DICE: Automatic Emulation of DMA Input Channels for Dynamic Firmware Analysis

Alejandro Mera, Bo Feng, Long Lu, Engin Kirda, William Robertson
Khoury College of Computer Sciences
Northeastern University
Boston, USA
{mera.a, feng.bo, l.lu, e.kirda}@northeastern.edu

***Abstract*—Microcontroller-based embedded devices are at the core of Internet-of-Things (IoT) and Cyber-Physical Systems (CPS). The security of these devices is of paramount importance. Among the approaches to securing embedded devices, dynamic firmware analysis (e.g., vulnerability detection) gained great attention lately, thanks to its offline nature and low false-positive rates. However, regardless of the analysis and emulation techniques used, existing dynamic firmware analyzers share a major limitation, namely the inability to handle firmware using DMA (Direct Memory Access). It severely limits the types of devices supported and firmware code coverage.**

We present DICE, a drop-in solution for firmware analyzers to emulate DMA input channels and generate or manipulate DMA inputs (from peripherals to firmware). DICE is designed to be hardware-independent (i.e., no actual peripherals or DMA controllers needed) and compatible with common MCU firmware (i.e., no firmware-specific DMA usages assumed) and embedded architectures. The high-level idea behind DICE is the identification and emulation of the abstract DMA input channels, rather than the highly diverse peripherals and controllers. DICE identifies DMA input channels as the firmware writes the source and destination DMA transfer pointers into the DMA controller. Then DICE manipulates the input transferred through DMA on behalf of the firmware analyzer. DICE does not require firmware source code or additional features from firmware analyzers.

We integrated DICE to the recently proposed firmware analyzer P^{2}IM (for ARM Cortex-M architecture) and a PIC32 emulator (for MIPS M4K/M-Class architecture). We evaluated it on 83 benchmarks and sample firmware, representing 9 different DMA controllers from 5 different vendors. DICE detected 33 out of 37 DMA input channels, with 0 false positives. It correctly supplied DMA inputs to 21 out of 22 DMA buffers that firmware actually use, which previous firmware analyzers cannot achieve due to the lack of DMA emulation. DICE's overhead is fairly low, it adds 3.4% on average to P^{2}IM execution time. We also fuzz-tested 7 real-world firmware using DICE and compared the results with the original P^{2}IM. DICE uncovered tremendously more execution paths (as much as 79X) and found 5 unique previously-unknown bugs that are unreachable without DMA emulation. All our source code and dataset are publicly available.

I. Introduction

Modern embedded devices, equipped with increasingly powerful MCUs (microcontrollers) and rich network connectivity, are used as the building blocks in Internet-of-Things (IoT) and Cyber-Physical Systems (CPS). It is expected that 5.8 billion Enterprise and Automotive connected devices (IoT and CPS) will be in use in 2020 [12]. Therefore, the (in)security of embedded devices has profound implications on millions of devices, in terms of both data privacy and physical safety. Security vulnerabilities in firmware may allow attackers to control affected devices deployed in smart homes, connected vehicles, intelligent factories, power grids, etc., and in turn, steal critical data or manipulate device behavior. Such attacks have been on a rise and launched on cars [5], [28], Wi-Fi SoC [31], [33], manufacturing controllers [43], and more.

To improve embedded device security, researchers explored various approaches, including runtime attack mitigation [16], remote attestation [41], [37], and firmware analysis [10], [44], [6], [7]. Compared with other approaches, dynamic firmware analysis has low false positives, requires no hardware or software modification, and incurs zero overhead on production devices. Therefore, it is generally considered more practical.

A major challenge facing dynamic firmware analysis is the inability to fully run and test firmware at scale, due to hardware dependence on diverse peripherals. Many previous works either rely on real hardware components during analysis or port firmware to a conventional computer platform (e.g., x86), for which full emulators exist. P^{2}IM [10] is a recent work that for the first time enables dynamic firmware analysis without requiring actual hardware, source code, or porting firmware to a non-native platform. P^{2}IM removes hardware dependence by identifying processor-peripheral interfaces and supplying viable input data through such interfaces on behalf of peripherals. As a result, firmware can boot, run, and be tested in a generic emulator without using peripheral hardware or emulation.

Despite the tremendous progress made by previous works, one fundamental problem remains open: existing dynamic analyzers cannot support firmware taking input from peripherals via DMA (Direct Memory Access). When firmware reads from a DMA buffer in memory, which is supposed to contain input written directly by a peripheral, existing analyzers or emulators would fail to recognize it as a DMA read, and instead, treat it as a regular memory read (i.e., returning zero or invalid value to firmware). As a result, firmware cannot obtain any DMA inputs, which causes the execution to idle, a large portion of the firmware code to be unreachable/untested, or even the analysis session to crash.

The root cause of the problem lies in the very nature of DMA, which allows peripherals to access memory directly (with the help of a DMA controller, which is also a peripheral).

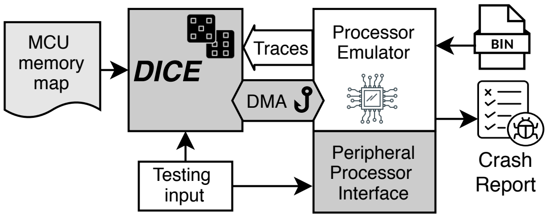

Fig. 1: DICE design overview as a drop-in component of firmware analyzer

A firmware analyzer, if not fully emulating all peripherals or entirely aware of their DMA activities, is unable to determine when and where in memory DMA-based I/O may occur. Therefore, the analyzer cannot tell, when firmware reads from a buffer in memory, whether the read operation is a DMA-based input event or just a regular load of data from memory.

Due to this open problem, existing analyzers either treat DMA to be out of scope (i.e., not supporting firmware that uses DMA), or use very simple heuristics to statically infer locations of DMA buffers in memory, which can be highly inaccurate and incomplete due to the dynamic nature of DMA.

Supporting DMA input is critical and necessary for dynamic analysis of embedded firmware, for the following reasons. First, to perform comprehensive dynamic tests of firmware, all input channels used by firmware need to be covered. Embedded devices take inputs from a wide range of peripherals through several channels, including MMIO (memory-mapped I/O) and DMA. Analysis of DMA-enabled firmware cannot reach or exercise the code that depends on DMA inputs. Second, many embedded devices use DMA (roughly 25% among the surveyed firmware, see §IV). In fact, DMA offers additional benefits to embedded devices. For example, besides improving data transfer rates, DMA allows processors to enter or stay in sleep or power-saving mode during data transfers, which is desirable for power-constrained embedded devices. Third, DMA is the only input channel used by certain peripherals and buses on embedded devices. For instance, input from CAN (Controller Area Network) and USB is accessible to firmware only via DMA.

In this paper, we present DICE (DMA Input Channel Emulation), a drop-in component (Fig. 1) for existing and future dynamic firmware analyzers to recognize and manipulate DMA-based peripheral input, and in turn, expand their analysis to cover firmware code, states, and vulnerabilities dependent on DMA input.

DICE adopts a non-conventional approach to supporting firmware input during analysis. Instead of aiming to emulate DMA controllers or DMA-enabled peripherals, which is practically impossible due to the complexity and diversity of these hardware components, we design DICE to emulate the *DMA input channels*, an implicit interface through which DMA input data flow to (or is consumed by) firmware execution. These channels manifest as memory buffers which peripherals directly write data to and firmware subsequently reads input from. DICE identifies DMA input channels when the firmware programs the sources and destinations of DMA transfers into the DMA controller. The sources and destinations are addresses within specific ranges (such as RAM), and are written to a fixed region in memory, namely the memory-mapped registers of the DMA controller. By identifying the creations and removals of such channels, DICE maps the sources and destinations of possible DMA data transfers. When firmware reads from a memory buffer that belongs to a DMA data transfer destination, DICE detects it as a DMA input event and signals the analyzer for further actions (e.g., supplying the read with fuzzer-generated input).

There are three main technical challenges that we tackled while pursuing this approach. First, DMA input channels are created and removed on demand by firmware during execution. DICE needs to dynamically identify these channels and monitor input events through them. Second, the high diversity in embedded hardware and software makes it difficult to develop a generic design for DICE that works on different types of architectures, peripherals, DMA controllers, and firmware. Third, a large number of firmware analyzed in practice are in binary form without source code or debug symbols. DICE should not need source code for identifying and emulating DMA input channels. Our design and implementation addressed all these challenges.

To evaluate the performance of DICE and demonstrate the benefits that it brings to existing firmware analyzers, we built two prototypes of DICE, based on P^{2}IM (for ARM Cortex-M) and a PIC32 emulator (for MIPS M4K/M) [36], respectively. We performed unit tests or micro-benchmarking on 83 sample firmware, representing 11 different MCU SoCs and covering ARM and MIPS architectures. DICE detected 33 out of 37 DMA input channels correctly, with 0 false positives. For the 22 channels that firmware actually used during the tests, DICE supplied inputs to 21 of them without errors. We also conducted fuzzing tests on 7 real firmware and measured the code and path coverages with and without DICE enabled on P^{2}IM. Thanks to DICE's automatically emulated DMA input channels, the basic block coverage increased by 30.4% and the path coverage jumped remarkably by 79X. Even with a fairly primitive memory sanitizer and a 48-hour fuzzing session, DICE detected 5 unique previously unknown bugs in the firmware.

The source code of DICE, the integrations with firmware analyzers and emulators, and all the firmware tested are publicly available at https://github.com/RiS3-Lab/DICE-DMA-Emulation.

In sum, our work makes the following contributions:

- We study and advocate the importance of supporting DMA-based peripherals and input in dynamic firmware analysis; we identify the lack of DMA support as a common limitation for all existing firmware analyzers.
- We present DICE, which enables dynamic firmware analyzers to support DMA-based peripherals, and recognize and manipulate DMA inputs for testing otherwise skipped or unreachable code/states in firmware.

- When designing DICE, we overcome the challenges posed by the dynamic nature of DMA, the diverse hardware and software of embedded devices, and the unavailability of firmware source code.
- We implemented two prototypes of DICE based on P^2IM and a MIPS PIC32 emulator, respectively. Our evaluation shows that DICE: (1) achieved highly accurate DMA input channel identification and emulation, (2) helped P^2IM significantly improve its analysis coverage and discover 5 new bugs in tested firmware.
- We analyzed all the bugs discovered by DICE and found all of them are remotely exploitable. They have security consequences such as information leakage, data corruption, and denial-of-service. These bugs cannot be found by exiting firmware analyzers due to the lack of DMA support or emulation.

