GDMA: Fully Automated DMA Rehosting via Iterative Type Overlays

Tobias Scharnowski, Simeon Hoffmann, Moritz Bley, Simon Wörner, Daniel Klischies[1],
Felix Buchmann, Nils Ole Tippenhauer, Thorsten Holz, and Marius Muench[2]

CISPA Helmholtz Center for Information Security
[1]*Ruhr-Universität Bochum,* [2]*University of Birmingham*

Abstract

Embedded systems are the critical interface between the physical and the digital world, where security breaches can lead to significant harm. In recent years, rehosting has proven to be an effective method for dynamic security testing of embedded systems. However, existing approaches largely ignore the automated rehosting of Direct Memory Access (DMA), a key mechanism for receiving untrusted data. The only fully automated DMA rehosting approach considers just one out of six common DMA mechanisms, leaving significant gaps in the security analysis of firmware.

In this work, we introduce GDMA, a comprehensive solution for fully automated DMA rehosting. GDMA successfully emulates *all six* DMA configuration mechanisms by analyzing emulation traces to identify the two critical DMA usage steps: DMA configuration and DMA buffer usage. More specifically, it first collects type information on MMIO registers that consistently behave like pointers. We organize this information in *type trees*, which capture relationships between MMIO registers and the memory regions they reference. GDMA then overlays and merges these trees to iteratively distill a DMA configuration. By applying this configuration in a generic DMA peripheral, GDMA enables effective testing of DMA-dependent firmware. We evaluate GDMA on a total of 114 firmware images. Compared to the state of the art, GDMA is the first to successfully emulate *all* samples of the state-of-the-art benchmark, reaching 3x the DMA mechanism coverage. We also introduce a fully reproducible data set to systematically evaluate DMA rehosting of all six mechanisms. GDMA successfully rehosts all of these, which is a factor of 6x compared to existing methods. Finally, we evaluate GDMA on various DMA-enabled firmware and discover 6 new bugs with 6 assigned CVEs following a coordinated disclosure.

1 Introduction

Firmware is the fundamental level of code that connects the hardware components and the higher-level software applications in embedded systems. Ensuring its security is critical, as vulnerabilities at this level can compromise the entire system, leading to unauthorized access and similar security incidents. With the growing connectivity and integration of embedded systems, firmware security has become increasingly important. Among the various techniques used to improve firmware security [30, 46], fuzz testing (*fuzzing* for short) stands out as particularly effective. By systematically providing semi-random inputs to the firmware under test, fuzzing uncovers software faults often missed by other testing methods.

To implement firmware fuzzing effectively, it is essential to execute the firmware and provide diverse inputs in a controlled environment [49]. *Rehosting*–the process of running firmware outside its native hardware context in an emulator–makes this possible efficiently and at scale [14]. Rehosting eliminates the need for physical hardware during testing and enables a dynamic analysis across numerous firmware samples. This capability has made rehosting an essential method for dynamic security testing, enabling extensive testing that is not feasible on physical devices [8, 9, 15, 31, 32, 47, 48].

Current rehosting approaches rely on techniques such as hardware abstraction layer (HAL) replacements [10, 35] and peripheral emulation [50] to emulate the hardware environment that the firmware expects. One major challenge remains largely unaddressed: the automated handling of Direct Memory Access (DMA) operations. DMA descriptors instruct a DMA peripheral to transfer data to or from system memory without continuous intervention of the processor. This enables an efficient data transfer, but makes emulation more complex. Recent rehosting efforts, both source-based [20] and specification-based [50], have largely avoided automated DMA handling due to its complexity. Instead, they rely on manual, case-by-case implementations that require domain expertise and do not scale. An accurate modeling of DMA is challenging, especially given that DMA operations are indirect and often involve a non-obvious communication between hardware and memory. Overcoming this limitation is essential for scalable and automated rehosting solutions.

A notable attempt to address this challenge is DICE [27], which introduced a simple heuristic to emulate the most ba-

sic forms of DMA automatically. DICE assumes fixed, predictable memory-mapped I/O (MMIO) register layouts to reduce false positives during emulation. While a step forward, this approach is limited in scope and applicability. We systematically analyze technical reference manuals of popular microcontrollers and find that there are six popular variants of DMA configurations; five of them are *not* covered by DICE. More specifically, our analysis reveals that hardware developers often use diverse MMIO register layouts, making DICE's assumptions unreliable. Furthermore, the method ignores a full class of DMA configuration mechanisms: RAM-based DMA descriptors. As a result, we find that the robust identification of DMA configurations is still an unsolved challenge.

In this paper, we address the problem of fully generic DMA rehosting by proposing GDMA, a method to enable automated modeling of DMA behavior. By iteratively refining a DMA behavior model based on memory access traces, our method enables the handling of diverse DMA mechanisms *without* requiring source code or hardware specifications. More specifically, we propose a four-step process to achieve fully generic DMA rehosting by taking advantage of the inherent properties of DMA: the configuration of buffer locations and the (pseudo-)uninitialized nature of receive buffers. First, we isolate pointer-like MMIO registers to avoid unsolicited pointer writes by identifying only registers that consistently point to RAM as potential DMA descriptor candidates. Next, we collect type information from MMIO and RAM access traces in the form of many *type trees*, which represent data structures at RAM locations pointed to by MMIO registers. We overlay and merge these trees iteratively to distill a common type per MMIO register. This ensures consistency of type information across firmware executions and avoids false positives due to unexpected RAM contents. Based on the distilled type tree, we synthesize a DMA configuration by matching the type layout against the six common variants and by checking for potential DMA buffer accesses in the leaf nodes of the distilled type tree. Finally, we extend the rehosting system by a generic, configuration-driven DMA peripheral to handle DMA by applying the automatically synthesized configurations. These specify the MMIO register addresses and DMA descriptor structures and enable the generic DMA peripheral to inject fuzzing data. Our iterative approach ensures that DMA behaviors are discovered and refined incrementally.

A practical challenge for the evaluation of our approach is the lack of a comprehensive data set to measure the DMA capabilities of different approaches. To address this gap, we have developed a data set of fully reproducible, standardized firmware images specifically designed to test the DMA capabilities of rehosting techniques. This benchmarking not only facilitates the evaluation of our approach but also provides a valuable resource for future research in this area.

We evaluate GDMA on a total of 114 firmware images. In an empirical evaluation, we show that GDMA is the first solution to successfully achieve full coverage of DMA rehosting on an existing benchmark, which is 3x the coverage of DMA mechanisms of the state of the art. On the new benchmark that covers all six DMA mechanisms, our evaluation indicates that GDMA even achieves a factor of 6x in successfully rehosting DMA mechanisms. Furthermore, our evaluation shows that GDMA allows a fuzzer to successfully cover complex DMA firmware behavior, leading to higher code coverage (an increase between 3.5% and 152.6% compared to the non-DMA-enabled baseline) and the identification of 6 previously unknown bugs that were fixed by the vendors.

In summary, our key contributions are as follows:

- We present the design and implementation of GDMA, a generic DMA rehosting approach capable of identifying and automatically modeling both MMIO-based and RAM-based DMA descriptors.
- We address an important problem affecting the firmware rehosting research community by providing a set of fully reproducible, standardized firmware images designed to test the DMA capabilities of rehosting techniques.
- We perform a comprehensive evaluation demonstrating that GDMA is the first fully automated DMA modeling technique to correctly model *all* of the current state-of-the-art data set and to provide coverage for 6x the DMA mechanisms compared to the state of the art.
- We demonstrate the security impact of GDMA by identifying and disclosing 6 previously unknown bugs, leading to the assignment of 6 CVEs in the core network stacks of well-known real-time operating systems.

