What You See is Not What You Get: Revealing Hidden Memory Mapping for Peripheral Modeling

Jun Yeon Won
The Ohio State University
won.126@osu.edu

Haohuang Wen
The Ohio State University
wen.423@osu.edu

Zhiqiang Lin
The Ohio State University
zlin@cse.ohio-state.edu

ABSTRACT

Nowadays, there are a massive number of embedded Internet-of-Things (IoT) devices, each of which includes a microcontroller unit (MCU) that can support numerous peripherals. To detect security vulnerabilities of these embedded devices, there are a number of emulation (or rehosting) frameworks that enable scalable dynamic analysis by using only the device firmware code without involving the real hardware. However, we show that using only the firmware code for emulation is insufficient since there exists a special type of hardware-defined property among the peripheral registers that allows the bounded registers to be updated simultaneously without CPU interventions, which is called the *hidden memory mapping*. In this paper, we demonstrate that existing rehosting frameworks such as P2IM and μEMU have incorrect execution paths as they fail to properly handle hidden memory mapping during emulation. To address this challenge, we propose the first framework AutoMap that uses a differential hardware memory introspection approach to automatically reveal hidden memory mappings among peripheral registers for faithful firmware emulation. We have developed AutoMap atop the Unicorn emulator and evaluated it with 41 embedded device firmware developed based on the Nordic MCU and 9 real-world firmware evaluated by μEMU and P2IM on the two STMicroelectronics MCUs. Among them, AutoMap successfully extracted 2,359 unique memory mappings in total which can be shared through a knowledge base with the rehosting frameworks. Moreover, by integrating AutoMap with μEMU, AutoMap is able to identify and correct the path of the program that will not run on the actual hardware.

CCS CONCEPTS

• **Security and privacy → Embedded systems security**; **Software security engineering**.

KEYWORDS

Firmware analysis, Embedded devices, Firmware emulation, Peripheral modeling

ACM Reference Format:
Jun Yeon Won, Haohuang Wen, and Zhiqiang Lin. 2022. What You See is Not What You Get: Revealing Hidden Memory Mapping for Peripheral Modeling. In *25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022), October 26–28, 2022, Limassol, Cyprus.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3545948.3545957

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
RAID 2022, October 26–28, 2022, Limassol, Cyprus
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9704-9/22/10…$15.00
https://doi.org/10.1145/3545948.3545957

1 INTRODUCTION

Today, smart embedded devices (*e.g.*, smart locks, smart lights, and smart plugs) are ubiquitous, due to the rapid growth of the Internet-of-Things (IoT). It is anticipated that the number of smart embedded devices will reach 30.9 billion by 2025 [45]. For an embedded device, it usually comes with a microcontroller unit (MCU), which can sense and process input from the environment through a variety of peripherals according to the computation defined by the manufacturers. There are various vendors for embedded devices, such as Nordic [12] and STMicroelectronics [17], which provide integrated MCU hardware and software development kits (SDKs) to facilitate third-party developers (*e.g.*, IoT device vendors) to produce embedded devices.

Fundamentally, an embedded device is controlled by the firmware code programmed on its MCU, which inevitably contains vulnerabilities just as other computer software does. To detect potential vulnerabilities in these devices before they are exploited for malicious purposes, many analysis techniques can be used to vet the device firmware, such as static analysis (*e.g.*, [27, 31, 40, 50, 53, 57]) and dynamic analysis (*e.g.*, [39, 44, 51]). In particular, as fuzzing [56] has become increasingly popular for efficient bug and vulnerability hunting, enabling dynamic analysis of embedded device firmware becomes the critical foundation. To increase the scalability of analysis, many dynamic analysis approaches execute the firmware without actual hardware, which is known as emulation or rehosting [32]. However, there are many challenges in enabling firmware emulation due to the proprietary and sophisticated dependencies of hardware, and an emulation framework must properly model specific hardware, especially the peripherals. Otherwise, the program is likely to stall infinitely or enter invalid states. As a result, many emulation frameworks have been proposed to tackle this challenge [30, 32, 33, 37, 46, 60]. For example, μEMU [60] proposes an invalidity-guided symbolic execution for peripheral modeling.

While existing rehosting frameworks can certainly be adopted to emulate an MCU embedded device firmware, we find that they still fall short in executing incorrect program paths that should never be executed on actual hardware. Specifically, we notice a hidden hardware property that allows the registers in different (or the same) peripherals to be updated simultaneously without CPU intervention [3], and we call it *hidden memory mapping*. In other words, when the value of a certain peripheral register changes, many other peripheral registers are also updated automatically. Without properly handling hidden memory mappings, an emulator is likely to model the peripherals in the firmware incorrectly, which can further lead to false program states due to executing

wrong program paths. In this paper, we show that even state-of-the-art rehosting frameworks such as P2IM [33] and μEMU [60] have incorrect peripheral modeling results as they do not take hidden memory mapping into consideration. Fundamentally, as such hidden memory mapping is completely defined by hardware and is not visible from the software's (*i.e.*, firmware) perspective, using just the firmware code for emulation is not sufficient.

However, to the best of our knowledge, none of the prior works notices hidden memory mapping, and their proposed techniques cannot handle this property. Therefore, in this paper, we propose the first framework, AutoMap, to automatically reveal hidden memory mappings among peripheral registers for faithful firmware emulation. It is not trivial to design and implement AutoMap and we need to address three main challenges. First, hidden memory mappings do not require CPU intervention, and thus cannot be directly inferred from the device firmware code. Second, hidden memory mappings may have sophisticated dependencies on specific peripheral registers, and it will lead to incorrect mapping results if those dependencies are not properly handled in advance. For instance, we discover that some mappings will be enabled only when another (or some other) peripheral register is set to be a specific value. Third, hidden memory mapping has bit-level granularity, which means updating a single bit of a peripheral register can affect other registers. Therefore, acquiring the whole memory mapping knowledge base takes a significant amount of time.

We have overcome the aforementioned challenges and implemented AutoMap on top of the Unicorn emulator [48]. First, to trigger hidden memory mapping, we must rely on the corresponding MCU hardware, as it is not visible from the firmware code. Consequently, AutoMap uses a differential hardware introspection approach to extract the hidden memory mappings. Second, to handle sophisticated mapping dependencies, we replicate the previously executed peripheral register writes in the firmware to reliably trigger memory mappings, as we observe that the dependencies will be naturally handled by the firmware code logic. Finally, we propose an on-demand approach that only resolves the memory mappings of the specific register values only when needed, to avoid revealing a massive number of hidden memory mappings that are unnecessary for emulation. The resolved memory mappings will be stored in a knowledge base, which can be reused to emulate firmware on the same MCU hardware.

To evaluate AutoMap, we utilize 50 MCU-based firmware from three different vendors to show how well AutoMap can discover hidden memory mappings, including 41 example firmware from the NRF52832 MCU and 9 real-world firmware from P2IM and μEMU's dataset [33, 60] which are developed for two other MCUs, STM32F103 and STM32F429. The results of our experiment show that AutoMap can extract 2,359 unique memory mappings (47 mappings per firmware on average) in total among these firmware. Moreover, we show that the knowledge base method can significantly decrease the time up to 97.01% to model memory mappings in AutoMap. Furthermore, we demonstrate a use case by integrating AutoMap to μEMU. With AutoMap, μEMU can execute more basic blocks and even previously uncovered blocks (up to 15.6%) on all 5 firmware tested than μEMU alone.

Contributions. Our paper makes the following contributions.

- **Novel Findings**. We discover special bindings between peripheral registers called *hidden memory mapping*. We show that state-of-the-art rehosting frameworks would miss branches or execute invalid branches without properly handling hidden memory mappings.
- **New Tool**. We propose AutoMap, the first tool to reveal hidden memory mapping of peripheral registers on-demand for accurate firmware emulation.
- **Empirical Evaluation**. We have evaluated AutoMap with 41 example and 9 real-world firmware, where it extracted 2,359 unique hidden memory mappings. We also show that the knowledge base can improve AutoMap's efficiency by up to 97.01%. We demonstrate a use case by integrating AutoMap to μEMU and it is capable of executing many uncovered basic blocks (up to 15.6%) than μEMU alone.

2 BACKGROUND

2.1 Peripherals and Their Internals

MCU embedded devices have integrated a great number of peripherals, such as power, universal asynchronous receiver/transmitter (UART), and general purpose input and output (GPIO), each of which is responsible for a specific task. For instance, the power peripheral supports global system ON and OFF modes. Each peripheral plays an important role during firmware execution, as the firmware frequently examines the peripheral register values at run-time to check their status. Each peripheral occupies a small memory region and consists of hundreds or thousands of *peripheral registers*, which have different roles such as enabling and disabling interrupts. At a high level, these peripherals can be classified into vendor-specific and platform-specific peripherals, and we explain them in detail as follows.

Vendor-specific Peripherals. The memory layout of MCU is specified by its vendor, and diverse vendor-specific peripherals reside in a memory region. For instance, in the Nordic NRF52832 MCU [8], every peripheral locates in a memory region from `0x40000000` to `0x60000000` and each peripheral occupies 4,096 bytes and includes 1,024 registers at most. A peripheral register has 32 bits, and each bit has its specific usage. For example, according to the Nordic MCU specification [6], the register enabling interrupt of the clock peripheral uses only its first four bits, while the register indicating the starting status of the clock uses only the first bit. In addition, each peripheral register has a unique initial value (*e.g.*, NRF52832 PIN_CNF [9]), and only a few of them have non-zero initial values.

Platform-specific Peripherals. In addition to the vendor-specific peripherals, there are also many other platform-specific peripherals such as Instrumentation Trace Macrocell (ITM) and Data Watchpoint and Trace (DWT). ITM supports printf style debugging tools to analyze its operating system. According to the ARM Cortex-M4 technical reference manual [2], these peripherals locate in a region ranging from `0xe0000000` to `0xe0100000`, which work similarly to the vendor-specific peripherals mentioned before.