II. Motivation

MCU-based embedded devices (or MCUs in short) integrate on a single chip a main processor, RAM, Flash, and diverse peripherals. Their energy-efficiency and sufficient computing power make them the ideal building blocks of IoT devices and cyber-physical systems. The entire software stack on MCUs, referred to as firmware, contains OS/system libraries, drivers, and application-level logic in a monolithic form.

Firmware, similar to other software, may contain programming errors or bugs that can be exploited by attackers [32], [33], [31]. These security bugs in firmware, although often similar in nature, can cause severe and unique consequences because MCUs are widely used in mission-critical settings (e.g., industrial systems) and have direct physical outreach (e.g., controlling vehicle movements).

Unfortunately, bugs in MCU firmware are more difficult to detect than bugs in conventional computer software, due to existing analysis tools' limited support for highly diverse and heterogeneous MCU hardware. Specially, various kinds of peripherals are used as the main communication channels through which firmware communicates with other devices and interacts with the physical environment. However, existing firmware analysis methods cannot fully model or emulate peripherals, and thus, fail to trigger or reach a large portion of firmware code during analysis, missing opportunities to detect bugs.

Take a MCU-based GPS device as an example (Figure 2). It uses UART (universal asynchronous receiver-transmitter) to receive NMEA [13] serialized messages from a GPS antenna. These messages are copied to RAM via DMA. The firmware then parses the messages and computes the location information, which is later copied via DMA to an LCD (Liquid Crystal Display) attached to the SPI (Serial Peripheral Interface). This device also has other peripherals for receiving inputs or delivering outputs.

Without peripheral awareness or support, dynamic analysis of this GPS firmware cannot reach most of the code because the firmware execution cannot receive any input or even boot up the device. Some recent work [44], [6], [10] addressed the peripheral dependence issue using different approaches. However, they mostly focused on the simple peripherals that only use memory-mapped I/O (e.g., those inside the box ② in Figure 2). [7] relies on manually identified DMA buffers to partially support simple DMA-based I/O, and [2] completely removes DMA through replacing HAL (hardware abstraction layer) functions with manually-written hooks.

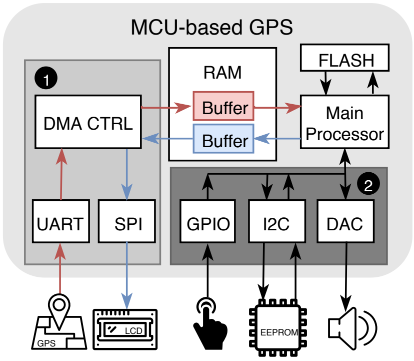

Fig. 2: A MCU-based GPS and its I/O flows: ① DMA-based interfaces, ② MMIO/interrupt-based interfaces.

So far no existing work supports complex peripherals that use DMA to communicate with firmware (e.g., those inside box ① in Figure 2). Therefore, firmware using DMA still cannot be fully analyzed and all their bugs/vulnerabilities detected. According to our survey (§IV), most nontrivial firmware use DMA for both performance and energy-saving reasons. In these firmware, the majority of code cannot run or be tested without DMA support.

III. Background

A. DMA on MCU Devices

Direct Memory Access (DMA) is a widely used architectural feature that allows peripherals to access the main system memory (RAM) without involving main processors. The goal of DMA is to improve performance when transferring data between RAM and peripherals. The introduction of DMA dates back to the 1960s on the DEC PDP-8 minicomputer [3]. DMA is ubiquitously adopted by today's computers.

DMA is also widely used in modern embedded devices powered by microcontrollers (MCU). Similar to DMA on conventional computers, DMA on MCU devices benefit from the performance improvement in data transfer between RAM and peripherals. Unlike conventional computers, MCU devices use DMA not just for performance reasons but also for saving power or energy. DMA allows large or slow data transfers to take place while the main processor (i.e., a major power consumer on embedded devices) is asleep or stays in the low-power mode.

From the programmers' perspective, DMA provides a standard interface that abstracts away peripheral internals. Programmers can use the same interface exposed by DMA controllers to exchange data with different peripherals. An example of this abstraction is the HAL library of ChibiOS [9],

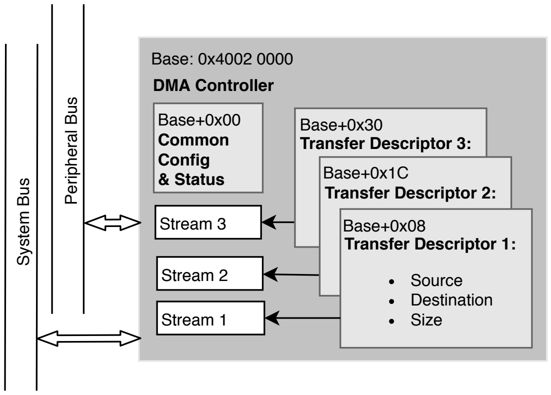

Fig. 3: DMA controller and detail of DMA transfer descriptors, each using 20 bytes of memory span in the MMIO region (some fields of transfer descriptors are omitted for simplicity).

where the low-level drivers use DMA transparently through a uniform interface to receive input from different peripheral.

We surveyed the availability of DMA on MCUs and the uses of DMA in firmware. We analyzed 1,356 MCUs from a major vendor and 1,000 repositories from Github targeting MCUs exclusively. Our analysis showed that 94.1% of the modern MCUs (32-bit architecture) are DMA capable, which demonstrates the ubiquitous support for DMA on modern embedded devices. As for the firmware, 25.1% of the compilable or precompiled MCU repositories contain DMA related debugging symbols, a strong indicator of firmware using DMA. The results underline the importance and urgency of supporting DMA-capable peripherals and DMA-based input in firmware analysis. We present the details of our survey in §IV.

B. DMA Workflow on ARM Cortex-M

We choose ARM Cortex-M as the reference architecture for designing DICE because it is the most common architecture used in modern MCU devices and IoT. DMA works on this architecture in a similar way as it does on others, such as MIPS, which DICE also supports. Below we describe a simplified DMA workflow and introduce the basic concepts, which are necessary to understand the design of DICE.

A *DMA controller* plays the central role in the DMA workflow. It is an on-chip peripheral that transfers data into memory on behalf of peripherals. A *DMA transfer* is a single movement of data to memory. It starts with firmware creating a *transfer descriptor*, specifying the transfer's source, destination, size, etc., and writing this transfer descriptor to the DMA controller, a step called DMA configuration. In addition, firmware also specifies which *DMA stream* (a physical data channel inside the DMA controller) should be used for the transfer. Figure 3 shows a DMA controller with three DMA streams configured for three different DMA transfers. After the configuration finishes and the data becomes ready at the source, the DMA controller performs the transfer asynchronously without involving the firmware or the processor. The source and destination of a transfer are also referred to as *transfer pointers*.

A single MCU may have more than one DMA controllers, each covering a set of connected peripherals. A DMA stream in a DMA controller can only be used for supporting one DMA transfer at any given time. High-speed on-chip peripherals, such as Ethernet, USB, and CAN, may integrate a DMA controller for higher throughput. Note that it is the DMA controller, not the source peripheral, that directly writes the data to memory. DMA controllers are the only peripherals that can directly access RAM thanks to the special permission they have to access the system bus as master devices.

C. Challenges of Supporting DMA in Firmware Analysis

Dynamic nature of DMA: The dynamic nature of DMA (*e.g.,* configurations, streams, and transfers) makes it challenging to detect DMA input channels or infer transfer pointers. Some previous work relies on manual and static identification of DMA buffers [7]. This strategy requires source code or reverse engineering of firmware. Despite the poor scalability, it can neither be complete nor accurate because it may miss dynamically created DMA buffers or mistakenly treat regular memory regions as DMA buffers. A reliable and practical approach should consider that DMA transfers are created and performed dynamically on demand. It should not require human efforts or rely purely on static inference.

MCU hardware and software diversity: The diversity of MCU firmware prevents the application of DMA models designed for desktop OS, where a generic hardware abstraction layer (HAL) exists (e.g. for Linux [39]). For MCU firmware, there is no standard DMA interface or a prevalent OS that provides a generic abstraction for various hardware peripherals. Moreover, it is practically impossible to develop an emulator for each DMA controller and peripheral due to the high diversity of the hardware, as observed by [10], [44], [29], [6]. Therefore, it is necessary yet challenging to design a single DMA emulation method that can work with a wide range of firmware, architectures and peripherals.

IV. Survey on DMA Availability and Usage on MCUs

We surveyed 1) the availability of DMA controllers for MCUs on the complete product portfolio of a top MCU vendor, and 2) DMA usage by firmware on a large collection of open-source repositories from GitHub.

A. DMA Availability on MCUs

We analyzed on March 2019 the MCU product portfolio (which documents all MCU models a vendor produced) of Microchip Semiconductors [24], a top MCU vendor in terms of market share [19]. Our analysis included 1,356 MCUs from 32-bit (ARM Cortex-M, MIPS), 16-bit (PIC16), and 8-bit (PIC8 and AVR) architectures. We excluded the legacy architecture 8051. We grouped MCUs into families using Microchip's quick reference guides [27], [25], [26]. MCUs within the same family, although different in memory size and packaging, use the same architecture and have mostly the same

MCU architecture	DMA support	No DMA support
32-bit (ARM Cortex-M, MIPS)	32 (94.1%)	2 (5.9%)
16-bit (PIC16)	2 (11.1%)	16 (89.9%)
8-bit (PIC8, AVR)	7 (11.3%)	55 (88.7%)

TABLE I: DMA availability on Microhip MCUs families (March 2019)

on-chip peripherals and DMA availability (as DMA controller is also an on-chip peripheral). Therefore, grouping MCUs into families allow us to better analyze and describe DMA availability on those MCU families. Our analysis demonstrated that 94.1% of modern 32-bit MCU families include one or multiple DMA controllers, or include DMA capable peripherals (e.g., USB, CAN, Ethernet). On the other hand, only 11% of 16-bit and 8-bit MCU families support DMA (Table I).

B. DMA Usage by Firmware

We collected 1,000 unique repositories from GitHub by searching keywords and topics related to microcontrollers, IoT, and DMA. Our dataset only included repositories for ARM Cortex-M architecture, the most popular architecture for IoT devices and cyber-physical systems. We found most repositories cannot be compiled because of missing building scripts (e.g., makefiles) or libraries, or unavailability of proprietary IDE (Integrated Development Environments) and toolchains. We were able to compile or directly download 350 ELF binaries from our dataset. We performed a basic static analysis on both source code and EFL files.

Our analysis shows that 920 out of 1,000 (92%) repositories include DMA related header files or DMA driver source code. However, only 88 out of 350 (25.1%) ELF files contain DMA related debug symbols. This is because DMA header files and driver code are always distributed as part of SDKs, regardless of firmware usage. Therefore, we cannot use the inclusion of DMA header files or driver code as an indicator of DMA usage. Instead, we use the appearance of DMA debug symbols in ELF files as the indicator for DMA usage by firmware, and estimate 25.1% (88 out of 350) firmware use DMA.