2 Technical Background

2.1 Security Assessment of Firmware

Embedded Systems Firmware. Embedded systems consist of microcontroller units (MCUs) with purpose-built firmware. Firmware interacts with the outside world through peripherals such as sensors, actuators, and network interfaces. Due to the resource-constrained nature of such systems, embedded firmware code executes bare metal and interacts with its peripherals directly. These interactions take three common forms: Memory-Mapped IO (MMIO), interrupts, and Direct Memory Access (DMA). Via MMIO, the CPU configures and queries peripherals by accessing a dedicated memory region. Peripherals raise interrupts to notify firmware, e. g., about the arrival of a network packet. In contrast to MMIO, DMA allows peripherals to transfer data (such as network packets) to main memory asynchronously without involving the CPU, thereby avoiding the expensive process of reading MMIO data from peripherals byte-by-byte.

Firmware Security Testing. As embedded systems are the backbone of critical infrastructure systems, their resilience against manipulation by adversaries is paramount. In particular, embedded firmware provided by third parties must be assessed for security vulnerabilities. Fuzzing is an effective

dynamic testing technique to perform such analyses [8,23,35] but is met with several challenges for embedded firmware [30]. Fuzzing relies on a high throughput of testing samples, but fuzzing firmware on embedded hardware is typically slow.

Rehosting. A promising approach to overcome slow hardware for embedded firmware fuzzing is to host it on virtual hardware (called *rehosting*). Rehosting needs to handle the diverse peripherals of the embedded system to allow the firmware to execute normally and receive outside input via MMIO and DMA. While DMA rehosting has largely been unaddressed (see Section 3), automated solutions exist to model MMIO peripherals [9, 15, 31, 32, 48]. These solutions allow the fuzzing of MMIO-based firmware in a scalable way.

2.2 Direct Memory Access (DMA)

MCUs implement DMA support via controller peripherals, which are present in most 32-bit MCUs [27]. These controllers expose MMIO registers to set up asynchronous data transfers between peripherals, flash, and RAM. To configure transfers, DMA controllers allow for the population of *DMA descriptors*. A DMA descriptor is the smallest unit of DMA configuration. It contains information for a given transfer, including the transfer source and destination (common sources and destinations are RAM buffers or a peripheral data register). In the following, we refer to transfer destination buffers that reside in RAM interchangeably as *receive buffers*. After configuring a DMA descriptor, the firmware activates the transfer, prompting the DMA controller to start moving data. Once the DMA controller has finished the transfer, it either notifies the firmware via an interrupt or the firmware explicitly polls the status by reading the DMA controller's status register(s). The firmware subsequently processes the contents of the receive buffers or initiates further transfers.

Scatter-Gather DMA. A complex form of DMA configuration is *Scatter-Gather DMA*. Instead of using a single DMA descriptor that holds a source-destination pair, multiple descriptors are used to either populate multiple destinations from a single source (scatter) or to write multiple sources to a single destination (gather). For scatter-gather operations, DMA controllers use structures that contain multiple DMA descriptors such as *descriptor chains* or *descriptor pointer tables*, which we explain in Section 3.2.

3 DMA Rehosting: A Solved Problem?

DMA is a core mechanism through which firmware receives data from the outside world. To effectively fuzz test firmware, it is necessary to rehost DMA transfers (i. e., filling DMA buffers with fuzzing input). As we will show in Section 6.5, being able to test DMA-enabled parts of the firmware input processing allows for achieving more code coverage and identifying previously unknown vulnerabilities.

Table 1: Categorization of DMA handling in rehosting works.

Approach Type	Tool Name
Automatic	DICE [27]
Hardware	GDBFuzz [12] Shift [28] uAFL [22]
Manual HAL Hooks	HALucinator [10] MetaEmu [7] ParaRehosting [21] SafireFuzz [35]
Manual Annotations	CO3 [23] EmberIO [13] MultiFuzz [9] Perry [20] Fuzzware [31] SEmu [50]

In this section, we first review the state of the art of DMA rehosting. While DMA is mentioned regularly in previous works, we found that upon closer inspection, currently *only one fully automated approach* exists. Next, we systematize the different mechanisms by which DMA peripherals can be configured to understand the extent to which DMA rehosting is currently addressed. We identify *six prevalent mechanisms* of configuring DMA, of which *only the simplest one* is addressed by the current state of the art. Finally, we analyze why only the simplest form of DMA has been addressed and deduce the challenges involved in fully automatically rehosting complex DMA configuration mechanisms.

3.1 State of the Art of DMA Rehosting

As shown in Table 1, most rehosting systems either off-load DMA to physical hardware, leverage hardware abstraction layers to replace DMA, or rely on manual annotations to implement custom peripherals. Two works are considered fully automated by related work: SEmu [50] and DICE [27].

SEmu parses MCU specification files to extract condition-action rules that allow for high-fidelity MMIO emulation. However, as the authors point out, and as is also evident in the code base, manual, platform-specific patches are required to support DMA [51]. Consequently, SEmu is *not* a fully automated approach for DMA rehosting. We thus exclude SEmu from the evaluation.

DICE, the only fully automated approach, identifies DMA transfers via an MMIO write pattern. Specifically, DICE considers *two addresses* written to *two directly adjacent MMIO registers* as the source and destination addresses of a DMA transfer. If this exact pattern matches, DICE assumes DMA. Otherwise, DICE assumes no DMA. In their evaluation, the authors state that their approach does not work on all tar-

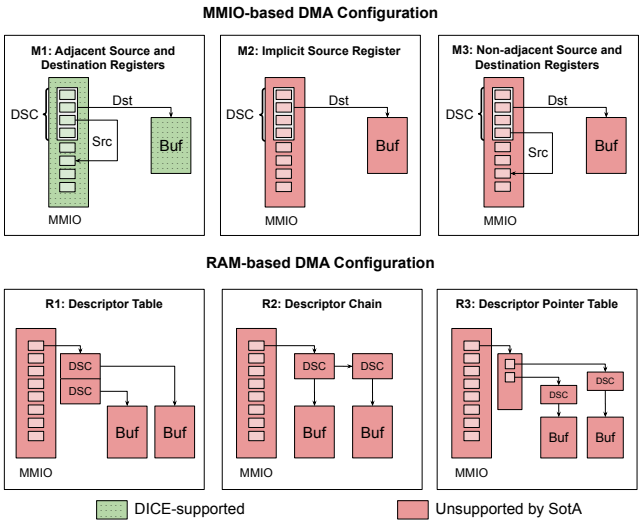

Figure 1: Prevalent DMA configuration mechanisms. We identify six mechanisms across two categories: MMIO-based DMA configurations reside solely in MMIO registers. For RAM-based DMA configurations, DMA descriptors (DSC) reside in RAM and are referred to by an MMIO register.

gets [27]. They detail one unsupported mechanism in the paper: Two platforms of the data set (nRF51 and nRF52) only use a single MMIO pointer register. As a second adjacent MMIO pointer is not present, DICE misses this mechanism.

Summary. Our review of the current state of the art shows that existing approaches to DMA modeling are largely manual. Additionally, the only fully automated approach, DICE, addresses only *one* specific configuration pattern.

3.2 DMA Configuration Mechanisms

To better understand the scope of currently unaddressed challenges in DMA rehosting, we analyzed the reference manuals of 10 popular MCU vendors, 6 of which were already analyzed by DICE. The first insight that we draw from our analysis is that there are *six prevalent mechanisms of DMA configurations*, which we term M1-M3 and R1-R3. Figure 1 provides an overview of these mechanisms. We categorize them in two sets with three members each, *MMIO-based configurations* and *RAM-based configurations*.

