Y. R. Choe, C. Kruegel, and G. 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 (RAID)*, 2019, pp. 135–150.

[39] P. Srivastava, H. Peng, J. Li, H. Okhravi, H. Shrobe, and M. Payer, “Firmfuzz: Automated iot firmware introspection and analysis,” in *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things (IoT S&P)*, 2019, p. 15–21.

[40] therealsaumil, “Arm-x firmware emulation framework,” 2019, https://github.com/therealsaumil/armx.

[41] R. Nico, “Emulation and exploration of bcm wifi frame parsing using luaqemu,” 2017, https://comsecuris.com/blog/posts/luaqemu_bcm_wifi/.

[42] L. Harrison, H. Vijayakumar, R. Padhye, K. Sen, M. Grace, R. Padhye, C. Lemieux, K. Sen, L. Simon, H. Vijayakumar *et al.*, “Partemu: Enabling dynamic analysis of real-world trustzone software using emulation,” in *Proceedings of the 29th USENIX Security Symposium (USENIX Security)*, 2020.

[43] W. R. Systems, “Wind river simics,” 2021, https://www.windriver.com/products/simics/.

[44] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, “Scalable graph-based bug search for firmware images,” in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 480–491.

[45] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. 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 (CCS)*, 2017, pp. 363–376.

[46] X. Wang, Y. Sun, S. Nanda, and X. Wang, “Looking from the mirror: evaluating iot device security through mobile companion apps,” in *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019, pp. 1151–1167.

[47] Q. N. A. LAU Kai Jern, “Qiling framework - advanced binary emulation framework,” 2020, https://www.qiling.io/.

[48] N. Redini, A. Machiry, R. Wang, C. Spensky, A. Continella, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Karonte: Detecting insecure multi-binary interactions in embedded firmware,” in *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 2020, pp. 431–448.

[49] X. Jiang, X. Wang, and D. Xu, “Stealthy malware detection through vmm-based” out-of-the-box” semantic view reconstruction,” in *Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS)*, 2007, pp. 128–138.

[50] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, “Copperdroid: Automatic reconstruction of android malware behaviors,” in *22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[51] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, “Virtuoso: Narrowing the semantic gap in virtual machine introspection,” in *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P)*, 2011, pp. 297–312.