Peripheral internals. To illustrate how peripherals work and how they communicate with each other, we present a high-level architecture and the peripheral internals in Figure 1. Specifically, a peripheral has a peripheral core to execute its own task and

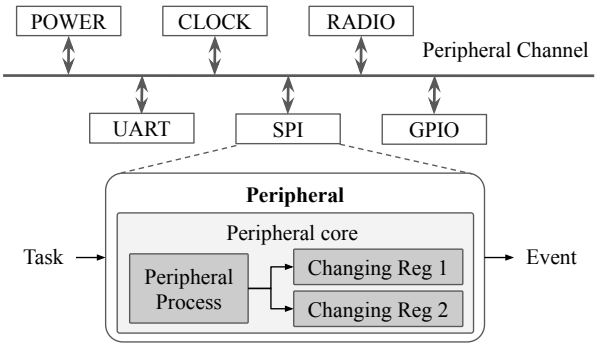


Figure 1: Peripheral channel interconnected between peripherals and peripheral's internal process

Framework	Main Technique			Supportive Technique					Termination criteria		
	Inference	Symbolic Execution	Replacement	Pattern Based	Path Selection	Invalidity Guided	Library Matching	Bit-usage Analysis	Return to Callee	Targeted Exploration	No Invalidity
P2IM [33]	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗
DICE [46]	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗
Laelaps [25]	✗	✓	✗	✗	✓	✗	✗	✗	✗	✗	✓
μEMU [60]	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗	✓
HALucinator [30]	✗	✗	✓	✗	✗	✗	✓	✗	✗	✗	✗
Jetset [37]	✗	✓	✗	✗	✓	✗	✗	✗	✗	✓	✓
Fuzzware [52]	✗	✓	✗	✗	✗	✗	✗	✓	✗	✗	✗

Table 1: Summary in SOTA MCU-based rehosting frameworks. Each column presents a technique involved.

update its internal registers, which is done independently without CPU intervention. Such a process is called an autonomous peripheral operation [16]. Meanwhile, as shown in Figure 1, each peripheral core receives and handles tasks from its own as well as other peripherals, and emits events for inter-peripheral communication. In particular, such inter-peripheral communication is called distributed programmable peripheral interconnect (DPPI) in the Nordic MCU [3]. Through this communication channel, peripherals can send and receive tasks and events independently. The channel configuration can be defined by the developers.

2.2 State-of-the-art MCU Firmware Rehosting Frameworks

As mentioned in §2.1, each peripheral plays an essential role in an embedded device. Unlike traditional devices running on general-purpose OS such as Linux, MCU-based embedded devices usually run on a proprietary tiny OS or without OS (*i.e.*, bare metal devices), and need to interact with a great number of peripherals. Therefore, to correctly emulate an MCU firmware, peripheral modeling is mandatory, which is essential to properly handle the accesses (*e.g.*, read and write) to the peripheral registers in the firmware code. However, this is fundamentally challenging due to the absence of hardware, and a rehosting framework must properly infer and assign a value to a peripheral register when it is accessed by the firmware code.

In the following, we briefly describe 7 state-of-the-art (SOTA) rehosting frameworks for MCU-based firmware. At the high-level, these frameworks share a common objective, as they attempt to model the MCU peripherals (*i.e.*, infer the value of a peripheral register) to enable hardware-independent emulation. To address the challenge of peripheral modeling, these frameworks propose different approaches, and we summarize them in Table 1, which presents their corresponding main techniques and supportive techniques, as well as the termination criteria. While there are other firmware rehosting frameworks such as FirmAE [39], they are designed for non-MCU-based firmware (*e.g.*, those running on general-purpose OSes). They attempt to tackle the peripheral modeling challenge by using OS-specific functions (*e.g.*, `syscall`) and system logs to handle peripheral accesses, which are unavailable in MCU-based firmware. Hence, these non-MCU-based firmware rehosting frameworks fall out of our scope. In the following, we describe each framework in more detail.

P2IM. The key idea of P2IM [33] is to infer the values of peripheral registers based on their unique usage patterns. Specifically, it classifies peripheral registers into four types: 1) Control register, 2) Status register, 3) Control-status register, and 4) Data register. For example, a control register follows a read-modify-write pattern. Based on the classification, P2IM handles each type of peripheral register accordingly. In particular, for a status register, P2IM performs explorative executions to test different bits on the status register and chooses the one that can execute further branches. If the execution correctly returns to the caller with the assigned status register value, P2IM terminates the explorative execution. For a control register, P2IM uses directly its previous value.

DICE. In terms of peripheral register modeling, DICE [46] uses the same strategy as P2IM. Additionally, DICE implements a process related to Direct Memory Access (DMA) process on top of P2IM.

Laelaps. As shown in Table 1, Laelaps [25] uses symbolic execution with path selection. To be more specific, it performs symbolic execution on every peripheral register access point. For efficient path selection, Laelaps defines standards, such as avoiding infinite loops and prioritizing new paths, and uses the Context Preserving Scanning Algorithm (CPSA) to find the most promising path. Based on the selected path, Laelaps carries out symbolic execution to infer the value. To mitigate path explosions, Laelaps only considers a few next basic blocks during symbolic execution.

μEMU. Compared to P2IM and Laelaps, μEMU [60] performs an invalidity-guided symbolic execution as presented in Table 1. Specifically, μEMU defines different types of invalidity, such as infinite loops and invalid memory access. Moreover, μEMU uses different caching strategies, such as storage model and replay-based matching, based on the characteristics of peripheral register usage in firmware. For example, the storage model is similar to the way of handling a control register in P2IM. When the current caching strategy does not work, μEMU uses other strategies.

Next, it performs symbolic execution on a peripheral register access point (*i.e.* a peripheral register read) to execute a new branch depending on the selected caching strategy. If it does not cause invalidity, it saves the result of symbolic execution and continues to emulate the firmware. Otherwise, it executes different branches by applying other values from symbolic execution.

HALucinator. Unlike the aforementioned frameworks, HALucinator [30] replaces Hardware Abstract Layer (HAL) functions with their implementation since access to a peripheral register normally occurs in HAL functions, which are software libraries to handle hardware operations. Therefore, HALucinator first scans and identifies the HAL library functions from a firmware image through the library matching algorithm. Next, it replaces all HAL library functions with self-programmed functions.

Jetset. Jetset [37] uses symbolic execution with path selection until it reaches a specified destination address. First, Jetset constructs a control flow graph (CFG) from an entry point to the destination address. During emulation, it adds symbols to peripheral registers read by the firmware. Since there could be multiple paths to the destination address, Jetset uses the Tabu search [34] algorithm and context-sensitive distance to choose an optimal path. When the path is determined, Jetset performs symbolic execution to resolve all constraints on the path and infers peripheral behavior.

Fuzzware. Fuzzware [52] uses locally-scoped dynamic symbolic execution to infer peripheral values that are meaningful to the firmware logic to build access models of a peripheral register. However, compared with other frameworks, it also attempts to decrease the overhead of input as much as possible by analyzing bit-usage. Therefore, it utilizes the access modeling approach and efficiently performs fuzzing tests. Since it does not prioritize specific types of program paths (*e.g.*, removing paths that are not of interest), it can execute every reachable basic block based on the firmware logic.

Summary. While the techniques utilized in these frameworks are indeed useful, there are two fundamental limitations. First, they rely on heuristics and empirical observations to infer the peripheral register values (*e.g.*, the pattern-based inference implemented in P2IM), which may assign incorrect values to peripheral registers. Second, many approaches appear to be biased and may not faithfully replicate the actual firmware execution traces, such as the invalidity-guided approach demonstrated by μEMU. Consequently, these frameworks may execute program branches that should never be executed in real hardware (*i.e.*, false positives of executed basic blocks).

3 THE HIDDEN MEMORY MAPPING

In this section, we first describe cases that two SOTA rehosting frameworks, P2IM and μEMU, model a peripheral register incorrectly (§3.1). Then, we describe and illustrate the root cause of incorrect modeling, and hidden memory mapping among peripheral registers (§3.2).

3.1 The Failure Cases from Existing Rehosting Frameworks

In §2.2, we describe how each existing rehosting framework actually works and its limitations. To more concretely explain how these frameworks produce incorrect results, we present two real-world firmware examples where P2IM and μEMU fail to correctly model a peripheral register, respectively in Figure 2 and Figure 3. The fundamental reason is that without the corresponding hardware, P2IM and μEMU will lead to execution on incorrect program branches that should never be executed in real hardware, as they

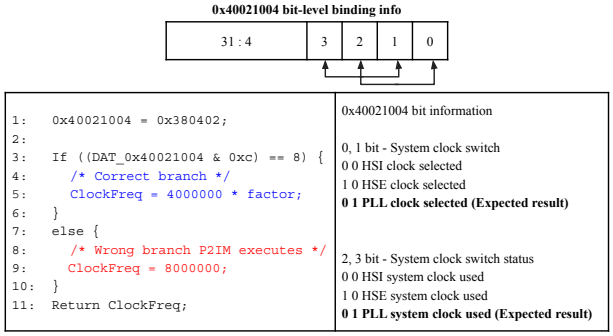


Figure 2: Simplified Real-world firmware code snippet where P2IM models a peripheral register in a wrong way.

fail to reflect an actual change of certain peripheral register. In the following, we describe these cases in detail.

P2IM. Figure 2 shows the source code snippet in a real-world firmware evaluated by P2IM, which updates a `ClockFreq` variable through a peripheral register located at `0x40021004` in the memory. Specifically, line 1 allocates `0x380402` to the register located in `0x40021004`. Next, it performs an `and` operation on the peripheral register value and a constant value `0xc`. Depending on the result, the program sets the clock frequency differently. When firmware runs on the corresponding hardware, the if condition is satisfied as the resulted value is 8 after the `and` operation, which further updates `ClockFreq` as `4000000` times `factor`.