MMIO-based DMA configuration. In MMIO-based DMA configurations, DMA descriptors (denoted as DSC in Figure 1) are embedded directly into the MMIO region of the DMA controller. In this case, MMIO registers hold all information about a transfer, including its source and destination addresses. We found three mechanisms of MMIO-based configurations. The first mechanism (M1) is the mechanism that DICE captures: The source and destination address are placed into directly adjacent MMIO registers. For the second mechanism (M2), only a single pointer is configured via an MMIO write, while the peripheral address is implied. This is the case that the DICE paper mentions as a limitation. The third mechanism of MMIO-based configuration (M3) uses a source and destination pointer in MMIO, but these pointers are not adjacent, thus also missed by DICE, albeit not explicitly mentioned in the paper.

RAM-based DMA configuration. As opposed to MMIO-based configurations, RAM-based DMA configurations store descriptors in RAM, not in MMIO registers. They only write the *address of RAM-based metadata* into an MMIO register instead of the transfer's metadata itself. This introduces a *layer of indirection* between the (obvious) MMIO accesses and the information required to perform a DMA transfer. We categorize RAM-based configurations into three mechanisms. The first RAM-based mechanism, R1, configures its DMA descriptors (DSC) in a table layout. Each DMA descriptor contains all the metadata required for a DMA transfer. The second RAM mechanism, R2, uses a chain of descriptors. Here, in addition to its usual transfer-related metadata, each descriptor refers to the next descriptor in the chain. R3 also uses a table, but each table entry contains a pointer to a descriptor instead of the descriptor itself.

3.3 Challenges of DMA Modeling

As current fully automated DMA rehosting techniques address only the simplest of the six identified DMA mechanisms, the core challenge of automated DMA rehosting remains unaddressed: the *robust identification of DMA configurations*.

To better understand previous design choices, it may be instructive to reflect on the challenges faced by rehosting DMA when we compare them to the challenges of MMIO rehosting, which has recently seen significant progress. Unlike MMIO, accesses to DMA buffers cannot be detected by observing a well-known region of memory. As DMA buffers are largely free to reside anywhere in system RAM, their accesses blend in with regular accesses to system memory. Great care has to be taken not to confuse accesses to regular RAM with an access to a DMA buffer.

If a DMA transfer is accidentally detected for regular RAM, fuzzing input may be written to an unexpected location. As a result, the state of the firmware, such as global variables, heap buffers, or the stack, would be corrupted. We assume that this *risk of false positives* leads existing work to place very specific constraints on what is considered a DMA descriptor.

In an effort to model DMA holistically, we identify three challenges for modeling DMA in a fully automated manner:

C-1 **Spurious MMIO-written pointer values**: A typical MCU contains hundreds of MMIO registers with diverse semantics. Consequently, a set of arbitrary-looking values is written to each MMIO register. Some of these values will appear pointer-like, which introduces a *high risk of false positives*, especially for one-shot configuration detection techniques.

Current approach: Limit the rehosting scope and only address configuration mechanism M1, which inherently places tight restrictions on adjacent MMIO registers.

C-2 **Variety of DMA configuration mechanisms**: DMA can be configured in many different ways. DMA has to be rehosted without introducing potential false positives.
Current approach: Leave the variety unaddressed.

C-3 **Spurious RAM contents**: In addition to pointer-like writes to MMIO registers, once we attempt to identify DMA descriptors in system RAM, the RAM contents (e. g., those that are referenced by these MMIO registers) may disguise as potential DMA descriptors (mechanisms R1-R3). Thus, a one-shot attempt at detecting RAM-based DMA descriptors based on arbitrary RAM contents introduces another *source of false positives*.
Current approach: Limit the scope to MMIO-based DMA mechanisms and do not address RAM-based ones.

Conclusion. Our review of existing firmware rehosting, including HAL-based approaches and MMIO modeling techniques that mention DMA has shown that fully automated DMA remains largely unaddressed. Based on our analysis of technical reference manuals, we introduced a *systematization of six prevalent mechanisms* of DMA configurations. We found that current rehosting work only considers one of these variants and leaves a full class of DMA (RAM-based configurations) unaddressed. Our findings demonstrate that the lack of DMA support in prior work on firmware rehosting severely limits scalability for firmware security testing. A comprehensive novel approach that addresses the previously neglected challenges is urgently needed to cover more than just the simplest mechanisms of DMA configuration.

4 Generic DMA Modeling with GDMA

We now introduce GDMA, our novel method for *fully automated* DMA rehosting to enable fuzzing of DMA-enabled firmware for *all six mechanisms* of DMA configurations that we discussed in Section 3.2.

4.1 Binary-only Prerequisites

To be as generically applicable and as scalable as possible, we design our approach to require minimal information and *no manual intervention*.

As we use no HAL abstractions, source code, or hardware specifications, we inherit only the minimal assumptions of generic MMIO-based rehosting systems (an approximate memory layout, ISA emulation, and binary firmware). To *avoid other sources of information* that other rehosting works rely upon, we follow a binary-only approach. This means:

1. No access to firmware source code.
2. No access to hardware reference manuals.

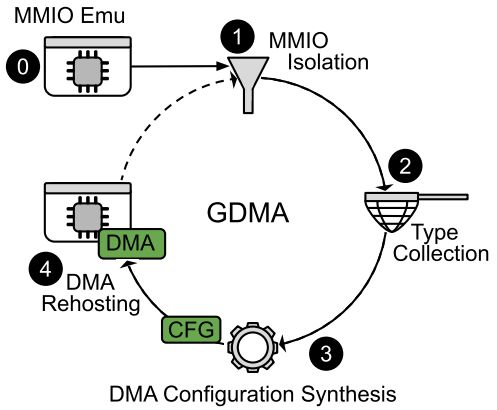

Figure 2: High-level design overview of GDMA.

3. No access to firmware symbols.
4. No knowledge about the firmware HAL.

Requiring access to any of this additional information would inherently restrict the applicability and scalability of our approach. Access to (parts of) the firmware source code or hardware reference manuals is often not available, symbols are not contained in binary firmware images, and relying on hardware abstraction layers would introduce manual effort and expert knowledge.

4.2 Design Overview

We design our DMA modeling on the simple idea of identifying a connection between (a) a potential DMA descriptor and (b) potential DMA buffers. To make this actionable, we use the insight that DMA has two distinct inherent properties: the necessary *configuration of DMA buffer locations* and the *(pseudo-)uninitialized nature of DMA receive buffers*.

First, firmware needs to configure a DMA peripheral to make it aware of DMA buffers (see Section 2.2). While the specific mechanisms vary, these configurations are based on RAM addresses that are written to MMIO registers (see Section 3.2). To identify the DMA mechanisms used by the target firmware, we collect type information on the data in RAM that pointer-like MMIO registers refer to. We represent this type information via what we term *type trees*. We describe type trees in more detail and with an example in Section 4.4. To detect DMA configurations, we match these type trees to the DMA configuration mechanisms described in Section 3.2. To ensure consistent type information and to avoid false positive detections, we iteratively overlay and merge these type trees for each MMIO register.