We also observed that most firmware that use DMA are for battery-powered devices (e.g., drones, handhelds, smart watches), or require high data communication throughput (e.g., DSPs and LCDs). We admit that usage of DMA is an architectural and/or design decision made by firmware developers for the specific application.

V. System Design

DICE provides dynamic firmware analyzers the ability to recognize and handle DMA-based input from peripherals, thus allowing firmware using DMA to be analyzed and firmware code dependent on DMA input to be executed and tested, which is previously impossible. DICE meets the following design goals that we set:

- **Hardware independence:** DICE should not rely on actual hardware peripherals. It should be generic to support a wide range of architectures, peripherals, and DMA controllers used in embedded devices.
- **Firmware compatibility:** DICE should be compatible with all possible ways that firmware may use DMA as input channels, regardless of how DMA is configured or data is consumed.
- **Dynamic DMA:** DICE should fully consider the dynamic nature of DMA and be able to capture DMA input events through dynamically allocated memory regions.
- **No source code:** DICE should not require source code or debug symbols of firmware.
- **Integration with analyzers:** DICE should not need hardware or software capabilities that common firmware analyzers do not have. Integrating DICE should not require major changes to existing analyzers.

DICE achieves the design goals thanks to a novel approach to supplying DMA input to the firmware. The approach is inspired by our observation on the generic patterns that firmware follows when performing DMA configurations and data transfers. These patterns are observed across different firmware on various embedded devices using distinct DMA controllers and peripherals. In fact, these patterns reflect the de facto protocol used by embedded firmware and peripherals when exchanging data via DMA. By detecting these patterns and intercepting the DMA configuration and data transfer events, DICE monitors DMA input channels as they are created, used, and disposed. As a result, DICE can capture (and respond to) all DMA input events, which inevitably go through the monitored interfaces.

Next, we explain the abstract notion of DMA input channels and discuss the DMA configuration and data transfer patterns that DICE uses to identify DMA input channels.

A. DMA Input Channels

DMA input channel is an abstract notion we formulated, on which the core idea of DICE is derived. Such channels can be viewed as the conceptual bridges through which firmware and peripherals exchange data via DMA. They manifest as dynamically allocated memory buffers that firmware and peripherals agree upon for transferring data. A DMA controller serves as the proxy for a peripheral to write data in the DMA memory buffers without involving the main processor. The data is then read from the memory buffer by the firmware as input from the peripheral. Therefore, if all DMA input channels (or their manifestations, i.e., memory buffers used as DMA data exchanges) can be recognized upon their creation, access, and disposal, all DMA data transfers can be monitored and interposed, which allows DICE to supply DMA input to the firmware execution without using actual peripheral hardware or understanding the inner workings of peripherals or DMA controllers. *DMA output channel* follows the same definition with *DMA input channel*, but with the opposite transfer direction (i.e., data is transferred to peripherals via DMA).

Table II shows all possible combinations of sources and destinations for DMA input and output channels. Those with Flash as the destinations are invalid because DMA is only meant for transferring data to memory, including RAM and

Source	Destination	Valid	Type
Peripheral	RAM	Yes	DMA input channel
Peripheral	Peripheral	Yes	DMA output channel
Peripheral	Flash	No	N/A
RAM	Peripheral	Yes	DMA output channel
RAM	RAM	Yes	DMA input channel
RAM	Flash	No	N/A
Flash	RAM	Yes	DMA input channel
Flash	Peripheral	Yes	DMA output channel
Flash	Flash	No	N/A

TABLE II: Possible combinations of sources and destinations for DMA input and output channels

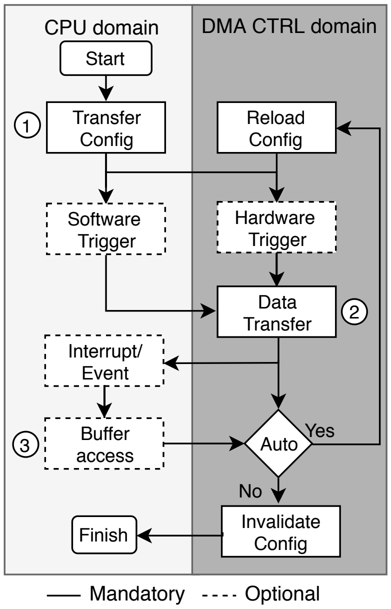

Fig. 4: Life-cycle of a DMA transfer

memory-mapped peripheral regions. Among the valid combinations, we focus on those with RAM as the destination because they are DMA input channels that can directly influence the execution and analysis of firmware. DICE does not handle DMA output channels which do not directly influence firmware execution. Unless otherwise noted, we refer to DMA transfers that go through DMA input channels simply as DMA transfers.

Each DMA transfer takes three steps. First, firmware establishes the DMA input channel by sending a transfer descriptor to the DMA controller as part of the transfer configuration (① in Fig. 4). The transfer descriptor specifies, among other things, which physical DMA stream inside the DMA controller should be used for transferring the data, along with the source and destination addresses as well as the transfer size. The source address points to the to-be-transferred data stored in peripheral registers, Flash, or RAM. The destination points to the memory buffer where the firmware expects the input data to be transferred. We refer to a source or destination address as a transfer pointer. Second, when the input data becomes available at the source, as indicated by an interrupt, the DMA controller copies the data from the source to the destination. Note that the actual data transfer and direct memory access are performed by the DMA controller on behalf of peripherals (② in Fig. 4). Third, after finishing the data transfer, the DMA controller signals the firmware and closes the DMA stream. It tears down the current DMA input channel and leaves the input data in the memory buffer for the firmware to use (③ in Fig. 4).

A transfer descriptor is valid only for one DMA transfer, which goes through the DMA input channel identified by the transfer descriptor. Firmware may sometimes use the auto-reload feature of DMA controllers, whereby a transfer descriptor is valid for multiple consecutive DMA transfers until reconfigured.

The lifespan of a DMA input channel starts when DMA controller receives the transfer descriptor and the DMA stream is configured. It terminates when the data transfer finishes. A channel's exit end is attached to a dynamically allocated memory buffer (i.e., the destination). This per-transfer and highly dynamic nature of DMA input channels makes it challenging to detect and monitor them. For the same reason, previously proposed techniques for detecting DMA input using static heuristics are inaccurate and incomplete.

B. Capturing Stream Configurations

DICE dynamically captures DMA stream configuration events to extract the information needed for identifying DMA input channels, such as destinations and sizes. However, stream configurations are not directly visible by firmware emulators or analyzers due to the semantic gap—such a configuration (i.e., firmware writing a transfer descriptor to memory-mapped registers), in the eyes of an emulator, looks the same as a regular memory write by firmware.

We observed a fairly distinct pattern followed by DMA stream configurations: writing some values within a specific range to a fixed region in memory. This pattern echos the essential operation performed in each stream configuration—writing the source and destination transfer pointers to the DMA controller registers. These peripheral registers are always mapped in the MMIO region (0x40000000–0x5fffffff). DMA transfers through DMA input channels move data from peripheral/Flash/RAM to RAM. Therefore, the value of source transfer pointers must be in the range of 0x40000000–0x5fffffff (for peripheral MMIO), 0x20000000-0x20004fff (for RAM), or 0x8000000-0x801ffff (for Flash) [1]. Similarly, destination pointers must point to the RAM region. These regions are not very big in size (2kB to 512kB), which means the value range of transfer pointers is not very wide. DICE uses this pattern (*i.e.,* two writes of pointer values to consecutive locations in the MMIO region) to detect DMA stream configurations.

We also observed that a DMA controller can support multiple DMA streams (Figure 3). The source and destination

[1] The RAM and Flash address ranges are taken from the STM32F103 MCU as an example. Different MCU may use slightly different ranges for RAM and Flash, which are specified in their data-sheets. When data-sheets are not available, DICE uses the largest ranges allowed by the architecture (a 512MB region) as the RAM/Flash region.

transfer pointers of a particular stream are recorded in two consecutive MMIO registers in the DMA controller. The write operations to these registers are 32-bit in width and 4-byte aligned in address. This pattern allows DICE to identify multiple stream configurations on the same DMA controller, and more importantly, filter out pointer-like values that are written to the MMIO region yet not transfer pointers (*i.e.,* the write operations are sparse, unaligned or in a different data width).

Based on our tests on real firmware and devices of various kinds (§VII-A), this pattern reliably indicates stream configurations and is never seen in other types of memory write operations. It means that, empirically, firmware writes two pointer/address values to two consecutive MMIO registers only for the purpose of DMA stream configuration.

DICE looks for the stream configuration pattern while it monitors memory writes by firmware during execution. When one is observed, DICE captures the DMA stream configuration and extracts from it the transfer pointers. Then DICE needs to identify the direction of the DMA transfer, because when DICE identifies two transfer pointers that point to, for example, peripheral and RAM regions, it does not know whether the DMA transfer is from peripheral to RAM, or the opposite.

DICE determines the transfer direction by monitoring memory accesses made by the firmware through the transfer pointers. If the firmware reads from the RAM address referenced by the transfer pointer, the DMA transfer is from peripheral to RAM (a DMA input channel that we are interested in). If the firmware writes to the RAM address referenced by the transfer pointer, the DMA transfer is from RAM to peripheral. This is determination of transfer direction is intuitive. For incoming DMA data, firmware needs to read the data after it is transferred into RAM from a peripheral. For outgoing data, firmware needs to write it into RAM before DMA transfers it to a peripheral.

DICE captures a DMA stream configuration transferring data to RAM, and finds its destination address (i.e., the beginning of the memory buffer for receiving the current DMA transfer). A newly captured DMA stream configuration marks the establishment of a DMA input channel for an upcoming DMA transfer. Its destination address locates the memory buffer that the firmware will read the DMA input from. DICE keeps track of all active DMA input channels and their destination addresses.

Pattern variations: We encountered one variant of the stream configuration pattern described above, namely, multiple destination addresses are specified in one configuration. This occurs when a DMA transfer operates in the circular mode [40]. Although slightly different from normal stream configurations, this variation still obeys the pattern that DICE uses to capture stream configurations. It simply uses one (or more) extra destination transfer pointer (*e.g.,* three pointers written to three consecutive MMIO peripheral registers).