However, to our surprise, we observe that P2IM emulates the firmware into the wrong branch for this piece of code (*i.e.*, the `else` branch starting at line 7), and we further explain it in detail. Recall in §2.2, P2IM determines the value of a peripheral register based on its usage pattern. As in the example, P2IM classifies the peripheral register located at `0x40021004` as a control-status register. Hence, it directly uses `0x380402` as a value of the peripheral register, which causes the program to execute the incorrect branch (*i.e.*, lines 7-9) since the `if` condition is not satisfied.

Through our observation, the reason for this incorrect emulation is that in the real hardware the value of the register at `0x40021004` becomes `0x38040c` after being assigned a value of `0x380402` at line 1, which is due to an automatic update of the value by the hardware (not visible in the code). Such an automatic allocation occurs due to the property of each bit in the peripheral described in Figure 2. As P2IM directly uses the assigned value at line 1 (`0x380402`) since the value does not violate the P2IM rules, it fails to set the register to the desired value, causing the program to execute the wrong branch.

μEMU. Similarly, Figure 3 presents the code snippet of a real-world firmware from μEMU's dataset, and the bit-level information of a register at `0x40023800`. At line 1, the program stores `0x1000000` to the register located at `0x40023800`. Next, it performs an `and` operation on the register and a constant value of `0x2000000`. It returns a different value depending on the result. As in the actual hardware, the program executes the `else` branch and returns `HAL_OK`.

However, similar to P2IM, μEMU emulates the firmware on the wrong branch (*i.e.*, line 4). When the program reads the peripheral register at `0x40023800`, μEMU performs symbolic execution to

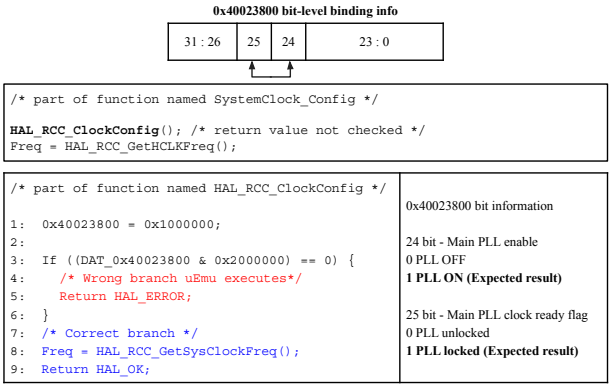


Figure 3: Simplified Real-world firmware code snippet where μEMU models a peripheral register in a wrong way.

infer the value. μEMU assigns `0x0` to the peripheral register and executes the incorrect `if` branch. Recall in §2.2 that μEMU uses an invalidity-guided approach for value inference. As the assigned value `0x0` does not cause an invalid state (because the caller function `SystemClock_Config` does not check the return value from its callee `HAL_RCC_ClockConfig`), μEMU directly assigns value and thus fails to execute expected basic blocks.

Similar to the example in Figure 2, the reason for this incorrect modeling is that when a value `0x1000000` is assigned to the register at `0x40023800` (*i.e.*, line 1), the value becomes `0x3000000` due to an automatic update of the hardware. The reason is the property of each bit in the register described in Figure 3.

3.2 Understanding the Hidden Memory Mapping

As described above, the root cause of the two failure cases is the hidden value update automatically performed by the MCU, which occurs among the peripheral register bits that are semantically related to each other. For instance, from Figure 2, the 0 and 1st bits of the register at `0x40021004` are related to the 2nd and 3rd bits, respectively. In other words, the change of a peripheral register can alter one or multiple peripheral registers (including the changed register itself) automatically *without CPU intervention*, and we define such a hardware property as *hidden memory mapping*. Through our observation of the firmware execution, we find that hidden memory mappings can be classified into two types: (1) intra-peripheral memory mapping (*i.e.*, all the mapped registers are within the same peripheral such as the one in Figure 2 and Figure 3) and (2) inter-peripheral memory mapping (*i.e.*, the mapped registers reside in other peripherals).

Rationale of Memory Mapping. Hidden memory mapping exists because of the special hardware design of embedded device peripherals. Recall in §2.1, peripherals are interconnected through a peripheral channel (*e.g.*, the PPIBus of Nordic SoCs [3]), which allows them to transmit tasks and events to one another without the intervention of CPUs. As a result, developers can make use of such a feature to improve throughput and lower latency to save the energy of devices. Moreover, many peripheral registers have semantic-level relationships so their values should be updated simultaneously. For instance, an `ENABLE` register and a `DISABLE` register within a peripheral have exactly opposite semantics, and thus when one of their values gets changed, the other should be updated simultaneously as well through the memory mapping process [3].

4 OVERVIEW OF AUTOMAP

4.1 Objective

As illustrated in §3.1, existing firmware rehosting frameworks have incorrect peripheral modeling results due to hidden memory mappings. Therefore, in this paper, we present the first framework, namely AutoMap, which can be an extension of any of the existing emulators (*e.g.*, P2IM, and μEMU) to automatically reveal hidden memory mappings to support faithful firmware emulation. Specifically, given an MCU-based firmware with the corresponding hardware, AutoMap is able to extract the knowledge of hidden memory mapping of peripheral registers, which can be used by the emulators to correctly emulate the firmware. AutoMap provides a memory mapping knowledge base to store inferred mapping information, which can be directly applied to the emulators if applicable (*e.g.*, when emulating firmware from an MCU with existing mapping knowledge). Otherwise, the values of peripheral registers need to be inferred for emulation.

4.2 Challenges and Solutions

In the following, we detail the challenges we encountered when designing AutoMap, and the corresponding solutions.

Challenge(C) 1. Reliably Triggering Memory Mapping. As described in §3.2, hidden memory mapping is not visible from the firmware code. As a result, using just the firmware code is impossible to infer the memory mappings, and thus there are only two ways. First, we can obtain a few memory mappings from the vendor's specifications (*e.g.*, [11, 20]). However, after exhaustive investigation, we find that the vendors only provide a very limited number of hidden memory mappings, which are far from complete and not sufficient for firmware emulation. Alternatively, the other way is to use the corresponding MCU hardware to obtain the memory mappings. However, as hidden memory mappings are highly diversified due to different hardware implementations, we need a systematic solution to trigger the memory mapping.

Solution(S) 1. Differential Hardware Memory Introspection. To address the challenge, we design a hardware-in-the-loop approach to extract memory mapping knowledge by using differential hardware memory introspection. Specifically, AutoMap requires the corresponding hardware MCU to reliably trigger hidden memory mappings, in which it consecutively introspects the hardware memory before and after writing a specific peripheral register. Therefore, the changes in the peripheral memory region reflect the corresponding altered registers and their values, which are used to construct the hidden memory mapping relationship.

C2. Handling Mutual Dependencies to Trigger Memory Mappings. Although with S1 we can trigger hidden memory mappings, we still need to handle sophisticated dependencies among peripheral registers. To be more specific, we discover that a few mappings will be enabled only when another specific peripheral register is

set to be the desired value. For instance, to trigger the memory mapping properly on the TASKS_HFCLKSTOP register, we must set another register TASKS_HFCLKSTART to a proper value in advance. Without satisfying the dependency, the hidden memory mapping on TASKS_HFCLKSTOP will not be triggered, which further leads to incorrect or incomplete results. As a result, we also need to satisfy their dependencies in advance by assigning proper values to the dependent registers. However, there are neither vendor specifications nor publicly available information to infer them. Meanwhile, a brute-force approach (*i.e.*, traversing all possible registers in the memory) is unrealistic due to the huge size of the peripheral region (*e.g.*, the peripheral region size is 0x20000000 in Nordic NRF52832 [10]), which makes it extremely difficult to locate the exact dependent registers and the desired values for assignment.

S2. Memory Context Preparation. Interestingly, we discover that these dependencies are naturally handled by the firmware code logic, which can be leveraged to resolve the dependencies to trigger the memory mappings as needed. Specifically, for a specific peripheral register write to be executed in the firmware, its dependent registers should have already been initialized by the previously executed memory write instructions. Therefore, to resolve the dependencies of a peripheral register, we replicate all the executed peripheral register writes before it, which represents a specific memory context.

C3. Efficiently Extracting Memory Mapping. The third challenge we need to address is to efficiently extract hidden memory mappings, which is challenging for two reasons. First, as mentioned in C2, the peripheral memory space is extremely large, and thus it is time-consuming to traverse the whole space to get all hidden memory mappings. Second, as illustrated in §3.1, hidden memory mapping has bit-level granularity as each bit of a peripheral register may have mappings with the bits of other peripheral registers. More specifically, since the size of a peripheral register is 4 bytes, there are 2^{32} cases that can be written to the peripheral register. Therefore, we must efficiently extract the hidden memory mappings. Although we can use the memory layout from the vendor's specifications [10] to narrow down the search space, this is not sufficient and we still need a more efficient solution.

S3. On-demand Memory Mapping Inference. We use an on-demand memory mapping inference approach to address this challenge, as we find that not all hidden memory mappings are necessary for firmware emulation. Therefore, to increase the efficiency, AutoMap only infers the hidden memory mapping only when it is needed. For example, when AutoMap encounters a specific peripheral register write, it only infers the hidden memory mapping triggered by this specific write, and uses the aforementioned differential hardware memory introspection to extract the hidden memory mappings. Additionally, to minimize hardware query operations, we use a knowledge base to store the extracted memory mapping for future use.

4.3 Framework Overview

As mentioned before, AutoMap can be an extension of any SOTA firmware emulation framework, and we present the high-level architecture of AutoMap in Figure 4. To integrate AutoMap into an emulator, we need to first modify the emulator for its peripheral modeling logic, and also develop the hardware interaction components for hidden memory mapping extraction. In the following, we briefly explain the necessary emulation modification and AutoMap's components.