Second, a distinctive feature of DMA receive buffers is how they are accessed. As receive buffers contain external input, the firmware will, *after configuring a DMA transfer*, wait for the transfer to complete and use its DMA receive buffer in an *uninitialized manner*. More specifically, the firmware

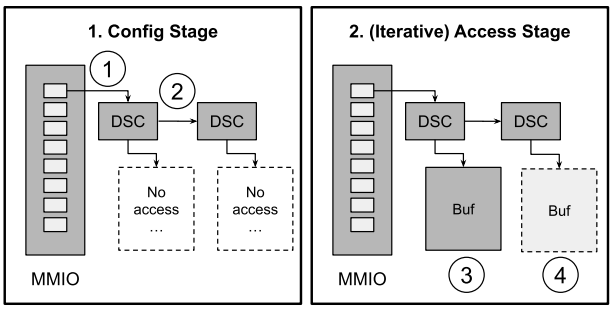

Figure 3: GDMA overview of the iterative modeling approach by the example of a type R2 transfer.

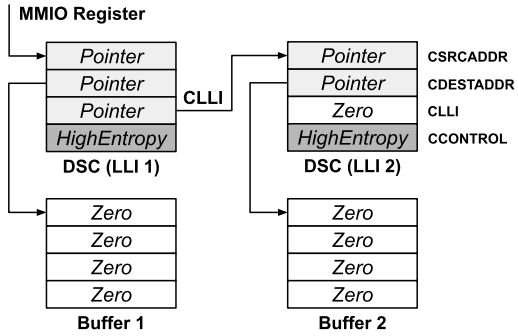

Figure 4: Example type tree for the LPC1837 R2 scatter-gather DMA configuration. It uses a descriptor chain with a DMA buffer and control information, as well as a pointer to the next element per descriptor.

will read from the DMA buffer before writing to it, as it would otherwise overwrite the received contents. We use this property to identify DMA buffers in leaf nodes of type trees.

Overall, we propose the following four-step process, which we also visualize in Figure 2:

1. **Isolation of pointer-like MMIO registers.** In this step, we address challenge **C-1** of spurious pointer-like MMIO register writes. We consider only MMIO registers that consistently represent pointers to RAM as potential DMA descriptor candidates (Section 4.3).
2. **Collection of raw type information.** Based on traces of MMIO and RAM accesses, we gather type trees of the respective data in RAM which each consistently pointer-like MMIO register refers to (Section 4.4). Type trees form the basis for flexibly checking the adherence to DMA configuration types (challenge **C-2**).
3. **Distillation of DMA models.** To address challenge **C-3**, we then infer a common type tree for each consistently pointer-like MMIO register. To achieve this, we iteratively overlay and merge type trees to distill common type information (Section 4.5). Based on this type information, we synthesize concrete models of DMA configuration for the firmware target (Section 4.6).
4. **DMA rehosting based on DMA models.** To perform DMA transfers during fuzzing, we extend the rehosting system with a generic DMA peripheral. The peripheral parses a synthesized DMA model configuration and writes fuzzing input into the identified DMA receive buffers according to the detected structure (Section 5.2).

4.3 GDMA Iterative Modeling Approach

We observe that a DMA transfer is an iterative process. As shown in Figure 3, the transfer is conceptually split into a configuration phase and an (iterative) access phase. First, the firmware sets up pointers (①) to DMA descriptors (②). After configuring the DMA transfer and waiting for it to complete, the firmware will access the first DMA buffer (③). Only in case the (application-specific) contents of the first buffer pass validity checks can the second buffer be accessed (④).

Consequently, the DMA mechanisms appear only over time. The fuzzer has to first discover the DMA configuration stage before the use of DMA starts becoming visible. The firmware will access the first buffer only after the fuzzer indicates a DMA transfer completion (for example, via an injected interrupt). Initially, when the DMA transfer is not yet emulated, the buffer contents will remain static. As these contents will likely not pass any validation (such as for the adherence to an application-specific header structure), the firmware may not access the second buffer. Only after the DMA transfer for the first buffer is modeled will the fuzzer be able to pass the content validation, allowing the second buffer to be accessed. Thus, we design GDMA to follow this iterative flow during modeling. Conceptually, the modeling approach uses two core characteristics of modern rehosting environments:

1. Their ability to gradually explore firmware functionality. We integrate our design by providing DMA functionality *as its use is discovered*.
2. The ability of the underlying fuzzing engine to produce diverse firmware behavior. We use this to *distill consistent behavior* for DMA configuration synthesis.

GDMA uses the rehosting environment to uncover diverse, *MMIO-based* behaviors of the firmware under test. More specifically, we base our analysis on RAM and MMIO access traces from the input queue of the already integrated fuzzing engine. The inputs that the fuzzer retains within the queue represent the set of interesting behaviors that the fuzzing engine has been able to trigger.

For these inputs, we generate traces of the accesses to MMIO ranges, as well as to firmware RAM. We *isolate* MMIO registers that consistently have pointer-like values written to them. We discard any registers with an instance of a non-pointer write and consider the remaining ones *potential DMA configuration MMIO registers*.

4.4 Descriptor Type Tree Collection

After isolating candidates for DMA configuration MMIO registers, we now gather type information based on the RAM location pointed to by each MMIO register.

To be able to later perform DMA model detection, we base our type inference approach on what we refer to as *type trees*. A type tree contains *nodes*, where each node contains *fields*. Each field has one of three types. We differentiate between the three types `Zero`, `Pointer`, and `HighEntropy`. While a `Pointer` represents a value that is a valid address, `HighEntropy` represents a non-zero value that does not represent a valid address. For each RAM `Pointer` field in a node, we associate a child node. Consequently, each node without a `Pointer` field is a leaf node of the *type tree*. The root element of a type tree itself is a `Pointer` to a node.

Example (type tree): As a practical example of a type tree, we consider the scatter-gather DMA mechanism implemented on the LPC1837 platform [36]. The mechanism uses a chain of Linked List Items (LLI). Each LLI contains four fields: the transfer source address (CSRCADDR), the transfer destination address (CDESTADDR), a pointer to the next LLI (CLLI), and control information about the size of the transfer (CCONTROL). The linked list in the data structure makes this mechanism a type R2 descriptor (see Section 3.2). Each LLI configures one DMA transfer. In a receive operation, data will be copied from an MMIO register (source) to a DMA buffer residing in RAM (destination). Thus, CSRCADDR will be set to the address of a peripheral data MMIO register. Figure 4 shows how this mechanism translates to a type tree. As CSRCADDR holds a mapped address, GDMA assigns a `Pointer` type to the field. CDESTADDR and CLI point to the DMA receive buffer and the next LLI, respectively. As such, two `Pointer` types are assigned. As these are RAM addresses, additional *nodes* are created for each. Assuming that the values in the DMA buffer are zeroed, its node contains fields of type `Zero` (on the left in Figure 4). The type node referred to by CLLI mirrors the structure of its parent node, with the difference that a zero in the link of the last LLI indicates the end of the chain (on the right in Figure 4).

4.5 Common Data Type Inference

With the type trees collected, the goal of the next step is to distill common type information. We achieve this by *overlaying and merging* the type trees across interesting inputs. A merged type tree represents the common type information that is consistent across all firmware behaviors for a given MMIO register. This type information forms the basis for our DMA model synthesis, which we describe in Section 4.6.