Pattern limitation: We identified two limitations of the pattern for capturing DMA stream configuration. First, the pattern assumes that firmware always writes transfer descriptors, including the transfer pointers, to the memory-mapped peripheral region that corresponds to DMA controller registers. Although this is true for most firmware and MCU devices, we are aware of some rare cases where firmware writes transfer descriptors to RAM, rather than DMA controller registers. This type of stream configuration is used only in some high-end SoCs, which resembles the desktop architectures. In this case, DMA controllers fetch transfer descriptors from RAM in ways unique to the individual DMA controllers. We did not encounter such cases in our experiments. DICE cannot capture the stream configurations performed this way.

Second, some DMA controller models (e.g., NRF52832 easyDMA [1]) require only the destination address, not the source-destination pair, when firmware configures a DMA stream (using an implicit source address). As DICE needs to observe both the source and destination transfer pointers in order to identify a DMA stream configuration, DICE cannot identify destination-only stream configurations. However, based on our evaluation, only 2 DMA controller models support this type of configuration. DICE missed only 7 out of 52 DMA stream configurations due to this limitation. We discuss the details in §VII.

C. Responding to DMA Data Read

After DICE captures a DMA stream configuration and finds the destination address, it places an access hook on the destination address, which allows DICE to identify and respond to firmware's read from the corresponding DMA memory buffer. However, this DMA read identification and response process is not as straightforward as it may sound, due to two technical challenges: the unknown buffer size, and the dynamic termination of DMA input channels.

Unknown buffer size: Although DICE can reliably capture every DMA stream configuration and extract the destination address, it cannot accurately find the transfer size or the buffer size from a captured configuration event. This is because transfer sizes may take a wide range of values, unlike destination addresses, whose values are bounded by the valid DMA memory regions and therefore fairly distinguishable.

DICE needs to know where each DMA buffer ends in order to determine if a memory read falls in such a buffer. An intuitive solution is to extract buffer size from the debug symbols generated by compiler. At DMA stream configuration, debug symbols are looked up to figure out which buffer is allocated at the destination address and what the buffer size is. However, debug symbols are absent in MCU firmware, which are stripped binary blobs containing only code and data. Moreover, this approach cannot identify the size of dynamically allocated buffers that are widely used in DMA transfers.

Instead, DICE adopts an approach supporting dynamically created DMA buffers without using debug symbols. It dynamically infers the bounds of DMA buffers by observing firmware's access. The inference leverages the fact that firmware typically reads a DMA buffer consecutively in space (from the beginning to the end, but not necessarily consecutive in time). Although in theory firmware may not start reading DMA buffers from the beginning, we did not observe such a case in our experiments on real firmware. When input data comes from a peripheral that uses a different endianness than the MCU's, firmware may start reading the buffer several bytes after the beginning. DICE considers and handles such cases.

For each firmware access to a detected DMA buffer, DICE calculates a *span* (*i.e.,* the possible extent to which this DMA buffer may extend beyond the currently known boundary). The *span* size is set to twice of the data width of the memory read operation (LOAD). For instance, the span for a buffer accessed via a 32-bit LOAD will have the size of 64 bits (8 bytes). Having a span for each buffer access allows DICE to incrementally infer the buffer size and recognize the endianness conversion that the firmware may perform. For example, after multiple two-byte inputs are transferred from big-endian peripherals by DMA, firmware running on a little-endian CPU will read the DMA buffer in 1-byte data width to convert the endianness. As byte 1 is read before byte 0, having a span twice of the data width allows DICE to catch this behavior. DICE monitors memory read operations while the DMA input channel is in use. When a memory read falls in the span, DICE expands the detected DMA buffer to include the read address. This process is described in Algorithm 1. The dynamic and incremental expansion of detected DMA buffers allows DICE to identify and handle firmware's DMA read while continuously inferring the true buffer size, especially for dynamically allocated DMA buffers.

Algorithm 1 DMA buffer size inference

```
$PerceivedSize \leftarrow 0$
while DMA input channel is valid do
    if memory is read then
        $Span.Size \leftarrow 2 * Read.DataWidth$
        $Span.Base \leftarrow (Buffer.Base + PerceivedSize)$
        if Read falls in Span then
            $PerceivedSize \leftarrow (Read.addr - Buffer.Base) + Read.DataWidth$
        end if
    end if
end while
```

Dynamic channel terminations: DMA input channels are per DMA transfers and not permanent. A channel is created when the underlying DMA stream is configured and terminated when the stream is invalidated or used for a different transfer. DICE tracks the life cycles of DMA input channels and detects dynamic channel terminations. This is necessary because stale channels, if not recognized, can cause DICE and the firmware analysis to mistakenly treat regular memory access as DMA input events and thus corrupt firmware execution. It is worth noting that the static technique used by the previous works for identifying DMA buffers suffer from this issue.

DICE watches for two types of signals that indicate channel terminations. First, if a newly captured DMA stream configuration references the same DMA stream as a previously captured one did, the stream is now reconfigured to support a different DMA transfer, and therefore, the previous DMA input channel is now terminated. Second, if firmware writes to a memory buffer that corresponds to a DMA input channel, the channel is implicitly terminated and the buffer is no longer used for receiving DMA input. When a channel termination is detected, DICE removes the access hook on the buffer. As a result, DICE no longer treats it as a DMA buffer or supplies input to it. The previously generated input data still remains in the buffer, which firmware may continue using.

VI. Implementation

We implemented two prototypes of DICE as drop-in components on top of the P^2IM framework (for ARM Cortex-M architecture) and a PIC32 emulator (for MIPS M4K/M-Class architecture) [36].

DICE leverages the tracing and hooking capabilities of QEMU, which is the processor emulator used in P^2IM and the PIC32 emulator. It is worth noting that these emulators/analyzers do not provide any DMA capabilities as part of their original design. Also, DICE does not rely on any internals of these systems, making DICE portable to other firmware analyzers.

Although the implementation of DICE in the two prototypes are virtually the same, they could not be integrated under a single implementation because these systems are based on different and heavily modified QEMU forks. Also, each emulator has a different set of capabilities. P^2IM can run a broader range of firmware and perform fuzz testing thanks to its automatic modeling of processor-peripheral interfaces and integration with AFL [45]. The PIC32 emulator can only run a small set of firmware because it emulates a limited number of peripherals. We consider extending P^2IM to support MIPS M4K/M-Class architecture, or port P^2IM to the PIC32 emulator out of the scope of this paper.

On the PIC32 emulator, any access to non-emulated peripherals will halt the emulator, which significantly limits firmware execution. To avoid halting the firmware, we created default memory-mapped register handlers in the emulator. These handlers do not implement any meaningful peripheral functionalities. They treat registers as memory, i.e, they record the value that is written to a register, and return it when the register is read. Besides, we implemented a round-robin strategy (inspired by P^2IM design) for triggering DMA-related interrupts and invoking the associated ISR (interrupt service routine). This allows the firmware to read DMA buffers.

The DICE implementation on P^2IM includes 1,237 lines of C code added to QEMU. The implementation on PIC32 emulator has 1,620 lines of C code. Despite the different target architectures (ARM and MIPS), both implementations are fairly similar and the description below applies to both. The heuristic to capture DMA stream configurations is implemented in QEMU's `unassigned_mem_write` function, which manages MMIO operations related to the peripheral memory area (① in Fig. 5). This function has access to traces of the MMIO operations. These traces include information about the operation type (read/write), value, address and data

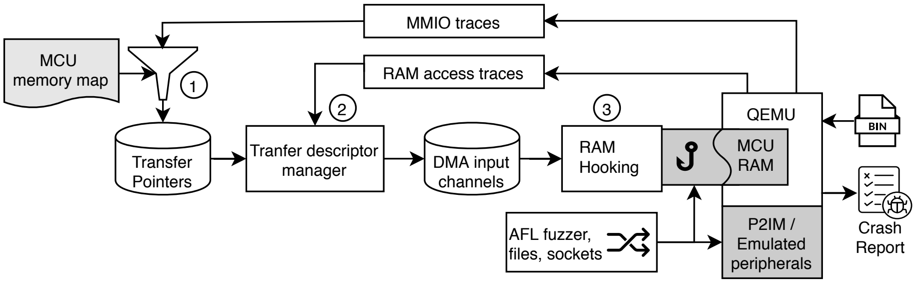

Fig. 5: DICE implementation (P^{2}IM version). ① Identification of DMA transfer pointers. ② transfer descriptor management. ③ Transfer size and DMA hooks.

width (8, 16 or 32 bits) of the MMIO. The transfer descriptor manager (② in Fig. 5) is a software component that tracks and orchestrates the life cycles of DMA input channels. We implemented this component and data structures in the `helper_le_ld_name` function. This function is defined in the softmmu_template.h file and has access to traces of any memory read in all memory areas of MCU. We modified QEMU to invoke this function on every LOAD instruction. This method is described as the "slow path" for memory access, according to QEMU's documentation [4].

The `helper_le_ld_name` function allows DICE to place memory hooks on-the-fly (③ in Fig. 5), according to the captured DMA input channels and accesses of firmware to the destination address. Also, `helper_le_ld_name` provides the effective data width (8, 16 and 32-bit) of the LOAD instruction emulated by QEMU. This information is used to adjust the *span* that is used to identify the buffer size.

The DICE implementation on P^{2}IM supports fuzzing. It reuses the underlying AFL engine and the TriForce [18] QEMU extensions of P^{2}IM. DICE supports files, network sockets, and other input methods that a firmware analyzer requires to provide input to firmware. This architecture allows DICE to be added as a drop-in component to other firmware analyzers and enhance it with the automatic emulation and manipulation of DMA input channels.

VII. Evaluation

We evaluated DICE from three different angles: 1) whether it can accurately identify DMA input channels on firmware that run on different architectures, MCUs and OSes; 2) how much its runtime overhead is; 3) whether it can support fuzz-testing on real firmware that uses DMA, and more importantly, find bugs that cannot be found by existing dynamic firmware analyzers.

To verify 1) and 2), we performed unit tests and micro-benchmarks on sample firmware in §VII-A. As for 3), we fuzz-tested 7 real-world firmware with DICE (integrated with P^{2}IM) and found 5 previously unknown bugs in §VII-B. We also discussed our insights into DMA emulation.

All experiments were conducted on a dual-core Intel Core i5-7260U CPU @ 2.20GHz, 8 GB of RAM, and a fresh installation of Ubuntu 18.04 LTS. We will release all the firmware images after the paper is published.