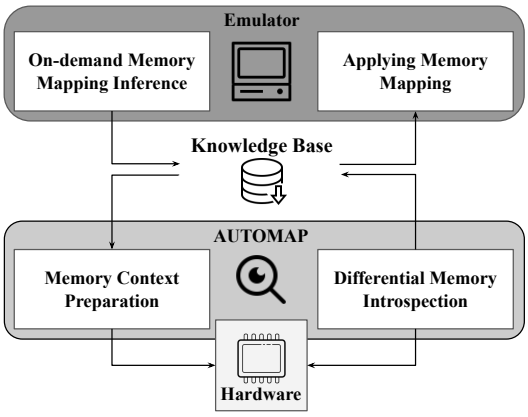


Figure 4: Overview of AutoMap.

Emulator Modification. We modify the logic of reading and writing a peripheral register from an emulator. Whenever reading an uninitialized peripheral register takes place, the emulator executes AutoMap to extract the initial value. In addition, whenever writing a value to a peripheral register occurs, the emulator checks the knowledge base. If the knowledge of the memory mapping of this write operation already exists in the knowledge base, the memory mapping is applied to update other affected registers. Otherwise, the emulator halts and invokes AutoMap to extract the hidden memory mapping. After the peripheral modeling is finished, the emulator continues to emulate firmware until new peripheral access occurs.

AutoMap. There are two components of AutoMap. First, to prepare the memory context before extracting hidden memory mappings, it needs to replicate each peripheral register written in the history, which is recorded during emulation. Second, with a given peripheral register and a corresponding value to be written, AutoMap performs differential hardware memory introspection to extract the hidden memory mappings, and the results will be stored in a knowledge base.

4.4 Scope and Assumptions

As AutoMap uses a hardware-in-the-loop approach, we assume that the corresponding hardware (*e.g.*, a Nordic NRF52832 SoC) is provided for the given firmware. In addition, the hardware should have an exposed debugging service (*e.g.*, openocd or J link, or st link to connect to GDB) that allows introspection of the device memory so that AutoMap can interact with it. For a proof-of-concept, in this work, we target three specific MCUs, Nordic NRF52832 [8], STMicroelectronics STM32F103 [19] and STM32F429 [21], which have interfaces such as J-link [15] and st-link [18] for hardware access. For every other MCUs evaluated by μEMU, they support J-TAG debug [4, 5, 14] or on-chip debug [13]. We also assume the

memory layout of the hardware is known, since they can be easily found from the vendor specifications (*e.g.*, [10]).

5 DETAILED DESIGN

In this section, we present the detailed design of AutoMap, including the necessary emulator modification in §5.1 and AutoMap's unique components in §5.2.

5.1 Emulator Modification

On-demand Memory Mapping Inference. Recall in §4.2, AutoMap uses an on-demand approach to reveal hidden memory mapping only when it is necessary for emulation. At a high-level, this approach consists of three steps. First, the emulator handles the peripheral register access (*e.g.*, read and write) during firmware emulation. Next, the emulator invokes AutoMap to query the memory mapping knowledge from a common knowledge base shared among firmware of the same MCU. Finally, when the memory mapping is returned from AutoMap, the emulator needs to further prepare the memory context to handle mutual dependencies. In the following, we detail each of these steps.

- **Peripheral Register Access Handling**. Before the emulator starts executing the firmware, it first initializes all the previously modeled registers from the knowledge base. When the initialization is finished, the emulator starts executing firmware. During the emulation, it needs to handle the access to peripheral registers within the peripheral regions (defined by the vendor specifications [10]). In summary, there are two types of peripheral register operations during the execution: 1) Reading a peripheral register, 2) Writing a value to a peripheral register. Reading a peripheral register (*e.g.*, `ldr`-related instructions) is to fetch the value from the peripheral register's address and store it into a register (*e.g.*, R0, R1 in ARM assembly code). In contrast, writing a peripheral register (*e.g.*, `str`-related instructions) involves both the address of a peripheral register and a value written to the register. When the emulator handles peripheral register accesses, it queries the knowledge base.
- **Memory Mapping Knowledge Base Query**. When the emulator encounters a read operation for an uninitialized peripheral register, it executes AutoMap with the address of the target register to extract an initial value from the hardware, as a peripheral register may be assigned a non-zero initial value by default (we will describe later how AutoMap can get the initial value in §5.2). An uninitialized peripheral register (with a zero value) means that the register may have a default zero value, or it has not been modeled by AutoMap yet. In this case, it is not necessary to query the knowledge base. Next, the obtained initial value is assigned to the register in the emulator.

  In contrast to a read operation, whenever the emulator encounters a peripheral register write, it first queries the knowledge base where the hidden memory mapping knowledge for the peripheral register may exist. Hence, it queries the knowledge base for this specific write operation, based on the peripheral register address and the value to be written. Note that the hidden memory mapping is hardware-specific and all firmware running on the same MCU share the same set of peripherals. Therefore, a specific peripheral register write always results in the same outcome for different firmware running on the same MCU (*i.e.*, the hidden memory mappings are identical). Given a set of the peripheral register and value, if the memory mapping can be found from the knowledge base, the emulator directly applies the mapping to update the corresponding affected registers. Otherwise, it further invokes AutoMap to extract the hidden memory mapping for this register write. As a result, the above memory mapping knowledge base query can be formulated as:

$$R : Q(A,T) = AutoMap(A),\ \forall V\ :\ KB(A,V,T) = \emptyset$$

$$W : Q(A,V,T) = \begin{cases} KB(A,V,T), & KB(A,V,T) \neq \emptyset \\ AutoMap(A,V), & KB(A,V,T) = \emptyset \end{cases}$$

  where A, V, T, and Q respectively denote the address of a peripheral register, the value to be written to A, the MCU model, and query function. In addition, R and W denote different queries for peripheral register read and write, as a peripheral register read only requires to query the value of the register while a register write requires to query the memory mapping when the register A is written by V.
- **Memory Context Collection**. As mentioned in §4.2, AutoMap uses memory context to resolve the complicated dependencies among peripheral registers to trigger memory mappings. Therefore, before invoking AutoMap to reveal the hidden memory mapping, the emulator needs to collect the memory context which will be used later in AutoMap. To collect memory context of the target register, the emulator saves every peripheral register write operation during the operation. Therefore, when a peripheral register write occurs, the emulator stores the address of a peripheral register and the corresponding value including repeated writing as well.

5.2 AutoMap

In this section, we describe the components of AutoMap in detail. As hidden memory mapping cannot be inferred from the software's perspective, the objective of AutoMap is to interact with the corresponding hardware to extract the hidden memory mapping to model the peripheral for emulation. At a high level, AutoMap receives a peripheral register and a value written to the peripheral register from the emulator when the peripheral register write operation occurs. It first prepares the memory context for mutual peripheral dependencies, and then uses differential hardware memory introspection to extract the hidden memory mapping. Additionally, when the emulator loads a value from an uninitialized peripheral register, it needs to get its initial value from the hardware since the initial value may not be zero and can affect the program flow. As described in Algorithm 1, there are three functions. The function BUILD_MEMMAP (line 15) is called by the emulator and it executes two other functions, PREPARE_DEPENDENCY and MEM_INTROSPECTION (lines 17 and 24). Specifically, BUILD_MEMMAP describes how AutoMap models a peripheral register, which takes the history of register writings (`prev_writes`), a peripheral register address (`peri_addr`), and the value written to the peripheral register (`value`) as input, and outputs the memory mapping knowledge for this particular peripheral register. In the following, we further describe this algorithm in detail.

Algorithm 1 Modeling memory mapping

```
1:  Input: prev_writes: history of register writes; peri_addr: Address of peripheral
    register; value: A value written to the peripheral register
2:  Output: knowledge: memory mapping knowledge

3:  procedure PREPARE_DEPENDENCY(prev_writes)
4:      peri_reg_addrs, values ← prev_writes
5:      idx ← 0
6:      while idx < LEN(peri_reg_addrs) do
7:          MEM_WRITE(values[idx], peri_reg_addrs[idx])
8:          idx++
9:          execute CPU cycles (# of instructions)

10: procedure MEM_INTROSPECTION(prev_mem, next_mem)
11:     idx ← 0
12:     while idx < LEN(prev_mem) do
13:         if prev_mem[idx : idx + 4] != next_mem[idx : idx + 4] then
14:             knowledge ← compared reg addr & next_mem[idx]
            idx = idx + 4

15: procedure BUILD_MEMMAP(peri_addr, value, prev_writes)
16:     knowledge ← MEM_READ(peri_addr)
17:     PREPARE_DEPENDENCY(prev_writes)
18:     prev_mem ← MEM_READ(peri_regions)
19:     MEM_WRITE(value, peri_addr)
20:     execute CPU cycles (# of instructions)
21:     next_mem ← MEM_READ(peri_regions)
22:     knowledge ← MEM_READ(peri_addr)
23:     if prev_mem != next_mem then
24:         MEM_INTROSPECTION(prev_mem, next_mem)
```

Memory Context Preparation. Since a given set of a peripheral register and a value can have a dependency upon other registers, AutoMap prepares the memory context to handle the dependency of the peripheral register beforehand, which is to set other dependent registers with the corresponding value in advance. The function PREPARE_DEPENDENCY shows how the memory context is prepared. First, AutoMap gets every previous write from a memory context database (line 4) which is established by the emulator as in §5.1. Specifically, a memory context record consists of two pieces of information: 1) the peripheral register's address, 2) the corresponding value written to the register. Based on the information, AutoMap replicates every write in order (line 7). Meanwhile, after each write, it executes a pre-defined number of instructions (line 9) because some writes need a few cycles to finish their process, which can vary depending on MCU types. For example, from Nordic documentation [7], it needs several CPU cycles to write a value to the LATCH register from the GPIO peripheral. Hence, we execute enough instructions after a write to finish the process of the peripheral core.