Figure 5 illustrates how we generate the merged type tree in multiple stages. Initially, we merge the type trees that we collect from each MMIO pointer write within a given trace. We then perform the same merging process across

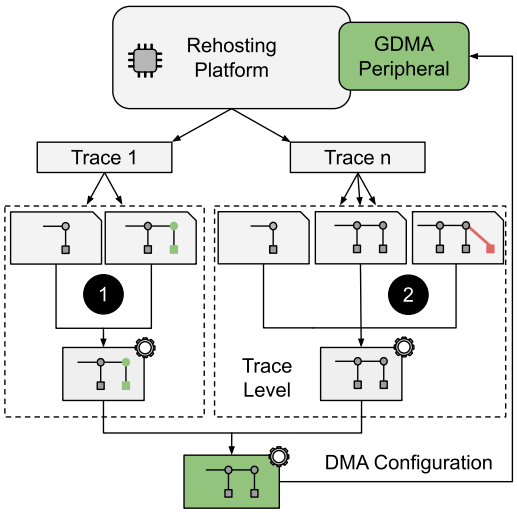

Figure 5: Overview of our descriptor type inference approach.

interesting inputs. Using the distillation process, we address the following two challenges. First, we need to *extend* the type tree via iteratively discovered DMA interactions. For example, no type information was present before the fuzzer discovered the DMA configuration stage within the firmware (see ① in Figure 3). Second, values in RAM may lead to instances of `Pointer`-typed fields (see Challenge **C-3**), which we aim to *prune* in a process of *data type merging*. Data type merging is performed via the type transitions shown in Appendix A. The intuition behind these transitions is to find the least specific, common denominator of types observed.

Example (extension): For an instance of *extending* a type, we re-visit the example of the LPC1837 DMA descriptor chain. Consider the scenario where, during DMA configuration, the firmware first sets only a single DMA buffer in the chain. In this case, the *CLLI* field of the first descriptor (LLI 1) would be set to zero (instead of `Pointer` for the third field of LLI 1 in Figure 4), so the field is initially assigned type `Zero`. If the firmware at some point chooses to configure a DMA transfer with two linked list items, the CLLI field of the first item will contain the address of the second item. Thus, the field is assigned type `Pointer` in the type tree of the later MMIO access. When the two types `Zero` and `Pointer` are merged, this will result in the common type `Pointer`. This is illustrated as ❶ in Figure 5.

Example (pruning): For an instance of *pruning* of the type tree, we consider the control field CCONTROL of an LPC1837 LLI. Here, the initial value of CCONTROL had a usual arbitrary bit assignment with a resulting type of `HighEntropy` (the fourth field of LLI 1 in Figure 4). We now consider an instance of a spurious value where the bit pattern of CCONTROL happens to represent a valid RAM address. The resulting type tree would then contain the spurious type `Pointer` for this field. When merged with the existing

type `HighEntropy`, the two types are merged into a common `HighEntropy`. This eliminates the spurious type in the first merging stage. This is visualized as ❷ in Figure 5.

4.6 Synthesizing DMA Configurations

Based on the merged type information, we synthesize DMA configurations by matching a given type tree layout to one of the six DMA mechanisms that we described in Section 3.2.

DMA buffer identification. As indicated in Section 4.2, we identify DMA buffers by their *(pseudo-)uninitialized nature*. To this end, we consider a RAM location a DMA buffer candidate in case it adheres to a set of requirements regarding additional per-node metadata in the type tree: its *initialization state* and its *access state*. Regarding the initialization state, the RAM location is considered a DMA candidate in case it is either uninitialized or initialized with a pattern of repeating values since the DMA transfer has been configured. Regarding the access state, we check the buffer to be read from before it is written to (if it is written to at all).

Configuration synthesis. For the MMIO-based mechanisms M1-M3, we perform the *MMIO register isolation* and buffer identification steps. For the RAM-based mechanisms R1-R3, we match the structure of the type tree against the respective DMA configuration mechanisms. For a potential descriptor (see DSC in Figure 1), we search for a combination of a possible source MMIO address, a possible destination DMA buffer RAM address, and an optional link pointer. For optional link pointers, we recursively check the structure of the referenced *nodes* to match the same source address, destination address, and link layout.

Once a consistent DMA mechanism has been found, we generate a model that specifies the data structure of the DMA mechanism. We apply this configuration to rehost the DMA transfer and feed fuzzing input via our configuration-based DMA peripheral, which we outline in Section 5.2.

5 Implementation

Our prototype of GDMA consists of a fully automated end-to-end implementation of DMA detection, peripheral modeling, and DMA transfer handling. We integrated GDMA in Fuzzware [31] as an extension by adding the `--dma` DMA auto-modeling flag to the `fuzzware pipeline` command. The GDMA implementation is divided into two major parts: First, we implemented a DMA modeling engine in Rust, which synthesizes models of DMA descriptors, as outlined in Section 4. Second, to apply these DMA descriptors during rehosting, we implemented a generic, configuration-based DMA peripheral in Fuzzware. To configure the peripheral, we created the Generic DMA Descriptor Definition (GDDD). The format of GDDD closely mimics the type tree structure that we introduced in Section 4.4 and allows the DMA peripheral to perform DMA transfers accordingly.

5.1 Type Inference

The type inference process of GDMA begins with a regular Fuzzware run, where Fuzzware models MMIO interactions and provides fuzzer input as usual. We extend the pipeline component of Fuzzware by having it generate MMIO and RAM access traces for interesting inputs. GDMA processes these traces to incrementally perform the data type inference and model synthesis as described in Section 4. Raw type information is collected once new interesting inputs are available in the queue of the fuzzer. The implementation performs the distillation of the type information into DMA models every 10 minutes. Once a model is synthesized, the Fuzzware pipeline is instructed to update its emulator configuration by integrating the generated GDDD configuration. This triggers the GDMA peripheral to start rehosting DMA transfers, as described in the next section.

5.2 GDMA Peripheral

We extended Fuzzware with a generic DMA peripheral that abstracts over DMA descriptor models by accepting GDDD model configurations. Our peripheral is integrated into the snapshotting mechanism of Fuzzware. Our peripheral leverages the memory hooking ability of Fuzzware to monitor MMIO register writes. When a write to a designated DMA register is detected, GDMA parses the corresponding DMA descriptor according to the provided GDDD format. For the resulting DMA receive buffers, the peripheral hooks read accesses to the buffer and supplies fuzzing input upon access. For the identification of DMA buffer sizes, we adopt the linear buffer growth mechanism introduced by DICE [27].

6 Evaluation

We evaluate GDMA along the following research questions:
RQ1 Does GDMA support all 6 DMA mechanisms?
RQ2 What is the false positive rate of GDMA?
RQ3 How does GDMA perform against the state of the art?
RQ4 Can GDMA discover unknown vulnerabilities?

6.1 Experimental Setup

To answer our research questions, we first design a diverse data set comprising all six DMA mechanisms and analyze the DMA support provided by DICE and GDMA (**RQ1**). We exclude SEmu [50] from our evaluation as its DMA support is based on a manual implementation of DMA behavior [51] (see also Section 3.1). Then, we verify that GDMA does not falsely detect DMA transactions in firmware (**RQ2**). To this end, we first evaluate whether GDMA falsely detects DMA transfers in an established unit test suite from the rehosting space (Section 6.3). Second, we validate that no spurious GDMA transfers are detected in DMA-enabled firmware (Section 6.2 and

Table 2: DMA mechanisms covered by the existing DICE data set and our new data set. The top half of the platforms were already part of the DICE data set.

Platform	DICE						GDMA					
DMA mechanism	M1	M2	M3	R1	R2	R3	M1	M2	M3	R1	R2	R3
STMicro STM32F	●						●					
Nuvoton NUC123	●						●					
Atmel SAM3X	●		◐		○		●		●		●	
NXP LPC1837	●				○		●				●	
Nordic nRF52		◐						●				
Freescale MK64F			○		○				●		●	
Renesas RA4W1	◑					○	●					●
TI CC13				○						●		
Infineon PSoC6				○						●		
SiLabs EFM32				○						●		

●: Mechanism contained in data set, mechanism supported
◐: Mechanism contained in data set, mechanism not supported
◑: Mechanism not in data set, mechanism supported
○: Mechanism not in data set, mechanism not supported
Blank: Hardware does not use the mechanism

Section 6.4). To evaluate **RQ3**, we compare GDMA against DICE, the only existing fully automated tool for DMA emulation. Our experiments also evaluate against the respective non-DMA baselines to account for the different underlying rehosting engines (i.e., Fuzzware and P2IM, Section 6.4). We then narrow the focus and run fuzzing campaigns where we compare GDMA against its non-DMA baseline, Fuzzware, and assess its bug-finding capabilities (**RQ4**).