MCU	Architecture	DMA cntlr. model	Vendor
NRF52832	ARM Cortex-M4	a	Nordic Semiconductors
NRF51822	ARM Cortex-M0	a	
NUC123	ARM Cortex-M0	b	Nuvoton
LPC1837	ARM Cortex-M3	c	NXP
MK64F	ARM Cortex-M4	d	
SAM3X	ARM Cortex-M3	e,f	Microchip/ Atmel
PIC32MX795	MIPS M4K	g	
PIC32MZ2048	MIPS M-class	g	
STM32F103	ARM Cortex-M3	h	ST Microelectronics
STM32L152	ARM Cortex-M3	h	
STM32F429	ARM Cortex-M4	i	

TABLE III: Architectures, MCUs, DMA controller models and vendors covered by the 83 sample firmware. Some MCUs share the same DMA controller model. SAM3X has two different DMA controller models. For brevity, we use the letters (a-i) to differentiate the controller models.

A. Unit Tests on Sample Firmware

We conducted this experiment to show that DICE can accurately identify DMA input channels on different architectures, MCUs and OSes (i.e., DICE is accurate, hardware-independent and OS-agnostic). We collected a set of 83 sample firmware from the official MCU SDKs and open-source repositories. These sample firmware are developed by MCU vendors or open-source contributors and serve as templates for firmware developers. The sample firmware are suitable for our micro-benchmarking because each implements a self-contained logic and they collectively cover: 1) different architectures, vendors, MCUs, and DMA controllers, 2) different OSes/system libraries, 3) different combinations of DMA sources and destinations.

Experiment Setup: As shown in Table III, the 83 sample firmware cover 2 architectures, 11 different MCUs from 5 major vendors, and 9 unique DMA controller models.

The sample firmware are based on the real-time OSes (RTOS) or system libraries chosen by the vendors (SDKs), including BSD, Arduino, Mynewt, NuttX, Riot OS and ChibiOS (Table IV). This set of firmware include not only those that use all types of DMA input channels, but also those that do not use DMA (non-DMA-enabled firmware). The non-DMA-enabled firmware include the whole unit test suite used in [10] and 2 ports of BSD for MCUs. Including both DMA-enabled and non-DMA-enabled firmware allows us to comprehensively evaluate DICE, in terms of its accuracy and compatibility.

ARM Cortex-M0/M3/M4 DMA-Enabled Firmware

Firmware	MCU	OS/SDK	Source Code
ADC PDC *	SAM3x	Arduino	http://nicecircuits.com/playing-with-analog-to-digital-converter-on-arduino-due/
SPI DMAC Slave *	SAM3x	Atmel ASF	https://asf.microchip.com/docs/latest/sam.drivers.spi.spi_dmac_slave_example.sam3x_ek/html/index.html
USART DMAC			
ADC slider	STM32F103	ChibiOS	https://osdn.net/projects/chibios/downloads/70739/ChibiOS_19.1.0.7z/
I2C accelerometer	STM32F103		
SPI	STM32F103		
UART	STM32F103		
ADC slider	STM32F429		
SPI	STM32L152		
UART	STM32L152		
ADC SW DMA *	STM32F103	STM32CubeF1	https://www.st.com/en/embedded-software/stm32cubef1.html
ADC Timer DMA *			
I2C DMA IT			
I2C DMA Adv IT			
I2C TxRx DMA *			
SPI Half DMA *			
SPI Half DMA Init			
SPI Full DMA *			
USART TxRx DMA *			
USART Full DMA *			
UART H.Term. DMA			
UART 2Boards DMA			
SPI Full EX. DMA *			
I2C 2Boards DMA			
PDMA M-M	LPC1837	LPC Open	https://www.nxp.com/downloads/en/software/lpcopen_3_02_lpcxpresso_mcb1857.zip
Serial console	NRF52832 NRF51822	Mynewt	https://mynewt.apache.org/download/
SPI slave	NRF51822	Nordic SDK	https://www.nordicsemi.com/Software-and-tools/Software/nRF5-SDK/Download
SPI master	NRF52832		
SPI slave	NRF52832		
Serial DMA	NRF52832		
PDMA USART	NUC123	OpenNuvoton	https://github.com/OpenNuvoton/NUC123BSP.git
PDMA M-M			

ARM Cortex-M3/M4 Non-DMA-Enabled Firmware

Firmware	MCU	OS/SDK	Source Code
P^{2}IM unit test suite (44 firmware images) [10]	STM32F103 SAM3x MK64F	Arduino, Riot OS, NuttX	https://github.com/RiS3-Lab/p2im-unit_tests

MIPS M4K/M-class DMA-Enabled Firmware

Firmware	MCU	OS/SDK	Source Code
PIC32MX_UART	PIC32MX795	Microchip Harmony v3	https://microchipdeveloper.com/harmony3:pic32mx470-getting-started-training-module
PIC32MX_test	PIC32MX795		
PIC32MZ_UART	PIC32MZ2048		https://microchipdeveloper.com/harmony3:pic32mzef-getting-started-training-module
PIC32MZ_ef_curiosity	PIC32MZ2048		

MIPS M4K/M-class Non-DMA-Enabled Firmware

Firmware	MCU	OS/SDK	Source Code
PIC32MX_RetroBSD	PIC32MX795	RetroBSD	http://retrobsd.org/wiki/doku.php/start
PIC32MZ-BSD-Lite	PIC32MZ2048	LiteBSD	https://github.com/sergev/LiteBSD/wiki

TABLE IV: Sample firmware tested in unit tests. Firmware marked with * were also used in performance tests.

All the sample firmware include the essential routines for running on real devices, such as OS initialization and peripheral (including DMA controller) configuration and operation. As shown in Figure 6, each firmware accesses multiple peripherals (ranging from 4 to 18) and registers (ranging from 9 to 132). Each firmware configures up to 4 DMA streams simultaneously.

We run each unmodified firmware binary with the DICE prototypes supporting ARM and MIPS architectures and collected debug output from QEMU for evaluating DICE's true positive rates and false positive rates when detecting DMA input channels. In this experiment, we fill the detected DMA buffers (as determined by the transfer pointers and transfer sizes) with zeros, instead of meaningful or fuzzer-generated data. This is to make our experiment precise and reproducible.

The result shows that DICE can accurately identify DMA input channels on different architectures, MCU models and OSes. In other words, DICE is accurate, hardware-independent and OS-agnostic.

True positive and false negative analysis: To establish the ground truth, we looked up the DMA configuration information in the MCU data-sheets and verified it using the instrumented emulators. There were 52 firmware-executed (emulator-observed) DMA stream configurations in all the firmware. We use these as the ground truth for evaluating

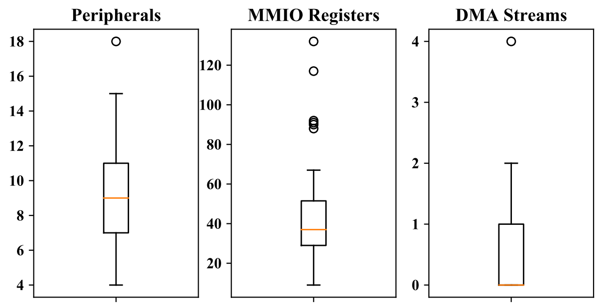

Fig. 6: Distributions of numbers of peripherals, MMIO registers and DMA stream configurations used in sample firmware (outliers marked by ○). All firmware access multiple and different peripherals, which shows their complexity and diversity.

	Observed by emulator (ground truth)	**TP DICE**
True DMA stream configurations	52	45 (87%)
- DMA input channels	37	33 (89%)
- DMA output channels	15	12 (80%)
Buffer accessed (size inferred)	22	21 (95%)
	Observed by emulator (ground truth)	**FP DICE**
False DMA stream configurations	35	0 (0%)
Buffer accessed (size inferred)	6	0 (0%)

TABLE V: Accuracy of DMA stream configuration detection: true positives (upper table) and false positives (lower table). False DMA stream configurations are pointer-like values written to MMIO that do not configure DMA.

Firmware	**P^{2}IM [s]**	**DICE [s]**	**Diff[%]**
ADC PDC	89.0	94.1	5.7
ADC SW DMA	5.3	5.3	0.0
ADC Timer DMA	5.3	5.4	1.9
I2C TxRx DMA	2.8	2.8	0.0
SPI DMAC Slave	17.3	17.7	2.3
SPI Full DMA	28.2	28.9	2.5
SPI Full EX. DMA	13.6	15.1	11.0
SPI Half DMA	18.1	18.7	3.3
USART TxRx DMA	5.4	5.5	1.9
USART Full DMA	20.7	21.6	4.3

TABLE VI: Time needed for firmware execution to reach a fixed point when running on original P^{2}IM (Col. 2) and DICE integrated on top of P^{2}IM (Col. 3). This shows the overhead that DICE adds to P^{2}IM.

true positives and false negatives. As shown in the upper half of Table V, DICE correctly detected 45 out of 52 DMA stream configurations. Among those detected, 33 are DMA input channels and 12 are DMA output channels (the latter is irrelevant to firmware analysis and thus their buffer access/size is omitted). DICE missed 7 DMA stream configurations (false negatives, 4 input and 3 output DMA channels) due to the pattern limitation discussed in §V-B. There were 22 DMA buffers accessed by the firmware during our tests. DICE detected 21 of them and correctly inferred their sizes. The undetected buffer corresponds to one of the seven configurations missed by DICE.

It is worth noting that firmware execution on the emulators never accessed the DMA buffers created by 12 DMA stream configurations correctly identified by DICE. This is due to the limitation of P^{2}IM and the PIC32 emulator. These emulators neither prioritize nor recognize DMA-related code paths in firmware. Without DMA buffers being accessed, DICE cannot identify DMA transfer sizes. We further verified that, in all the cases where the emulators executed the corresponding DMA access code path, DICE successfully identified the DMA transfer size.

False positive analysis: We instrumented the emulators to find pointer-like values written to MMIO during firmware execution. We then used the MCU data-sheets to select those that are unrelated to DMA configurations. We refer to them as "false DMA stream configurations" and use them as the ground truth for evaluating false positives. As shown in the lower half of Table V, among all the firmware tested, the emulators observed 35 pointer-like values written to the MMIO region that do not configure DMA. DICE did not consider any of them as transfer pointers, thus achieving a 0% false positive rate. This is because, for the heuristics to consider a value written to MMIO as a transfer pointer, the value not only needs to be in the narrow RAM range (*i.e.,* pointer-like) but also needs to be accompanied by another transfer pointer written to the adjacent MMIO location. As an example, the TIMER1 counter register of the PIC32MZ2048EF MCU is initialized with the value 255 via an MMIO write operation. This value on the MIPS M-class architecture is a valid RAM address (a pointer), which is allocated to a global variable. DICE observed this value, recognized it as a pointer to RAM. However, since no other pointer was observed to be written to the adjacent MMIO, DICE never considered this MMIO write as part of a DMA configuration. Among the 35 pointer-like values written to MMIO, 6 were dereferenced/accessed by firmware. Since they were not transfer pointers, DICE did not intercept the memory accesses or infer the buffer sizes.