We use a concrete example in Figure 5a to show how this component works. First, the program writes value 1 to two peripheral registers, `TASKS_HFCLKSTART` and `TASKS_HFCLKSTOP`, in order. In line 1, there is no memory context as the program has not started writing values to any peripheral register. However, in line 2, AutoMap needs to prepare the memory context which involves the peripheral register write at line 1. To show the importance of memory context, we further use two examples in Figure 5b and Figure 5c which present the different memory mapping results without and with preparing the dependency from Figure 5a, respectively. Two figures present that the first knowledge (*i.e.*, write a value 1 to the register located at `0x40000000`) is properly modeled. However, without memory context preparation (*i.e.*, without writing 1 to `0x40000000` before modeling), AutoMap directly models the next write in the reset state of hardware. That is, the value of the register located at `0x40000408` is 0. As a result, the second knowledge (*i.e.*, changes `0x40000408` to 0) misses memory mappings since the value is not set as 1 before modeling.

Differential Hardware Memory Introspection. After memory context preparation, AutoMap performs memory differential introspection on hardware to extract hidden memory mapping. As shown in Algorithm 1, AutoMap consecutively reads peripheral regions two times: before and after a peripheral register write (lines 18 and 21). Before comparing two memory regions, AutoMap reads the changed value of the target register since the register may not change to the exact written value as shown in Figure 2 (line 22). Then, it compares two memory regions. If the two resulting memory states are not identical, indicating other peripherals are simultaneously updated, AutoMap further invokes the MEM_INTROSPECTION procedure. Otherwise, the register write does not trigger memory mapping. Eventually, a hidden memory mapping record will be stored in the knowledge base, and we use the following expressions to define a hidden memory mapping record in the knowledge base:

$$KB(A, V, T) = (V_{init}, V_{after}, M_0, M_1, ...)$$

where A is the address of the peripheral register, V refers to the written value, and T is the MCU model. Furthermore, the remaining elements are defined based on the following expressions:

$$V_{init} = mem_{init}(A)$$
$$V_{after} = mem_{after}(A)$$
$$M_n = (A_n, mem_{after}(A_n))$$

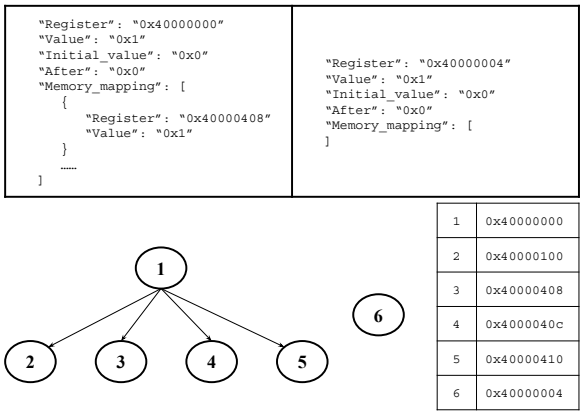
Specifically, V_{init} refers to the initial value of the peripheral register at address A, V_{after} is the value of the register at A after the register write, and M_n presents every memory mapping where n is an index starting from 0. mem_{init} and mem_{after} denote memory of hardware in an initial state (*i.e.*, memory consists of an initial value of peripheral registers) and after writing, respectively.

We further use the above symbols to explain in detail how MEM_INTROSPECTION works. As shown in line 10, the function takes two parameters, `prev_mem` and `next_mem`. Two variables include memory snapshot of whole peripheral regions before and after write, respectively. Because memory mapping can occur in different peripherals, it searches every peripheral region (line 12). If any change has been detected, it saves the corresponding register address and its changed value to the knowledge. M_n (*i.e.*, $(A_n, mem_{after}(A_n))$) is defined where A_n is the register address and the second parameter presents the changed value. The value of multiple registers can change so that the result might include multiple addresses and their changed value per a set of a peripheral register address and a given value (line 14). Hence, there are multiple mappings on the above formula, such as M_0 and M_1.

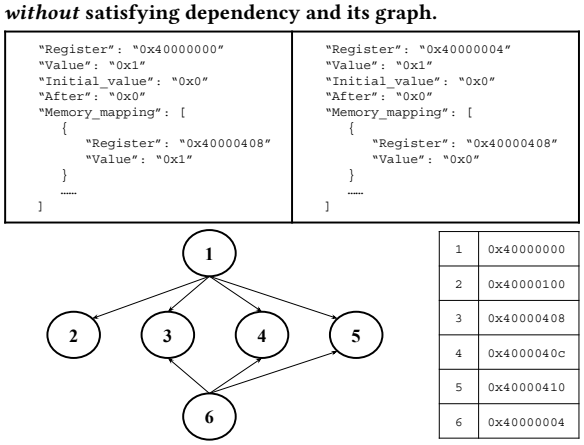
When AutoMap finishes, it outputs the hidden memory mapping knowledge of the given peripheral register (*i.e.*, A, V in the above formula). Memory mapping information can include none, one, or multiple addresses of registers with a changed value (*i.e.*, M_n). The new knowledge of memory mapping is added to the knowledge base when it does not exist in the base. The purpose of the knowledge base is to save the extracted hidden memory mapping to increase

```
1: NRF_CLOCK->TASKS_HFCLKSTART = 1;      /* 0x40000000 = 1 */
2: NRF_CLOCK->TASKS_HFCLKSTOP = 1;       /* 0x40000004 = 1 */
```

(a) An example code snippet presenting dependency with Nordic NRF52832.



(b) Knowledge of mappings from the above code snippet *without* satisfying dependency and its graph.



(c) Knowledge of mappings from the above code snippet *with* satisfying dependency and its graph.

Figure 5: An example of the clock peripheral in an NRF52832 MCU firmware with mutual register dependencies.

efficiency. As the hidden memory mapping is MCU-specific (*i.e.*, T), all the firmware running on the same MCU can share the knowledge base during the emulation, which saves a huge number of times for hardware memory introspection.

6 IMPLEMENTATION

In this section, we describe how we modify an emulator and implement AutoMap described in §5.

Unicorn Emulator Extension. We developed AutoMap atop the Unicorn [48] emulator. To begin with, the emulator initializes every peripheral register modeled in the knowledge base. Since Unicorn supports specific instruction-level hooking, we can hook every `ldr`- and `str`-related instruction to handle peripheral register access. When a `ldr`-related instruction (*e.g.*, `ldr` and `ldrb` instructions) loads a value from an uninitialized peripheral register, it executes AutoMap to extract the initial value.

When any `str`-related instruction (*e.g.*, `str` and `strb` instructions) is encountered, the emulator checks the address of the destination. If it is located in a peripheral region, it checks the knowledge base for memory mapping knowledge. When the corresponding knowledge exists, it directly uses the mapping to change one or multiple peripheral registers. However, when the peripheral register has not been modeled, the emulator invokes AutoMap to extract the hidden memory mapping.

AUTOMAP. We use Python to implement AutoMap for modeling memory mapping of peripheral registers, and use GDB to store a value and read memory regions of peripherals. There are many tools that can be used to connect GDB with MCUs, such as J-TAG, openocd [36], and J-link [15]. Since we evaluate AutoMap on MCU NRF52832 from Nordic Semiconductors, we use J-Link to connect the GDB server. In addition, we use openocd [36] to connect the GDB server on STM32F103 and STM32F426 MCUs through st-link [18].

7 EVALUATION

In this section, we present our evaluation of AutoMap[1] answering four research questions: (1) How many hidden memory mappings can AutoMap reveal? (2) How can the knowledge base method optimize the execution time of AutoMap? (3) How can we interpret the extracted hidden memory mapping results? (4) How can AutoMap improve the emulation performance of existing rehosting frameworks? First, we present the detailed result of the extracted hidden memory mapping by AutoMap in §7.1. Second, we evaluate how the knowledge base method can improve efficiency in §7.2. Third, we classify unique types of memory mappings based on their patterns in §7.3. Lastly, we demonstrate a use case by integrating AutoMap into μEMU for fuzz testing in §7.4.

Firmware Dataset. Table 2 shows the 50 firmware used for evaluation on AutoMap, which come from three different MCUs of two MCU vendors. Among these 50 firmware, 41 of them are the example firmware included in the Nordic NRF52832 SDK of version 17.0.2. The main objective of using the example firmware is that it utilizes various peripherals. Therefore, we can make a fair evaluation of AutoMap. For the remaining firmware, 5 and 4 of them are real-world firmware evaluated by P2IM and μEMU, which are developed for STM32F103 and STM32F429 MCUs, respectively. Note that the P2IM and μEMU datasets also have firmware from other MCUs but we exclude them as we only focus on two MCUs, namely STM32F103 and STM32F429.

7.1 Detailed Results of AutoMap on Each Firmware

In this section, we present the detailed results of hidden memory mapping extracted from the 50 firmware, which are shown in Table 2. To obtain statistics, the emulator executes firmware without any interrupt. That is, input is not required for the experiment. We also show the detailed statistics reported from AutoMap, including the number of unique writing to a peripheral register, the

[1]The source code is available at https://github.com/OSUSecLab/AutoMap.