Time Budget & Hardware Setup. Unless otherwise noted, we repeated each experiment 10 times on one core per target for 24 hours, as recommended by Klees et al. [18]. We used two Intel Xeon Gold 5320 CPUs (26 physical cores and 2.20GHz each), 256 GB of RAM, and SSD storage. All experiments were executed on Ubuntu 22.04. For all our fuzzing campaigns, we used the default fuzzing seed inputs provided by the open-source release of the respective tool.

6.2 Support for DMA Mechanisms

As no current data set captures all six DMA mechanisms, we introduce a new, comprehensive data set to evaluate **RQ1**.

Designing a Diverse DMA data set. So far, DICE [27] presented the most comprehensive data set to evaluate the efficacy of DMA rehosting approaches, consisting of 83 firmware samples. However, despite its large size, the data set is not designed to cover various DMA configuration mechanisms. Instead, most DMA-enabled samples feature M1 descriptors (with a strong bias towards STM32). RAM-based DMA configuration mechanisms (R1-R3) are not included in the data set. Due to these factors, we introduce a more varied and streamlined DMA data set. We build upon platforms already targeted by DICE and extend them by platforms of the popular vendors Texas Instruments (TI), Renesas, Infineon, and Silicon Labs.

Table 2 shows the DMA configuration mechanisms used by the platforms of the DICE data set and our extended data set. We designed our data set based on the following goals:

1. *Reproducibility*: We provide the firmware source code and docker containers to reproduce our binaries. Thus, all patches to the targets are made transparent and new changes can be made with ease.
2. *Full DMA mechanism coverage*: For each platform and its supported DMA mechanism, we include a sample that uses this particular mechanism.
3. *Conciseness*: We avoid redundancy and bias by not including multiple samples for the same combination of platform and DMA mechanism.
4. *Coverage-based ground truth*: To facilitate an automated and broadly applicable evaluation, we design the passing or failing of each test case to be determined by a simple firmware code coverage check.

Diverse DMA Data Set Composition. The data set consists of 15 samples in total, covering all 6 types of DMA and 10 different hardware platforms. For each vendor, we created one firmware sample for each DMA mechanism that the vendor's platform supports. We chose one hardware platform per vendor, as we observed that vendors reuse DMA mechanisms across a device family. For example, both the nRF51 and nRF52 series by Nordic Semiconductor reuse EasyDMA. The 10 covered vendors account for 85% of the worldwide MCU market share [39]. We base the firmware images themselves on sample applications of the corresponding vendor software development kits (SDKs). These SDKs contain the board support packages, real-time operating systems, and embedded network stacks that real-world products also build upon. The benchmark functionality is inspired by the "password" samples from the authors of Fuzzware [31]. Each sample receives data only via a DMA-enabled serial communication (e. g., via UART, SPI, or a similar peripheral). This data is compared to a magic value ("Password") byte-by-byte. A sample with contiguous DMA matches the password linearly as only a single DMA buffer is involved. A sample with non-contiguous DMA involves two buffers and is not matched linearly. Instead, the buffers are checked in turns. Initially, the first buffer needs to start with the value 'P'. If this condition is fulfilled, the first byte of the second buffer is checked for the value 'a'. We implemented this turn-taking mechanism to reduce the impact of the underlying fuzzer on the DMA buffer discovery.

This sample layout provides several benefits: First, each rehosting platform, with or without DMA support, can execute these samples. This provides a diverse set of firmware for future testing. Second, we can accurately measure the fuzzing progress: If the fuzzer discovers a correct character, it reaches a new basic block. Consequently, by curating a list of these milestones, we can quickly evaluate how far the fuzzer

progressed through password discovery and confidently assert whether DMA was successfully emulated. In total, as also shown in Table 2, our data set doubles the number of vendors and DMA mechanisms compared to DICE.

Evaluation. To answer **RQ1**, we evaluate GDMA on the introduced firmware samples. We mark a contiguous sample as passed if at least one password character is discovered. For samples with two buffers, we mark the sample as passed if at least one character per buffer is correctly discovered. Note that the discovery of the entire password greatly depends on the individual sample and the fuzzer's performance. We also evaluate these samples with Fuzzware [31] as a baseline.

The results are shown in Table 3. As expected, Fuzzware does not cover any password characters in any firmware due to its lack of support for automatic DMA detection. Indeed, Fuzzware reaches the checking functions, but the configured DMA buffers do not contain any data and thus do not match the expected password character.

DICE inherits P2IM's [15] limitation of requiring an AFL forkserver integration in its targets. As we strive to provide a broad data set for future use, we do not integrate custom AFL forkservers into our firmware targets. Instead of running the targets in DICE, we use a conservative theoretical evaluation of DICE support. If the DMA configuration writes two adjacent pointers to MMIO, the DICE heuristic matches, and we consider the emulation successful. Note that this is an upper bound: As we will see in Section 6.4, P2IM limitations can lead to undetected or misinterpreted DMA configurations.

DICE supports two target platforms fully and three more target platforms partially. The fully supported platforms are STM32 and Nuvoton NUC123, which both feature the DMA mechanism M1. The Renesas RA4W1, Microchip SAM3X, and NXP LPC1837 targets are partially supported. All three implement the DMA configuration mechanism M1 for single-source, single-target, contiguous transmissions and one of the RAM-based types R1-R3 (see Table 2). Notably, regarding **RQ2**, GDMA also did not detect any additional or misclassified DMA configuration mechanisms. Across the 10 platforms, GDMA successfully discovered password characters in all (contiguous and two-buffer) setups. The tool correctly identifies DMA behavior and emulates all six DMA mechanisms. Consequently, GDMA supports all 6 DMA configuration mechanisms (**RQ1**).

6.3 False Positive Analysis

As the second step of our evaluation, we evaluate the effectiveness of GDMA to not only detect DMA mechanisms but also to avoid detecting DMA mechanisms where DMA is not actually intended by the firmware (**RQ2**). Note that a false-positive analysis in terms of misclassifications for DMA-enabled firmware is included in the previous evaluation of our data set (see Section 6.2) as well as our analysis of the results of the DICE data set (see Appendix B for details on misclassifications on the DICE data set). To this end, we evaluate GDMA on all 44 unit tests introduced by P2IM [15], the MMIO-based rehosting system that underpins DICE. These unit tests do not utilize DMA mechanisms. Instead, these tests are designed to test different MMIO-based interactions with hardware peripherals.

Table 3: Results of DMA mechanism sample set passing.

Platform	Type	No-DMA	DICE	GDMA
SAM3X	M1	✗	✓	✓
	M3	✗	✗	✓
	R2	✗	✗	✓
STM32F103	M1	✗	✓	✓
LPC1837	M1	✗	✓	✓
	R2	✗	✗	✓
nRF52832	M2	✗	✗	✓
NUC123	M1	✗	✓	✓
MK64F	M3	✗	✗	✓
	R2	✗	✗	✓
CC1311P3	R1	✗	✗	✓
PSoC6	R1	✗	✗	✓
EFM32LG	R1	✗	✗	✓
RA4W1	M1	✗	✓	✓
	R3	✗	✗	✓
DMA Mechanisms		0/6	1/6	**6/6**
Platforms		0/10	2/10	**10/10**
Samples		0/15	5/15	**15/15**

We evaluate all 44 firmware images in GDMA and find that GDMA, just like DICE, identifies zero false-positive DMA configurations in all 44 unit tests. This shows the efficacy of the filtering mechanism of GDMA (**RQ2**).