Hypothetically, if firmware reads from a memory address that was falsely identified as a DMA channel, DICE may provide the firmware with analyzer-provided input (*e.g.,* fuzzer-generated data). This may in turn corrupt firmware execution. However, in our experiments, no false positive occurred thanks to the simple yet accurate heuristics.

Runtime overhead: The runtime overhead of DICE is fairly low. It adds only 3.4% on average to the execution time of the sample firmware. The main source of overhead is the instrumentation required to identify the DMA transfer size. This instrumentation affects every LOAD instruction. A secondary source of overhead is the instrumentation capturing the stream configurations. It only affects write operations on the peripheral MMIO area in memory. Therefore, the overhead of DICE is determined by the number of LOAD instructions executed, the number of streams configured, and the size (number of bytes) of the buffers. Table VI illustrates the overhead on 10 firmware selected from the entire set. These firmware were selected because they run smoothly on P^{2}IM and execute all the code paths related to DMA.

B. Fuzz-testing Real Firmware

In this experiment, we demonstrate that DICE can effectively support dynamic analysis on real-world firmware that uses DMA. To this end, we fuzz-tested 7 real-world firmware using DICE integrated with P^2IM. DICE accurately identified DMA input channels on all 7 firmware and found 5 previously unknown bugs (none of them were found by P^2IM alone). DICE supported fuzzing sessions for all firmware, whereas P^2IM alone failed to bootstrap a fuzzing session for 1 firmware. DICE also achieved a much higher code coverage than P^2IM, echoing the importance of DMA emulation and support during dynamic analysis.

Experiment Setup: We selected 7 real-world firmware from different sectors, ranging from industrial IoT to consumer devices. These firmware represent diverse use cases of DMA, such as data signal acquisition without CPU intervention and high throughput data exchange. All these firmware contain OS/system libraries (including scheduler, driver, interrupt service routine) and application logic. As shown in Table VII, they are based on various MCU models and multiple OSes (i.e., FreeRTOS, Mbed OS, bare-metal). We briefly describe below the firmware functionality and security consequences of its bugs:

Modbus: Modbus is a master-slave communication protocol that is widely used in commercial PLC (Programmable Logic Controller) devices. This firmware is a highly-optimized implementation of Modbus slave that uses DMA to receive commands from Modbus master. As PLC devices normally control critical industrial processes, bugs in this type of system can lead to Stuxnet-like [43] attacks and cause physical damage.

Guitar Pedal: This firmware includes the digital signal processing (DSP) routines for creating musical effects in an electric guitar. It also includes a graphical user interface (GUI) for configuring the effects. This firmware is a typical example of mixed-signal (digital and analog) application, which uses DMA to continuously sample analog channels at a fixed rate. Vulnerabilities in this firmware can crash the firmware or produce unexpected sound effects that are harmful to human hearing.

Soldering Station: This is a customized firmware for the KSGER mini soldering station. It includes a PID (Proportional–Integral–Derivative) temperature control routine and a graphical user interface for configuring and operating the device. The firmware uses DMA to read multiple ADC channels continuously in circular mode. Vulnerabilities in this firmware can destroy the heating element and cause injuries to operators.

Stepper Motor: This is the firmware for the stepper motor controller in a CNC (Computer Numerical Control) machine. CNC machines are widely used in 3D printers, drills, lathes, etc. The firmware implements a stepper motor control routine, a command parser, and a proprietary communication protocol. It uses DMA to achieve high communication throughput and control speed. Vulnerabilities in these devices can be exploited to modify the motor speed or bypass the safety checks in the CNC machine.

GPS Receiver: This firmware implements the GPS receiver communication protocol defined by the National Marine Electronics Association (NMEA) [13]. This firmware uses DMA for serial communication. We discuss this firmware as an example in §II. Vulnerabilities in this firmware can be exploited to manipulate navigation, and in turn, control autonomous or human-operated vehicles.

MIDI Synthesizer: This firmware implements the MIDI protocol and controls the synthesizer operation. It processes stereo audio inputs through the WM8778 audio codec and digital signal processing routines, and outputs audio to external amplifiers. It uses DMA to receive and parse MIDI messages with low latency and overhead. Vulnerabilities in this firmware can crash the firmware and produce output signals that can overcharge the amplifiers and damage the electronic circuit of the instrument.

Oscilloscope: This is a minimal oscilloscope that includes a touchscreen as the user interface and supports PC communication for data acquisition. This firmware uses DMA for continuous and fast sampling of electrical signals through ADC. Vulnerabilities in this firmware can crash the system, corrupt the data acquired and present false information to the user.

We use unmodified AFL [45] as our fuzzer (*i.e.,* generating DMA and other firmware input) and fuzz-tested all firmware using DICE on P^2IM. We launched the fuzzer with random seed input and fuzz-tested each firmware for 48 hours. As pointed out by [30], memory corruption errors are less likely to crash the MCU firmware than computer programs, which causes fuzzer to miss some bugs after triggering them. To mitigate this problem, we used the same simple memory error detector (or sanitizer) described in [10]. It grants read+execute permission to Flash, read+write permission to RAM and the peripheral MMIO region, and no permission to the rest of memory space. This simple detector allows for detection of access violations that cross region boundaries, but not those within a memory region. Besides, we implemented a more fine-grained error detector for buffer over/under-flow detection. We inserted red-zones before and after buffers at compile time. Accesses to red-zones will crash the firmware execution. Although red-zones make bugs more visible, they are not required to launch fuzzing. In other words, firmware binary can be fuzz-tested "as is" without re-compilation. We note that an advanced memory sanitizer may help detect more bugs in our experiment, but designing such a sanitizer is out of the scope of this paper.

Fuzzing Statistics: For all tested firmware, DICE was able to automatically and completely identify DMA input channels (transfer pointer + transfer size), and feed fuzzer-generated input to DMA buffers through RAM hooks. We did not observe any falsely identified DMA input channels (i.e., no false positives). On the other hand, P^2IM alone was able to fuzz test only 6 out of 7 firmware. It failed to fuzz test MIDI

Firmware	MCU	OS	Size	Source
Modbus	STM32F303	FreeRTOS	1.3MB	https://github.com/DoHelloWorld/stm32f3_Modbus_Slave_UART-DMA-FreeRTOS
Guitar Pedal	STM32F303	Mbed OS	2.4MB	https://github.com/Guitarman9119/Nucleo_Guitar_Effects_Pedal
Soldering Station	STM32F103	Baremetal	1.4MB	https://github.com/PTDreamer/stm32_soldering_iron_controller
Stepper Motor	STM32F466	Baremetal	1.4MB	https://github.com/omuzychko/StepperHub
GPS Receiver	STM32F103	Baremetal	798KB	https://github.com/MaJerle/GPS_NMEA_parser, https://github.com/MaJerle/STM32_USART_DMA
MIDI Synthesizer	STM32F429	Baremetal	0.7MB	https://github.com/mondaugen/stm32-codec-midi-mmdsp-test
Oscilloscope	STM32F103	Arduino	0.7MB	https://github.com/pingumacpenguin/STM32-O-Scope

TABLE VII: Real-world Firmware fuzz-tested with P^{2}IM and DICE

Synthesizer firmware because the firmware only consumes inputs through DMA input channels that are not supported by P^{2}IM.

As shown in Table VIII, DICE outperforms P^{2}IM on 5 out of 7 firmware in terms of fuzzing coverage. DICE improves the basic block coverage by up to 30.4%, and increase the number of paths triggering new execution patterns by up to 79 times.

The improvement in path coverage is much more significant than basic block coverage, for two reasons. First, a larger number of basic blocks in firmware are executed during the booting process, when MCU hardware and OS are initialized. As no DMA operation is involved during booting, these basic blocks are reachable even without DMA support, which allows P^{2}IM to achieve a fairly high basic block coverage. Second, firmware code is highly reused on MCUs due to constrained device storage. For example, the USART peripheral on the Modbus firmware is used by the console to print messages and by the Modbus protocol to reply to Modbus masters. The USART driver functions are invoked by the console, which operates without using DMA, and the Modbus protocol, which operates through DMA. Supporting DMA does not significantly increase basic block coverage for these driver functions as most of them are also used by non-DMA operations (i.e., console printing). However, without DMA emulation or support, code paths that depend on DMA input (i.e., Modbus protocol stack) can never be reached or tested, despite that these paths may share many basic blocks with other paths unrelated to DMA operations or input.

As evidenced by the result, many firmware contain a great number of DMA-related paths (hence the drastic increase in path coverage under DICE). Such paths cannot be explored by analyzers without using DICE.

Also significantly, DICE improves the Max Depth by up to 500% (on Stepper Motor). This improvement indicates that, with DICE (or generic DMA emulation), dynamic analyzers can now dig much deeper into firmware code, unveiling states and bugs residing at far ends of executions. This result also indicates that, with the ability to directly feed input to DMA buffers, even off-the-shelf fuzzers like AFL (without DMA awareness) can be used for fuzzing firmware relying on DMA input channels.

In terms of fuzzing speed, DICE is slower on 3 out of 7 firmware than P^{2}IM, with 18% as the worst-case slowdown (observed on Stepper Motor). The slower fuzzing speed is not only caused by the overhead of DMA support (discussed in §VII-A), but also the fact that more basic blocks and paths are executed on each fuzzer run thanks to the added DMA support.

As 2 rare cases, fuzzing Soldering Station and Oscilloscope firmware with DMA support turned out to be faster, 5.6% and 92.1% respectively, with slightly lower code coverage. We found that in these cases DMA input channels through ADC allows rapid consumption of input data, which caused the firmware execution to finish much earlier than without DMA support.

Detected New Bugs and Case Study: DICE found 5 unique, previously unknown bugs that P^{2}IM alone (i.e. without DMA emulation) cannot detect (Table IX). We manually examined these bugs (3 in Modbus and 2 in MIDI Synthesizer) and confirmed that it is the DMA support that makes DICE outperforms existing dynamic analysis frameworks, such as P^{2}IM. More specifically, Modbus and the MIDI Synthesizer firmware receive commands from the USART peripheral through DMA. Without DMA support, no command can be received by the firmware. As a result, the command parsing logic and application logic, where all 5 bugs were found, can never be executed.

We verified that all the bugs are real and reproducible on real devices with the same fuzzer-generated input. All the bugs detailed in Table IX are remotely exploitable. They are triggered by the commands that the firmware receives through the USART peripheral via a DMA input channel.