MCU	Firmware	# RW		# RW	T#	M#	# V	# MM			
		P	V	→MM	MM	/RW	peris	1-1	1-n	Intra	Inter
NRF52832	bk_freertos	12	17	21	52	7	3	6	15	11	2
	bk	1	9	9	26	3	1	0	9	8	0
	bk_rtc	12	17	21	52	7	3	6	15	11	2
	bk_systick	3	9	9	26	3	1	0	9	8	0
	bsp	7	50	35	95	11	5	5	30	24	4
	cli_libuarte	15	68	30	73	11	7	9	21	16	4
	csense_drv	12	61	37	97	11	5	9	28	24	4
	csense	12	44	24	51	7	5	9	15	13	2
	fatfs	3	45	23	81	11	3	2	21	17	3
	flash_fstorage	5	4	4	11	7	1	2	2	1	1
	flashwrite	10	38	26	67	11	5	9	17	11	8
	gfx	3	16	8	27	11	2	2	6	3	2
	gpiote	5	22	9	15	3	4	4	5	4	1
	i2s	5	49	22	59	6	3	4	18	15	2
	led_softblink	8	19	17	48	7	3	4	13	11	1
	libuarte	12	67	32	85	11	6	8	24	21	4
	power_pwm	8	22	19	48	7	3	4	15	13	1
	lpcomp	5	41	22	53	4	3	5	17	16	1
	change_int	3	10	7	13	3	2	2	5	4	0
	ppi	9	54	17	28	4	6	10	7	8	1
	preflash	15	55	35	78	7	6	10	25	22	2
	pwm_driver	13	82	41	115	14	6	9	32	26	6
	pwm_library	5	40	14	25	3	4	7	7	8	1
	pwr_mgmt	13	43	30	74	11	5	8	22	17	4
	qdec	7	40	18	35	4	4	7	11	10	1
	radio_test	13	46	31	74	11	7	12	19	13	9
	ram_retention	1	11	10	30	4	2	0	10	8	1
	rng	5	21	14	35	11	3	6	8	5	4
	rtc	7	23	19	50	7	3	4	15	13	2
	simple_timer	3	17	12	31	3	2	2	10	9	0
	spi_mngr	14	55	38	86	7	6	9	29	24	3
	spi	5	45	24	72	11	3	4	20	17	2
	spis	5	43	20	53	5	4	4	16	13	3
	temperature	3	17	8	19	4	3	2	6	4	1
	timer	3	20	12	29	3	2	3	9	10	0
	twi_mngr	11	51	37	97	11	5	7	30	25	4
	twis_slave	13	68	36	95	11	6	8	28	23	5
	twi_scanner	3	16	10	29	11	2	4	6	3	4
	twi_sensor	3	16	9	28	11	2	3	6	3	3
	uart	3	30	20	62	11	2	2	18	14	3
	wdt	11	40	33	76	7	5	7	26	24	2
STM32F103	Drone	16	112	44	176	88	11	26	18	18	24
	Gateway	10	78	48	54	3	8	44	4	38	8
	Soldering_Iron	27	147	140	418	89	13	64	76	6	117
	Reflow_Oven	8	21	10	12	3	5	9	1	6	3
	Robot	8	64	14	18	3	7	12	2	10	3
STM32F429	CNC	9	222	121	721	256	12	68	53	11	108
	MODBUS	18	137	79	627	255	15	60	19	16	57
	PLC	10	43	7	7	1	5	7	0	3	2
	USB	10	24	7	262	255	6	5	2	0	7

Table 2: Detailed memory mapping related information of result on each firmware. RW, MM, P, V, T#, and M# stands for register write, memory mapping, Platform, Vendor, total number, and max number, respectively.

total number of memory mappings triggered by writings, and the number of memory mapping types including one-to-one and one-to-many mapping. Specifically, the third and fourth columns of Table 2 present the number of unique register writes to platform and vendor peripheral regions, respectively. It is shown that each firmware has at least one write to both a platform and vendor peripheral register. Moreover, the number of writes to a vendor's peripheral register is larger than a platform's peripheral register for all firmware. In addition, since the firmware from the NRF52832 MCU are example firmware, the number of writes is smaller compared to real-world firmware with STM32F103 and STM32F429 MCUs, as they are simpler. Moreover, the eighth column presents the number of vendor's peripheral used in each firmware, which shows that at least one peripheral is used in every firmware. Additionally, from the third and eighth columns, we can find out that multiple unique writes can occur in the same peripheral. For example, in the `blinky_systick` firmware, every write takes place in a single peripheral.

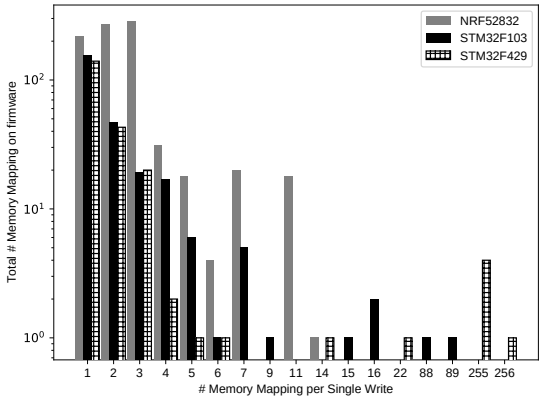


Figure 6: Total number of memory mappings triggered by a single register write from firmware.

The fifth column shows the number of register writes that trigger memory mapping. Therefore, the numbers in the fifth column are identical or smaller than the sum of the third and fourth columns. As shown, nearly more than 50% of the peripheral register writes can trigger hidden memory mapping. The sixth column presents the total number of memory mappings per firmware, which are triggered by the register writes in the fifth column. As the number of memory mapping is always greater than the number of register writes, it can be further inferred that many register writes can cause multiple memory mappings. To further show how one register can trigger multiple mappings, we show the maximum number of memory mappings triggered by a single register write in the seventh column.

As explained in the seventh column, since memory mappings can affect one or multiple registers, we can classify them into one-to-one and one-to-many memory mapping. The ninth and tenth columns of Table 2 present the number of the one-to-one and one-to-many memory mappings per firmware, respectively. Specifically, Figure 6 presents the number of memory mappings per single register write from every firmware. For example, there are 218 one-to-one memory mappings from the 41 example firmware. According to Figure 6, a single peripheral write can trigger up to 14 memory mappings, but in most cases it usually affects a single, two, or three registers in the NRF52832 MCU. Similarly, one-to-one, one-to-two, and one-to-three mappings also occur frequently in STM32F103 and STM32F429 MCUs. The numbers are close to example firmware even though the number of firmware evaluated is small as these real-world firmware have much more complicated logic. Moreover, the maximum number of mappings is larger than the example firmware (*i.e.*, 89 and 256 in STM32F103 and STM32F429, respectively).

Lastly, the last two columns present the number of intra- and inter-peripheral memory mappings, respectively. Intra-peripheral memory mapping means that it occurs only in the same peripheral region (*i.e.*, all the peripheral registers of this mapping reside in the same peripheral region). On the other hand, inter-peripheral memory mapping means that at least one peripheral register is in a different peripheral region. As shown in Table 2, even though the number of intra-peripheral mapping is more common than inter-peripheral mapping, we still need to search the whole peripheral

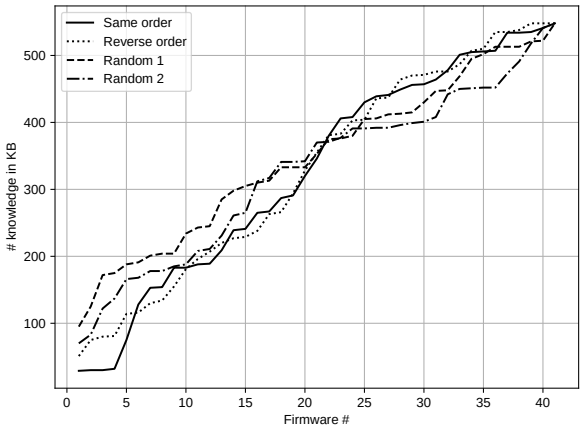


Figure 7: The Number of accumulated knowledge in example firmware. The firmware number in x-axis is based on the same order listed in Table 2 (so is the reverse order).

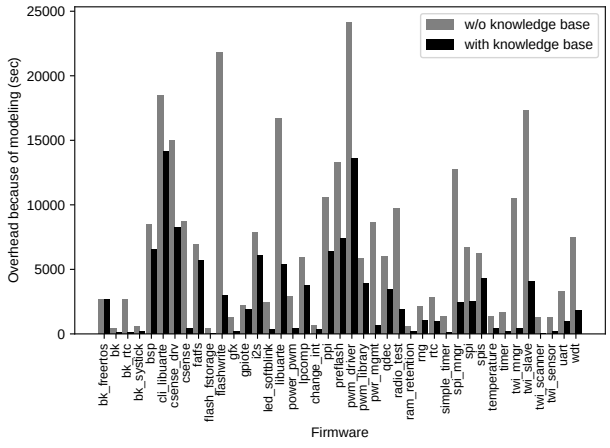


Figure 8: Memory mapping modeling execution time in each example firmware.

memory space to get the mapping knowledge in case we miss any inter-peripheral mappings.

7.2 Efficiency of Knowledge Base Method

As mentioned in the design of AutoMap, an important component is the knowledge base, which records the extracted memory mapping and can be shared among all the firmware running on the same MCU. In this section, we show how the knowledge base can improve AutoMap's efficiency. Since the knowledge base is only shared within the same MCU, we continuously apply AutoMap for the extraction of hidden memory mapping for the 41 Nordic example firmware for evaluation. Figure 7 shows the size of accumulated memory mapping knowledge, and it can be inferred that the size of knowledge increases as AutoMap models more firmware. Since the order of saving knowledge of memory mapping can also affect the growing speed of the knowledge base, we further re-run the experiment with four different firmware execution orders: 1) Identical order as Table 2, 2) Reverse order from Table 2, and 3) Two random orders.

Figure 7 shows that the size of the knowledge base increases rapidly when it is small. However, when the size of the knowledge base becomes larger, the knowledge base gradually converges and the growing speed becomes slower. Since the result comes from the 41 example firmware and each firmware usually utilizes different peripherals, it is difficult to see the efficiency of the knowledge base method clearly. However, considering that the total number of register writes from example firmware is 1,747, the knowledge base method has significantly reduced the number of hardware introspection from 1,747 to 548 (68.6%). If the knowledge base is absent, AutoMap has to interact with the hardware 1,199 times, which are not necessary and have been resolved in the previous firmware emulation.

To concretely show how the knowledge base can save the modeling time, we present Figure 8 to show the modeling time with and without knowledge base, where the 41 firmware were executed following the order in Table 2. As shown, the knowledge base method decreases modeling execution time significantly in all 41 firmware. Note that AutoMap spent a long time executing some of the firmware. For example, to model every peripheral register write in `flashwrite` firmware, it takes AutoMap 21,778 seconds to complete, where most of time spends on preparing memory context. In the experiment, we execute 2,000 instructions after each write to prepare memory context. The modeling time increases by 5 seconds whenever it executes 2,000 instructions for the context. However, Figure 8 clearly shows that knowledge base method can decrease the modeling time significantly.