6.4 Comparison with the State of the Art

We address **RQ3** and expand on **RQ2** by evaluating GDMA against DICE, which introduces two different evaluation data sets: a unit test set and a set of firmware samples used as fuzzing targets. We evaluate GDMA on both data sets to test the ability of GDMA to rehost DMA in a large set of firmware.

DICE Unit Tests. DICE introduces a unit test set comprised of firmware for the ARM and MIPS architectures, both with and without DMA. To evaluate if GDMA supports all targets that DICE supports, we transfer the ARM samples into a data set. This results in 33 firmware images from 5 different vendors. According to the authors, "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." As a ground truth, we manually in-

Table 4: Summary of DICE and GDMA DMA rehosting success rates on DICE unit tests.

Behavior	DICE		GDMA	
✓	15/33	45.5%	31/33	93.9%
(✓)	10/33	30.3%	2/33	6.1%
(✗)	1/33	3%	0/33	0%
✗	7/33	21.2%	0/33	0%

✓ Correct
(✓) Fuzzer limitation (DMA usage not reached)
(✗) Wrong transfer direction (read instead of transmit)
✗ Undetected descriptor (DICE heuristic mismatch)

spect the firmware binaries and annotate DMA configuration metadata. This ground truth consists of the buffer address and the MMIO address that refers to the DMA buffer. We publish this metadata alongside our data set.

We deem a DMA classification correct if the modeled configuration matches the ground truth. If the underlying fuzzer reaches the configuration and access of the given DMA buffer, but no DMA configuration is detected, or the configuration does not match the ground truth, we label this a misclassification. We run each unit test in both DICE and GDMA and show the results in Table 4. GDMA successfully identifies the correct DMA configuration in each case where the underlying fuzzer is able to cover the configuration and use of the DMA buffer (94% of the unit tests). DICE introduces two sources of misclassifications. First, the DICE heuristics do not apply to two out of the three DMA configuration mechanisms contained in the data set (M2 and M3), leading to no detected DMA. Second, as DICE differentiates between receive and transmit buffers by whether the buffer is read from, one transmission buffer is misclassified as a receive buffer. This results in 8 misclassifications, while GDMA does not introduce any misclassifications. We provide a detailed analysis in Appendix B. Overall, regarding **RQ2**, our experiment shows that GDMA does not introduce any misclassifications or false positives in the DICE data set. It also is the first solution to enable DMA rehosting for the full DICE data set.

DICE Fuzzing Tests. Another set of tests introduced by DICE aims to measure the coverage improvement of fuzzing achieved by DMA-enabled rehosting. While we run these tests with both GDMA and DICE, we note that a direct comparison of code coverage is not fair due to the different underlying fuzzers and rehosting systems. Thus, we mainly aim to ensure that for samples where DICE has been shown to improve code coverage over its non-DMA baseline, GDMA is able to do the same. For this, we report the relative coverage improvement in regard to the respective baseline and refer the interested reader to Appendix C for additional coverage details.

During fuzzing, GDMA identified correct DMA configurations in all cases. As shown in Table 5, GDMA consistently reaches coverage benefits over its non-DMA baseline Fuzzware, ranging from 0.1% to 109.6%. DICE's performance benefits over P2IM lie between 0% and 30.1%. Note that P2IM is known to be unable to emulate the `MIDI Synthesizer` target. Furthermore, we omit the `Oscilloscope` target because it is dysfunctional [11].

Table 5: Coverage improvements introduced by DMA rehosting on the DICE fuzzing targets. The table shows the improvements of DICE and GDMA over their respective no-DMA baselines. Absolute numbers are shown in Table 8.

	Δ DICE	Δ GDMA
GPS Receiver	30.1%	109.6%
Guitar Pedal	0.5%	0.1%
MIDI Synthesizer	— *	51.3%
Modbus	16.4%	16.9%
Soldering Station	0%	4.4%
Stepper Motor	2.2%	9.9%

* Not supported by baseline

In conclusion, for every sample in which DICE detects DMA and shows improved coverage over its baseline, GDMA is able to do the same. Furthermore, GDMA's coverage delta surpasses that of DICE in most cases (**RQ3**).

6.5 Fuzzing DMA-enabled Firmware

To assess **RQ4**, we run fuzzing campaigns against additional firmware samples utilizing the less explored DMA mechanisms (M2-R3). We deliberately exclude targets with DMA mechanism M1, as GDMA's performance on M1 targets is proven in Section 6.4. The resulting 10 firmware images cover a wide range of functionalities, such as network and bluetooth stacks and a variety of parsers of common data types, such as JSON and X.509 certificates. We perform the same experiment with GDMA's no-DMA baseline Fuzzware.

The results are shown in Figure 6. We plot the median, the 25th, and the 75th percentile of the 10 runs for both GDMA and Fuzzware. GDMA outperforms Fuzzware on all 10 targets. The average coverage improvement per target ranges between 3.5% and 152.6%. To determine whether the coverage improvements enabled by GDMA over Fuzzware are statistically significant, we use a bootstrap-based t-test as recommended by Schloegel et al. [33] and Vargha and Delaney's $\hat{A}_{12}$ metric to measure effect size [42]. The results show that for all samples, the coverage improvement is significant (at $p < 0.05$) and, due to GDMA always enabling a higher coverage than Fuzzware, a strong effect size of 1.00. We observe that Fuzzware converges after at most 12 hours on all targets, while GDMA continues to find new coverage. This indicates that Fuzzware reaches the point in the firmware where DMA is used, but further progress is impossible due to the lack of DMA support. GDMA, however, explores the firmware fur-

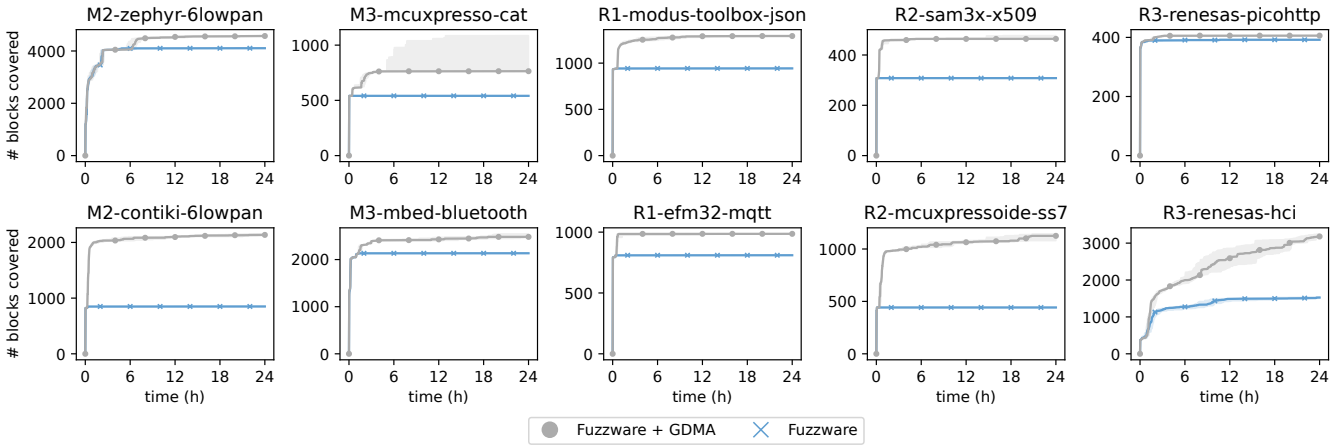

Figure 6: Performance of GDMA on new, complex targets. The plots show the median, 25, and 75 percentile over 10 runs.

ther, as it is able to synthesize a correct DMA configuration for all 10 targets. This allows GDMA to explore the parts of the firmware that depend on DMA input. Differences in improvements stem from the requirements of different parsers. Indeed, a manual inspection of the `R3-renesas-picohttp` sample shows that while GDMA provides DMA input for the firmware to parse, the fuzzer is unable to progress far into the HTTP parsing process, which requires several magic byte strings. Similarly, targets with a large coverage improvement (e. g., `R2-mcuxpressoide-ss7`) employ a parser with fewer magic bytes, enabling the fuzzer to cover more parsing logic.