The bugs found in the MIDI Synthesizer firmware (ID 4 and 5) are caused by freeing memory not on the heap. The firmware stores the MIDI messages received through DMA input channels in the buffers that are either statically allocated (as global variables) or dynamically allocated on the heap. When the buffers are allocated on the heap, the firmware invokes free() function to deallocate them after the MIDI messages is processed. In these bugs, the firmware uses global buffers, but still invokes the free() function which is supposed to free only buffers allocated on heap. This causes that memory not on the heap is freed, which may lead to firmware crash and denial-of-service. These bugs cannot be found by P^{2}IM alone because the free() function is only invoked when the MIDI messages received from DMA input channels are processed.

For the Modbus firmware, DICE identified 2 buffer overwrite and 1 buffer overread bugs. The root cause is improper validations of array indexes. Specifically, the firmware uses untrusted input for calculating array indexes, but fails to validate

	Modbus			Guitar Pedal			Soldering St.			Stepper Motor			GPS Receiver			MIDI Synth.			Oscilloscope		
Framework	P^2IM	DICE	Δ%	P^2IM	DICE	Δ%	P^2IM	DICE	Δ%	P^2IM	DICE	Δ%	P^2IM	DICE	Δ%	P^2IM	DICE	Δ%	P^2IM	DICE	Δ%
BBL Cov. [%]	52.6	58.7	11.6	16.9	17.0	0.6	31	31	0	22.3	25.6	14.8	11.5	15.0	30.4	0.0	40.8	N/A	27.3	27.3	0.0
Total Paths	16	1276	7875	3267	3773	15.5	172	166	-3.5	4595	5276	14.8	30	1988	6527	0	588	N/A	618	590	-4.5
Max Depth	2	8	300	4	5	25	3	3	0	2	12	500	5	6	20	0	3	N/A	5	4	-20
Speed [run/s]	41.6	41.0	-1.4	3.8	3.8	0	17.9	18.9	5.6	22.2	18.2	-18.0	49.4	48.9	-1.0	0	59.9	N/A	0.76	1.46	92.1

TABLE VIII: Statistics of fuzz-testing real firmware using DICE on P^2IM. P^2IM was unable to fuzz test MIDI Synthesizer firmware which solely consumes input through DMA input channels.

Firmware	Bug ID	Bug type	Security consequences
Modbus	1, 2	Buffer overwrite	Corrupt data structure with attacker controlled values.
	3	Buffer overread	Information leakage.
MIDI Synthesizer	4, 5	Free of memory not on the Heap	Firmware crash, denial-of-service.

TABLE IX: New bugs found by DICE in Modbus and MIDI Synthesizer firmware. None of them can be found by P^2IM alone.

the computed indexes or ensure the indexes are referencing valid positions within the arrays. Modbus receives commands from a shared fieldbus in a typical PLC device setup. Therefore, any malicious/compromised device connected to the fieldbus can exploit these bugs by sending crafted commands. These bugs allow an attacker to corrupt data structures or retrieve secrets stored in the firmware memory, e.g. critical parameters of the PLC control routine.

Listing 1 shows the code snippet of Bug #1. `startAddr` is calculated from `modbusRxTxBuffer[]`, which is the DMA buffer that holds untrusted input (Line 265). The firmware checks if `startAddr` is within the valid range of array `modbusMemory[]` (Line 266), and then uses it as an index for array access (Line 270). The input validation at Line 266 is wrong (the correct check should be `startAddr >= MODBUS_SLAVE_REGISTERS_NUM`), which causes buffer overwrite at Line 270.

```
265 uint16_t startAddr = modbusRxTxBuffer[2] << 8 |
        modbusRxTxBuffer[3];
266 if(startAddr > MODBUS_SLAVE_REGISTERS_NUM) //
        improper input validation
267   answerLen = modbusSlaveErrorSet(0x02);
268 else
269 {
270   modbusMemory[startAddr] = modbusRxTxBuffer[4] << 8
        | modbusRxTxBuffer[5]; // buffer overwrite
271   answerLen = modbusRxCount;
272 }
```

Listing 1: Code snippet of Bug #1

We also investigated the potential reasons for our experiment not finding bugs in the other five firmware. In general, fuzz-testing firmware with DICE on P^2IM faces the open challenges as with other existing tools, such as the lack of error detectors/sanitizers for MCU, limited ability to solve complex state machines and path constraints, etc., which are out of the scope for this paper. We also identified the following reasons specific to DMA.

First, some firmware using DMA can quickly drain fuzzer input, without going deep into the code. Such firmware tends to allocate large DMA buffers for high-throughput data transfers. For example, Stepper Motor uses 2 buffers of 4 KB for receiving and transmitting data. However, AFL prefers to generate short input sequences to achieve better fuzzing performance.

Second, some DMA input channels do not directly influence firmware control flow, but DICE still treats them as “risky” channels and lets the fuzzer generate and mutate inputs for them. For example, Soldering Station, Guitar Pedal and Oscilloscope continuously sample analog input using DMA and performs mathematical calculations on the sampled inputs. These DMA inputs can rarely change firmware execution paths or trigger bugs. However, due to the design requirement of being peripheral-agnostic, DICE cannot detect or exclude such fuzzing-unworthy DMA input channels, and thus, ends up spending too much time on them, instead of focusing on other fuzzing-worthy DMA input channels.

VIII. Discussion

A. Location of transfer descriptor

DICE identifies the DMA input channels based on the assumption that transfer descriptors are always written to DMA controller through MMIO operations. However, in some rare cases, transfer descriptors are stored in RAM, which are not supported by DICE. We admit it a limitation of DICE. To measure how prevalent the transfer descriptors are stored in RAM, we surveyed the complete STM32 MCU portfolio of ST Microelectronics, a top-five MCU vendor according to [19] and the most popular MCU vendor in terms of the number of Github repositories (19,870 unique entries by Nov. 2019). The product lines we analyzed include ultra-low-power, mainstream, and high-performance families. Our analysis showed that all MCUs (983 in total) store transfer descriptors in the peripheral memory area, while the STM32H7 product line (59 MCUs) also allows storing transfer descriptors in RAM. To conclude, only 6% of STM32 MCUs can optionally store transfer descriptors in RAM, and therefore, it is acceptable for DICE not to handle this rare case.

B. DMA Buffer Size Identification

Unlike source and destination, buffer size cannot be reliably identified from a DMA stream configuration event. To solve this, DICE adopts a conservative heuristic for buffer size identification. The heuristic, which gradually expands the perceived DMA buffers at memory read that falls right after the buffer boundary, may produce a smaller-than-actual size upon memory reads that are not consecutive in space. This inaccuracy, although possible in theory, is not observed in our evaluation. Moreover, when the firmware executes for long enough, DICE may progressively identify the correct buffer size. Therefore, it is reasonable to use such a conservative heuristic which trades identification accuracy for the firmware stability (when a wrongly identified DMA buffer byte is read, the firmware can crash).

C. Architecture beyond ARM and MIPS

DICE can be applied to other architectures that meet three requirements: (R1) the architecture uses designated memory regions for peripherals (MMIO), Flash and RAM; (R2) DMA streams configurations are written to the peripheral region via MMIO; (R3) DMA transfers follow the life cycle depicted in Figure 4.

We analyzed RISC-V, the increasingly popular architecture used in MCUs. We confirmed that RISC-V meets these requirements, and therefore, is compatible with DICE. Specifically, we studied the data-sheets of the RISC-V GD32VF103 MCU [35]. The MCU uses separate memory regions for peripherals, RAM and Flash (0x40000000-0x5003FFFF, 0x20000000-0x20017FFF, and 0x08000000-0x0801FFFF respectively), thus meeting R1. It also writes DMA streams configurations via MMIO to peripherals and meet R2. The MCU obeys the DMA life cycle illustrated in Figure 4 and meet R3. This result demonstrates that DICE is generically applicable to at least three different architectures, namely ARM, MIPS, and RISC-V.

D. Devices beyond MCUs

DICE is designed to support DMA in firmware analysis for MCU devices. It solves multiple challenges, especially, the hardware and software diversity of MCUs. Other platforms such as desktop and mobile devices use similar DMA transfer descriptors and follow the same DMA life cycle as MCUs. However, those more powerful platforms frequently store transfer descriptors in RAM, rather than in DMA controller registers (MMIO), for flexibility reasons (e.g., to support complex concatenated DMA transfers). As we discussed in §V, RAM-stored transfer descriptors are not supported by DICE. Therefore, DICE cannot be directly used for emulating DMA on platforms other than MCUs.

Existing work such as PeriScope [39] can handle DMA on Linux-based platforms during dynamic analysis. PeriScope instruments Linux kernel DMA APIs to monitor the creation and destroy of DMA input channels and to manipulate DMA input. PeriScope is OS-specific. Unlike DICE, PeriScope is not applicable to MCUs, which have highly diverse OS and firmware, and therefore, pose unique challenges for DMA emulation.

E. Integration with Other Firmware Analyzers

The design of DICE allows for easy integration with various firmware analyzers, providing them with the capability of analyzing firmware that use DMA. As a demonstration, we integrated DICE with Avatar2 [29], a flexible dynamic firmware analysis framework. The implementation is only 240 lines of Python code. This integration allows Avatar2 to recognize and manipulate DMA data when it was read by the firmware, and in turn, to analyze firmware that uses DMA for input.

Furthermore, we integrate DICE with Symbion [14] (an extension to the Angr [38] framework). This integration allows DMA-aware concolic execution on MCU firmware. Specifically, we used DICE and Avatar2 to identify the DMA buffers used by a firmware as the firmware runs on a real development board. Upon reaching an interesting point of analysis, Symbion moves the concrete state of this execution to Angr. Thanks to DICE, Angr can now recognize and symbolize the DMA buffers in the concrete state, achieving a more precise and comprehensive concolic execution.

F. Peripherals beyond DMA controller

Many merits of DICE, such as hardware-independent and firmware compatibility, are attributed to our abstraction of the DMA input channel. We believe that for other peripherals (e.g., counters and comparators), which have well-defined functionalities and identifiable configurations like DMA controllers do, can be abstracted in a similar way. We admit that identifying if and how a peripheral can be abstracted for dynamic analysis purposes, in a hardware-independent and firmware-compatible way, is an interesting research topic that we would like to pursue in the future.

IX. Related Work

A. Dynamic Firmware Analysis

Multiple existing works tackled the challenging problem of dynamic firmware analysis. They are divided into hardware-in-the-loop emulation approaches and full emulation approaches by whether real devices are required in the process of dynamic analysis. Avatar [44] proposed a novel hardware-in-the-loop emulation mechanism, which forwards peripheral operations to a real device while executing the firmware in the emulator. It conducted concolic execution for MCU firmware. Surrogates [22] improved the performance of peripheral operation forwarding by customized hardware. [30] fuzz-tested simple programs with artificially-implanted bugs using Avatar, which demonstrated that memory corruption vulnerabilities are much less likely to crash on MCU than on desktop. Avatar2 [29] extended Avatar with the record and replay capability for the forwarded peripheral operations. Charm [42] fuzz-tested Android device drivers by a hardware-in-the-loop emulator that uses a similar forwarding technique with Avatar. Prospect [21] forwarded peripheral operations made through syscalls, the abstraction of which is not available for MCU devices. [20] combined runtime program state approximation with peripheral access caching to facilitate dynamic analysis.