7.3 Classifying Types of Memory Mappings

By analyzing every memory mapping explained in §7.1, we further interpret these results by classifying them into two types based on the mapped peripheral register values: 1) identical memory mapping, and 2) bit-banding memory mapping. Identical memory mapping includes multiple peripheral registers that have identical values, which normally occurs in the set and clear registers. With the set register, we can change the value by `or` operation. On the contrary, with the clear register, we can change the value by `xor` operation. Therefore, when the value of the set register is `0xffffffff`, we need to use the clear register to change the value of the set register. Because the operation of the set register is `or` operation so that we cannot change the value. Whenever we change one of the peripheral registers having a property of identical memory mapping, the peripheral core will update other identical peripheral registers automatically.

The second type is bit-banding memory mapping. The terminology bit-banding comes from a feature that is used in different architectures such as Cortex M3 and M4 [1, 2]. There is a peripheral register that each bit corresponds to the different memory region. Setting a bit enables the corresponding memory region directly accessible through a single `ldr` instruction. Unlike traditional bit-banding, with a peripheral register in MCU, each bit is corresponding to other peripheral registers. That is, when a bit of a peripheral register is set, the peripheral core changes the value

Firmware	# executed BBs		BBs portion of AutoMap **not in** μEMU	
	AutoMap	μEMU	#	%
Drone	1,413	1,410	5	0.35%
Gateway	1,385	1,248	216	15.59%
Steering_Iron	1,402	1,289	116	8.27%
Reflow_Oven	845	830	17	2.01%
Robot	1,035	964	77	7.43%

Table 3: Fuzzing result comparison between μEMU and both AutoMap and μEMU.

of the corresponding peripheral register automatically. In addition, setting a single bit can change one or multiple registers.

7.4 Integrating AutoMap to μEMU

Since AutoMap can guide SOTA rehosting frameworks to correct the execution of wrong branches, we demonstrate a use case of AutoMap by integrating it into μEMU. More specifically, we manipulate μEMU's logic to make use of AutoMap's results for peripheral modeling during emulation. During the peripheral modeling stage of μEMU, it utilizes the hidden memory mappings from AutoMap's knowledge base (constructed using the 50 firmware in Table 2) if available. Otherwise, it executes the original symbolic execution to infer the proper value of the targeted peripheral register.

To evaluate the performance, we test the new implementation on the 9 STM32-based firmware in Table 2 by performing 24-hour fuzz testing and collecting the basic block coverage. We exclude all the Nordic-based firmware because μEMU does not perform fuzz testing on NRF52832 firmware in its evaluation. Among the 9 tested firmware, only 5 of them work properly on μEMU while the rest 4 triggered segmentation faults possibly due to implementation issues. To be specific, when it starts fuzzing after modeling peripheral registers, memory error happens from S2E [29] framework. Hence, the authors are trying to address the issues, but they still exist when we perform fuzzing. Table 3 summarizes the fuzzing results. As shown, with AutoMap, μEMU is able to execute more basic blocks than μEMU on all the 5 firmware. AutoMap can improve the basic block coverage. Moreover, in the best case scenario (*i.e.*, the Gateway firmware), AutoMap covers the uncovered basic blocks of μEMU by 15.6%.

8 DISCUSSION

8.1 Characteristics of Peripheral Registers in Different MCU

Each MCU has various peripherals. For example, the Nordic NRF52832 MCU includes 40 different peripherals. In contrast, the STMicroelectronics STM32F429 MCU has nearly 70 different peripherals. In addition, the size of the peripheral region varies. For instance, every peripheral has the same size of memory region as 4,096 bytes in NRF52832. However, in STM32F429 and STM32F103, the size of each peripheral region is usually 1,024 bytes.

Due to various peripherals and their registers from different MCUs, the property of hidden memory mapping also differs. For instance, NRF52832 peripheral registers have hidden memory mapping properties with other registers such as an identical set of memory mapping. In addition, STM32F429 and STM32F103 peripheral registers have bit-level memory mapping on the peripheral register. For example, if one bit is set in a peripheral register, one or multiple bits of the register may change. Such a bit-level mapping also can be detected by our methodology. Because AutoMap always saves the changed value of the register to the knowledge base.

8.2 Limitation

To extract hidden memory mappings of a peripheral register, AutoMap requires the corresponding hardware MCU, which limits AutoMap's scalability. However, as mentioned in §4.2, using MCU hardware is the only way to get memory mapping, and such modeling is one-time effort as the established knowledge base can be shared among the firmware using the same MCU. Moreover, the hardware memory introspection in AutoMap causes additional overhead for peripheral modeling, as AutoMap needs to use tools such as GDB to read or write the hardware memory. We leave the improvement of AutoMap's efficiency to the future work.

In addition, we also encountered a special type of data peripheral register which is usually unwritable by the MCU hardware so that AutoMap could not reveal the memory mappings on these registers. Generally speaking, data peripheral registers are for receiving signals or data from the external environment, which are thus usually regarded as the source of some fuzzing tools such as μEMU. However, as AutoMap requires to use the hardware to write values to the peripheral registers for hardware memory introspection, it cannot handle these data peripheral registers. For these data registers, AutoMap can still fall back to existing rehosting approaches such as the symbolic execution in μEMU, to resolve the data register values.

8.3 Future Work

First, while in this work we implemented AutoMap for three different MCUs, it can certainly be extended for other MCUs as well. Although different MCUs can have different types of memory mappings, we can still apply AutoMap to model the peripherals using the same strategy, and construct a knowledge base for each MCU.

Second, in this work we show a few cases to demonstrate how the existing rehosting framework can produce incorrect results because of hidden memory mapping, and it is also worthwhile to revisit and improve these frameworks such as P2IM and μEMU. Particularly, as these works use fuzzing for vulnerability and bug discovery, AutoMap can be further integrated into the fuzzing components which may help discover additional vulnerabilities with fewer false positives when hidden memory mapping is considered.

Third, we can improve the algorithm of AutoMap. Since not all peripheral registers have memory mapping properties, those do not have memory mappings can be pre-filtered by AutoMap in advance so that it will not extract memory mappings for them, which can decrease the modeling time. For example, as explained in §8.2, a data register is usually unwritable so that we cannot trigger memory mapping on the data register. Hence, by filtering out the data register, we can increase the efficiency of AutoMap.

Finally, we can model a peripheral by using a machine learning when we get enough memory mapping knowledge on every peripheral. Such implementation may be feasible since memory mappings

may have some clear patterns and statistical features. The ultimate goal is to infer the memory mapping from the trained model without hardware, making AutoMap more scalable and efficient.

9 RELATED WORK

9.1 Firmware Analysis

Firmware analysis is widely adopted in the security field for a long time. Both static analysis and dynamic analysis are used in firmware analysis. For instance, FIE [31] uses symbolic execution to emulate firmware on top of KLEE [24], targeting the MSP430 micro-controller. FirmUSB [35] introduces USB-specific firmware analysis, and uses targeting algorithms and domain-specific knowledge. Karonte [50] discovered vulnerable interactions between different firmware by modeling and tracking multi-binary interactions. FirmXRay [57] uncovered Bluetooth link layer vulnerabilities from Bluetooth IoT firmware.

In terms of dynamic analysis, Avatar [58] is a hybrid emulator using hardware for input and output. IoTFuzzer [26] uncovered memory corruption vulnerabilities in IoT devices. Firm-AFL [59] applies augmented process emulation which combines system mode and user mode emulation to fuzzing on IoT firmware. P2IM [33] applies explorative execution to handle peripheral registers with their usage pattern. HALucinator [30] replaces HAL functions to decouple the hardware from the firmware. DICE [46] is an extension of P2IM detecting DMA. μEMU [60] performs symbolic execution to infer a value of a peripheral register. Similar to μEMU, Jetset [37] utilizes symbolic execution to infer the value of a peripheral register to reach a destination address. Fuzzware [52] uses dynamic symbolic execution and tries to minimize the overhead of input. Due to its objective, it covers every reachable basic block unlike other tools. Periscope [55] provides kernel peripheral fuzzing. DTaint [28] detects the vulnerability of firmware by using taint analysis.

Among all these related works, Avatar [58] is the closest one because both AutoMap and Avatar use a hardware-in-the-loop approach. However, there are still substantial differences: (1) Avatar does not attempt to reveal any hidden memory mapping, whereas AutoMap is exactly designed for this; (2) Avatar needs to run the identical firmware image on the target hardware in parallel during emulation. In contrast, AutoMap does not have this constraint as long as hidden memory mapping can be extracted on the identical MCU; (3) Avatar keeps the identical whole memory state between an emulator and hardware during execution. Therefore, Avatar cannot apply the knowledge base method since any firmware cannot have an identical memory state with others. However, AutoMap aims at the characteristic of MCU. Hence, it can be applied to different firmware with common knowledge.

9.2 Embedded Device Security

Many researchers [41, 42, 49] elaborate on the security of embedded devices and systems. Moreover, different types of attacks such as exploiting bugs of software, network, and exploiting side channels to embedded security have been proposed and discovered. Hall Spoofing [23] attacks the Hall sensor of an inverter by spoofing the signal. It results in nearly 31% of voltage change. Multiple attacks on Internet-of-Things (IoT) devices such as light-bulbs and power switches also exist [47, 54]. For example, Amazon Alexa is vulnerable since it sent private chats randomly [43]. Distributed denial-of-service (DDoS) attacks occur because 100,000 internet-connected devices were attacked [38]. Exploiting key distribution schemes also exist [22].

10 CONCLUSION

In this paper, we discover the hidden memory mapping on MCU-based firmware, which allows the change of a peripheral register to update one or multiple peripheral registers simultaneously without CPU intervention. Because of such hardware property, existing firmware rehosting frameworks such as P2IM and μEMU can execute incorrect program branches that should never be executed. To address this challenge, we present AutoMap, the first framework that automatically reveals hidden memory mapping of peripheral registers on demand using a differential hardware memory introspection approach. We implemented AutoMap atop Unicorn emulator and evaluate it with 41 example firmware from the Nordic NRF52832 SDK and 9 real-world firmware for STM32F103 and STM32F426 MCUs evaluated by P2IM and μEMU. Among these 50 firmware, AutoMap extracted 2,359 unique memory mappings (47 mappings per firmware on average) in total, and detected at least one memory mapping in every single firmware. Moreover, we show that the knowledge base method can significantly decrease the time up to 97.01% for modeling memory mappings in AutoMap. We also demonstrate a use case by integrating AutoMap into μEMU. With AutoMap, μEMU can achieve higher basic block coverage for fuzzing and covers previously unexplored basic blocks (up to 15.6%) on all the 5 tested firmware than μEMU alone.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their insightful comments. This research was supported in part by DARPA Award N6600120C4020, NSF Awards 2112471 and 2118491, and ONR Award N00014-17-1-2995. Any opinions, findings, and conclusions or recommendations in this paper are those of the authors and do not necessarily reflect the views of the DARPA, NSF, or ONR.

REFERENCES

[1] Cortex-m3 technical reference manual. https://developer.arm.com/documentation/ddi0337/h/.
[2] Cortex-m4 technical reference manual.
[3] Dppi - distributed programmable peripheral interconnect. https://infocenter.nordicsemi.com/index.jsp?topic=%2Fps_nrf9160%2Fdppi.html.
[4] Kinetis k64f sub-family data sheet. https://www.nxp.com/docs/en/data-sheet/K64P144M120SF5.pdf.
[5] Max32600 data sheet. https://datasheets.maximintegrated.com/en/ds/MAX32600.pdf.
[6] Nordic clock peripheral. https://https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.nrf52832.ps.v1.1%2Fclock.html&cp=4_2_0_18&anchor=frontpage_clock.
[7] Nordic nrf52811 engineering a errata 173. https://infocenter.nordicsemi.com/index.jsp?topic=%2Ferrata_nRF52811_EngA%2FERR%2FnRF52811%2FEngineeringA%2Flatest%2Fanomaly_811_173.html&resultof=%22%43%50%55%22%20%22%63%70%75%22%20%22%63%79%63%6c%65%22%20%22%63%79%63%6c%22%20.
[8] Nordic nrf52832. https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52832.
[9] Nordic nrf52832 gpio peripheral documentation. https://https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.nrf52832.ps.v1.1%2Fgpio.html&cp=4_2_0_19&anchor=concept_zyt_tcb_lr.
[10] Nordic nrf52832 memory layout. https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.nrf52832.ps.v1.1%2Fmemory.html.

[11] Nordic nrf52832 product specification v1.1. https://infocenter.nordicsemi.com/pdf/nRF52832_PS_v1.1.pdf.

[12] Nordic semiconductor. https://www.nordicsemi.com.

[13] Sam r21e data sheet. http://ww1.microchip.com/downloads/en/devicedoc/sam-r21_datasheet.pdf.

[14] Sam3x series data sheet. https://ww1.microchip.com/downloads/en/devicedoc/atmel-11057-32-bit-cortex-m3-microcontroller-sam3x-sam3a_datasheet.pdf.

[15] Segger j-link. https://www.segger.com/products/debug-probes/j-link/.

[16] Smart autonomous 32-bit microcontroller peripherals push the boundaries of ultra-low-power embedded system design. https://www.silabs.com/documents/public/white-papers/low-power-32-bit-microcontroller-dtm.pdf.

[17] Stmicroelectronics. https://www.st.com/content/st_com/en.html.

[18] Stmicroelectronics st-link. https://www.st.com/en/development-tools/st-link-v2.html.

[19] Stmicroelectronics stm32f103. https://www.st.com/en/microcontrollers-microprocessors/stm32f103rb.html.

[20] Stmicroelectronics stm32f103 reference manual. https://www.st.com/resource/en/reference_manual/cd00171190-stm32f101xx-stm32f102xx-stm32f103xx-stm32f105xx-and-stm32f107xx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf.

[21] Stmicroelectronics stm32f429. https://www.st.com/en/microcontrollers-microprocessors/stm32f429-439.html.

[22] Lucas Apa and Carlos Mario Penagos. Compromising industrial facilities from 40 miles away. *IOActive Technical White Paper*, 2013.

[23] Anomadarshi Barua and Mohammad Abdullah Al Faruque. Hall spoofing: A non-invasive dos attack on grid-tied solar inverter. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1273–1290, 2020.

[24] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[25] Chen Cao, Le Guan, Jiang Ming, and Peng Liu. Device-agnostic firmware execution is possible: A concolic execution approach for peripheral emulation. In *Annual Computer Security Applications Conference*, pages 746–759, 2020.

[26] 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 *NDSS*, 2018.

[27] Libo Chen, Yanhao Wang, Quanpu Cai, Yunfan Zhan, Hong Hu, Jiaqi Linghu, Qinsheng Hou, Chao Zhang, Haixin Duan, and Zhi Xue. Sharing more and checking less: Leveraging common input keywords to detect bugs in embedded systems. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[28] Kai Cheng, Qiang Li, Lei Wang, Qian Chen, Yaowen Zheng, Limin Sun, and Zhenkai Liang. Dtaint: detecting the taint-style vulnerability in embedded device firmware. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 430–441. IEEE, 2018.

[29] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2e: A platform for in-vivo multi-path analysis of software systems. *Acm Sigplan Notices*, 46(3):265–278, 2011.

[30] 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 *29th USENIX Security Symposium (USENIX Security 20)*, pages 1201–1218, 2020.

[31] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 463–478, 2013.

[32] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. Sok: Enabling security analyses of embedded systems via rehosting. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 687–701, 2021.

[33] Bo Feng, Alejandro Mera, and Long Lu. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1237–1254, 2020.

[34] Fred Glover. Heuristics for integer programming using surrogate constraints. *Decision sciences*, 8(1):156–166, 1977.

[35] 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*, pages 2245–2262, 2017.

[36] Hubert Högl and Dominic Rath. Open on-chip debugger–openocd–. *Fakultat fur Informatik, Tech. Rep*, 2006.

[37] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. Jetset: Targeted firmware rehosting for embedded systems. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[38] S. Khandelwal. Friday's massive ddos attack came from just 100,000 hacked iot devices. http://thehackernews.com/2016/10/ddos-attack-mirai-iot.html, 2016-10-27.

[39] 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 *Annual Computer Security Applications Conference*, pages 733–745, 2020.

[40] Taegyu Kim, Vireshwar Kumar, Junghwan Rhee, Jizhou Chen, Kyungtae Kim, Chung Hwan Kim, Dongyan Xu, and Dave Jing Tian. Pasan: Detecting peripheral access concurrency bugs within bare-metal embedded applications. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[41] Paul Kocher, Ruby Lee, Gary McGraw, and Anand Raghunathan. Security as a new dimension in embedded system design. In *Proceedings of the 41st annual Design Automation Conference*, pages 753–760, 2004.

[42] Philip Koopman. Embedded system security. *Computer*, 37(7):95–97, 2004.

[43] D. Lee. Amazon alexa heard and sent private chat. https://www.bbc.com/news/technology-44248122, 2018-05-24.

[44] Wenqiang Li, Le Guan, Jingqiang Lin, Jiameng Shi, and Fengjun Li. From library portability to para-rehosting: Natively executing microcontroller software on commodity hardware. *arXiv preprint arXiv:2107.12867*, 2021.

[45] Knud Lasse Lueth. State of the iot 2020: 12 billion iot connections, surpassing non-iot for the first time. https://iot-analytics.com/state-of-the-iot-2020-12-billion-iot-connections-surpassing-non-iot-for-the-first-time/, 2020-11-19.

[46] A. Mera, B. Feng, L. Lu, E. Kirda, and W. Robertson. Dice: Automatic emulation of dma input channels for dynamic firmware analysis. In *2021 2021 IEEE Symposium on Security and Privacy (SP)*, pages 1938–1954, Los Alamitos, CA, USA, may 2021. IEEE Computer Society.

[47] Sukhvir Notra, Muhammad Siddiqi, Hassan Habibi Gharakheili, Vijay Sivaraman, and Roksana Boreli. An experimental study of security and privacy risks with emerging household appliances. In *2014 IEEE conference on communications and network security*, pages 79–84. IEEE, 2014.

[48] NGUYEN Anh Quynh and DANG Hoang Vu. Unicorn: Next generation cpu emulator framework. *BlackHat USA*, 476, 2015.

[49] Srivaths Ravi, Anand Raghunathan, Paul Kocher, and Sunil Hattangady. Security in embedded systems: Design challenges. *ACM Transactions on Embedded Computing Systems (TECS)*, 3(3):461–491, 2004.

[50] 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 *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1544–1561. IEEE, 2020.

[51] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 19–36, 2020.

[52] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise MMIO modeling for effective firmware fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[53] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, volume 1, pages 1–1, 2015.

[54] Vijay Sivaraman, Hassan Habibi Gharakheili, Arun Vishwanath, Roksana Boreli, and Olivier Mehani. Network-level security and privacy control for smart-home iot devices. In *2015 IEEE 11th International conference on wireless and mobile computing, networking and communications (WiMob)*, pages 163–167. IEEE, 2015.

[55] Dokyung Song, Felicitas Hetzelt, Dipanjan Das, Chad Spensky, Yeoul Na, Stijn Volckaert, Giovanni Vigna, Christopher Kruegel, Jean-Pierre Seifert, and Michael Franz. Periscope: An effective probing and fuzzing framework for the hardware-os boundary. In *NDSS*, 2019.

[56] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[57] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 167–180, 2020.

[58] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *NDSS*, volume 23, pages 1–16, 2014.

[59] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1099–1114, 2019.

[60] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic firmware emulation through invalidity-guided knowledge inference. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021.