Hardware-in-the-loop emulation approaches suffer from poor performance and scalability due to the slow forwarding speed and one-to-one binding between emulator instances and real devices. Several recent works addressed this by removing the need for real devices through full emulation. Pretender [15] generated approximated peripheral models from the peripheral operations that are forwarded to the real device by Avatar

[44]. With the model, it successfully executed and fuzzed-tested several simple firmware with manually-injected vulnerabilities without using any real device. P^{2}IM [10] completely removed the usage of real devices by automatically modeling the processor-peripheral interfaces while emulating the firmware. It fuzz-tested several real-world firmware of typical embedded applications and found real bugs. HALucinator [2] adopts a high-level emulation-based approach which replaces Hardware Abstraction Layer (HAL) functions with manually-crafted handlers by library matching on binary. It fuzz-tested network stack, file system, serial port and PLC, and found real bugs. PartEmu [17] fuzz-tested ARM TrustZone software stack by a hybrid approach of replacing software components with stubs and modeling peripheral hardware with manually crafted register value patterns. Various works dynamically analyzed Linux-based firmware by full emulation [6], [8], [34]. Those firmware are more similar to general-purpose desktop software than truly embedded firmware. Emulators have much better support for Linux-based firmware, which uses less diverse peripherals than MCU firmware. However, none of these works, either hardware-in-the-loop emulation or full emulation, were able to dynamically analyze MCU firmware using DMA without requiring any source code. The major obstacle is the un-emulated DMA controllers, which are vendor-specific and possibly proprietary. Our work can extend both hardware-in-the-loop (Avatar) and full emulation (P^{2}IM [10]) mechanism with DMA support and conduct dynamic analysis. Although HALucinator [2] can analyze DMA-enabled firmware (because its high-level emulation totally removes DMA operations), their approach (specifically, the library matching component) requires source code of HAL. Neither can HALucinator find bugs in firmware components that are replaced by the high-level emulation, such as drivers for DMA controller and other peripherals. DICE adopts a completely different approach which identifies DMA input channels from unmodified firmware. Therefore, DICE is able to find bugs in the whole firmware stack, without requiring any source code or manually created handlers.

B. DMA Attacks & Analyses

Various works revealed attacks enabled by DMA. To name a few, PCILeech [11] revealed that malicious peripherals with DMA capability (e.g., PCIe peripherals) can access/modify arbitrary physical memory addresses and gain full control over the victim computer if IOMMU is not enabled (IOMMU enables virtual memory for I/O devices). Thunderclap [23] further demonstrated that DMA attacks are still feasible even with IOMMU enforced. These attacks, however, are not feasible on MCUs because MCU peripherals normally do not have DMA capability, and DMA is conducted through a dedicated DMA controller which is part of the System-on-Chip (SoC) and considered trusted. PeriScope [39] identified a compromised peripheral device (e.g., Wi-Fi chip) can attack kernel device drivers by sending malicious input through DMA, and therefore, fuzz-tested the DMA channels. It identifies DMA channels by instrumenting Linux kernel APIs, which however is not applicable to bare-metal MCU firmware.

X. Conclusion

We presented a survey showing the prevalence and diverse usages of DMA on MCU-based embedded devices. We highlighted the importance of supporting DMA-enabled peripherals during dynamic firmware analysis. To address existing firmware analyzers' inability to test DMA-enabled firmware, we designed and built DICE, a drop-in solution that enables analyzer-generic and hardware-independent emulation of DMA input channels. By identifying and observing DMA configurations and accesses by firmware during emulated execution, DICE detects DMA input channels that are dynamically created by firmware. It also dynamically infers the locations and sizes of memory buffers used as DMA transfer destinations. Without requiring any human assistance or firmware source code, DICE allows firmware analyzers to run and test DMA-related code, and in turn, find bugs or vulnerabilities in firmware that otherwise cannot be reached or triggered.

We integrated DICE into P^{2}IM (for ARM Cortex-M) and a MIPS PIC32 emulator. We evaluated DICE using 83 sample firmware and 7 real-world firmware. Its runtime overhead is low (3.4%) and its emulation accuracy is very high (89% true positive rate and 0% false positive rate). When used for fuzzing the real-world firmware, DICE increased code path coverage by as much as 79X. Moreover, it helps detect 5 unique, previously unreported bugs, which would not have been found without the generic and automatic DMA emulation.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments. This project was supported by the National Science Foundation (Grant#: CNS-1748334), the Office of Naval Research (Grant#: N00014-18-1-2043), and the Army Research Office (Grant#: W911NF-18-1-0093). Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Nrf52832 datasheet. https://infocenter.nordicsemi.com/pdf/nRF52832_PS_v1.4.pdf. Accessed: Sep 2019.

[2] Halucinator: Firmware re-hosting through abstraction layer emulation. In *29th USENIX Security Symposium*, 2020.

[3] C. Gordon Bell, Allen Newell, and Daniel P. Siewiorek. Structural levels of the pdp-8. http://digitalcollections.library.cmu.edu/awweb/awarchive?type=file&item=356992, 1978. Accessed: Apr 2020.

[4] Fabrice Bellar. Qemu features/softmmu. https://wiki.qemu.org/Features/SoftMMU. Accessed: Apr 2020.

[5] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[6] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

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

[8] Andrei Costin, Apostolis Zarras, and Aurélien Francillon. Automated dynamic firmware analysis at scale: a case study on embedded web interfaces. In *ACM Asia Conference on Computer and Communications Security*, 2016.

[9] Geovanny Di Sirio. ChibiOS. http://chibios.org, 2017. Accessed: Sep 2019.

[10] Bo Feng, Alejandro Mera, and Long Lu. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium*, 2020.

[11] U. Frisk. Direct memory attack the kernel. In *Proceedings of DEFCON'24*, 2016.

[12] Gartner. Gartner says 5.8 billion enterprise and automotive iot endpoints will be in use in 2020. https://www.gartner.com/en/newsroom/press-releases/2019-08-29-gartner-says-5-8-billion-enterprise-and-automotive-io. Accessed: Nov 2019.

[13] GPSinformation.org. NMEA data. https://www.gpsinformation.org/dale/nmea.htm. Accessed: Sep 2019.

[14] Fabio Gritti, Lorenzo Fontana, Eric Gustafson, Fabio Pagani, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. Symbion: Interleaving symbolic with concrete execution. In *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*, June 2020.

[15] Eric Gustafson, Marius Muench, Chad Spensky, Nilo Redini, Aravind Machiry, Yanick Fratantonio, Davide Balzarotti, Aurelien Francillon, Yung Ryn Choe, Christophe Kruegel, et al. Toward the analysis of embedded firmware through automated re-hosting. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID) 2019)*, 2019.

[16] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah D. Hester, Jacob Sorber, and David Kotz. Application memory isolation on ultra-low-power mcus. In *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 127–132, 2018.

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

[18] Jesse Hertz and Tim Newsham. Triforceafl. https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2016/june/project-triforce-run-afl-on-everything/. Accessed: Sep 2019.

[19] IC Insights. The mcclean report 2017 - april update. https://www.eenewsanalog.com/news/ma-moves-alter-mcu-vendor-ranking-0. Accessed: Sep 2019.

[20] Markus Kammerstetter, Daniel Burian, and Wolfgang 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.

[21] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: peripheral proxying supported embedded code testing. In *ACM Symposium on Information, Computer and Communications Security*, 2014.

[22] Karl Koscher, Tadayoshi Kohno, and David Molnar. Surrogates: Enabling near-real-time dynamic analyses of embedded systems. In *WOOT*, 2015.

[23] A Theodore Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon W Moore, and Robert NM Watson. Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[24] Microchip. Microchip product portfolio march 2019. https://www.microchip.com/ParamChartSearch/chart.aspx?branchID=30063, 2019. Accessed: March 2019.

[25] Microchip. Quick reference guide 16bit microcontrollers. http://ww1.microchip.com/downloads/en/DeviceDoc/30010109F.pdf, 2019. Accessed: March 2019.

[26] Microchip. Quick reference guide 32bit microcontrollers. http://ww1.microchip.com/downloads/en/DeviceDoc/60001455D.pdf, 2019. Accessed: March 2019.

[27] Microchip. Quick reference guide 8bit microcontrollers. http://ww1.microchip.com/downloads/en/DeviceDoc/30009630M.pdf, 2019. Accessed: March 2019.

[28] Valasek Chris Miller Charlie. Remote exploitation of an unaltered passenger vehicle. http://illmatics.com/RemoteCarHacking.pdf. Accessed: Nov 2019.

[29] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A multi-target orchestration platform. In *BAR*, 2018.

[30] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In *Network and Distributed System Security Symposium (NDSS)*, 2018.

[31] NIST. CVE-2019-6496. https://nvd.nist.gov/vuln/detail/CVE-2019-6496, 2019. Accessed: April 2020.

[32] Koopman Phil. A case study of toyota unintended acceleration and software safety. https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf, 2014. Accessed: November 2019.

[33] Google Project Zero. Over the air: Exploiting broadcom's wi-fi stack. https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html, 2017. Accessed: November 2019.

[34] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. Symdrive: Testing drivers without devices. In *OSDI*, 2012.

[35] GigaDevice Semiconductor. Gd32vf103 user manual. http://gd32mcu.21ic.com/data/documents/shujushouce/GD32VF103_User_Manual_EN_V1.2.pdf, 2019. Accessed: Nov 2019.

[36] Vakulenko Serge. Qemu for mips pic32. https://github.com/sergev/qemu/wiki. Accessed: April 2020.

[37] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla. Swatt: software-based attestation for embedded devices. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, pages 272–282, May 2004.

[38] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[39] 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 *Network and Distributed System Security Symposium (NDSS)*, 2019.

[40] STmicroelectronics. Stm32f4 reference manual. https://www.st.com/resource/en/reference_manual/dm00031020.pdf, 2019. Accessed: Nov 2019.

[41] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. Oat: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

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

[43] Wikipedia. Stuxnet. https://en.wikipedia.org/wiki/Stuxnet, 2010. Accessed: Sep 2019.

[44] Jonas Zaddach, Luca Bruno, Aurelien Francillon, and Davide Balzarotti. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[45] Michal Zalewski. american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: Sep 2019.