Found vulnerabilities. During our fuzzing campaigns, GDMA detected multiple vulnerabilities not found by Fuzzware. These vulnerabilities do not require a malicious or malfunctioning peripheral and can be triggered via regular network inputs. This demonstrates the benefits of DMA-enabled rehosting. In total, we found 6 different bugs in the underlying—and well-tested—embedded operating systems. The vulnerabilities include out-of-bounds writes, integer overflows, and incorrect error handling. We reported all bugs to the affected vendors. We provide an overview in Table 9.

We conclude that by rehosting all six DMA configuration mechanisms, GDMA greatly increases fuzzing coverage on DMA-enabled targets over its non-DMA-enabled baseline, leading to the discovery of new vulnerabilities (**RQ4**).

6.6 Bug Case Study: CVE-2023-48229

To exemplify the types of vulnerabilities GDMA can discover, we discuss CVE-2023-48229 as a case study. We discovered this vulnerability in the IEEE 802.15.4 (LR-WPAN) stack of Contiki-NG for nRF52 devices while working on GDMA.

Contiki-NG uses the proprietary nRF Radio HAL, which in turn leverages Nordic's easyDMA feature to asynchronously write contents of incoming radio packets into RAM. Nordic's easyDMA is an example of an M2 configuration mechanism: The position of the corresponding MMIO registers implies its peripheral address, so only one pointer has to be explicitly specified. Contiki-NG sets up the radio HAL to write incoming packages into a static *rx_buf* structure, which contains the IEEE 802.15.4 packet's physical header (PHR), the MAC Protocol Data Unit (MPDU), and a status bit to indicate whether a full packet was received. A firmware can set up Contiki-NG to either poll for incoming packets by checking this status bit or by receiving interrupts upon transfer completion.

```
static int read_frame(void *buf, unsigned short bufsize)
{
  int payload_len;
  /* [...] */

  payload_len = rx_buf.phr - FCS_LEN; // FCS_LEN is 2

  // Return if rx_buf.phr < 5 or rx_buf.phr > 127
  if(phr_is_valid(rx_buf.phr) == false) {
    LOG_DBG("Incorrect length: %d\n", payload_len);
    rx_buf_clear();
    enter_rx();
    return 0;
  }

  memcpy(buf, rx_buf.mpdu, payload_len);
  /* [...] */
}
```

Listing 1: Annotated Source Code for CVE-2023-48229.

The *rx_buf* structure is then polled inside the *read_frame()* function, as shown in Listing 1. The function has only one static call site directly after a full packet is stored in RAM via DMA. In the default configuration of Contiki-NG, *read_frame()* is now called with a 128-byte buffer as input. The function calculates the payload length specified by the *phr* field of the received packet and verifies that it stays within the expected range of an IEEE 802.15.4 packet. Hence, the contents of the packet in *rx_buf* are only copied to the input frame buffer if the packet consists of less than 128 bytes. At this point in time, static analysis may conclude that while *read_frame()* is not optimally implemented, it may not directly

introduce a security vulnerability, and existing automated rehosting systems are unable to reach this code path due to missing support for receiving input via the M2 mechanism.

However, when fuzzing this sample with GDMA, we detected a buffer overflow during the call to *memcpy*. The reason for this is an *indirect call* to *read_frame()* taken during runtime from the higher *nrf_ieee_driver* layer, which is easily missed during static analysis. This call to *read_frame()* aims to receive an acknowledgment packet over the air after sending an 802.15.4 Carrier Sense Multiple Access (CSMA) packet. This sending occurs in a completely different part of the radio stack. Such an acknowledgment packet has a fixed size of three bytes. Thus, *read_frame()* is called with a three-byte input buffer. However, a malicious sender can respond with a larger acknowledgment, leading to the vulnerability.

Only GDMA is able to automatically detect this vulnerability based on a default build of an nRF sample and by running Fuzzware with the newly introduced `--dma` flag. As the peripheral leverages Nordic's easyDMA feature (with type M2), DICE is unable to emulate the DMA functionality. While other work, such as HALucinator [10], which replaces the HAL, can handle DMA transfers in general, it would also miss the bug, as the bug itself resides in the HAL. This highlights the benefits of a full rehosting approach. By utilizing DMA emulation, a fuzzer can explore the entire breadth of the firmware – not only manually selected parts.

7 Related Work

DMA vulnerabilities, protection mechanisms, and related research are very diverse due to the ubiquitous nature of DMA.

DMA in general-purpose computing systems. Prior work has demonstrated that data transferred via DMA is a viable attack vector enabling malicious peripherals to compromise their host system [1, 17, 26]. To find vulnerable implementations, SADA [4] identifies unsafe DMA accesses in Linux device drivers statically. Other approaches implement dynamic, fuzzing-based techniques to identify security vulnerabilities in drivers [29, 37, 38, 45] and devices emulated by hypervisors [5, 6, 24, 25, 34] that arise from, among other factors, improperly handled DMA inputs. These approaches rely on standardized APIs of general-purpose operating systems, such as Linux or the Advanced Configuration and Power Interface (ACPI) standard. Embedded firmware lacks such standardized abstractions, so this prior work is inapplicable to the problem space targeted by GDMA.

To mitigate some of the risks associated with using DMA for peripheral communication, Wang et al. [43] proposed to implement pointer authentication schemes for DMA accesses. Bahmani et al. [3] developed a method to integrate DMA into the security model of trusted execution environments. So far, these approaches have not seen commercial adoption.

DMA in rehosted firmware. Firmware rehosting to enable the fuzzing of embedded devices has attracted significant research interest in recent years [14, 44]. One of the primary challenges identified in this domain is the emulation of peripherals, which is crucial for accurate and effective firmware analysis. Such peripherals typically communicate using either MMIO, which has been covered extensively in prior work [9, 13, 15, 16, 31, 32, 48], or using DMA.

Beyond DICE [27], VDEmu [19] reimplements DICE's DMA handling (M1) in Fuzzware and is therefore inherently limited by the same constraints w.r.t. DMA detection as DICE. Other state-of-the-art approaches, capable of emulating DMA accesses, generally follow one of three different approaches: (1) Bypassing DMA emulation and instead intercepting calls into the Hardware Abstraction Layer (HAL) of the firmware, (2) emulating specific DMA controllers (3) relying on physical hardware. HALucinator [10], Para-Rehosting [21], Safirefuzz [35] implement emulation of DMA peripherals by relying on HAL functions that offer a standardized interface for interacting with various peripherals. Other approaches [20, 50] implement specific DMA controllers based on externally provided information, such as textual specifications or driver source code. Lastly, several approaches [12, 23, 28] bypass emulation entirely and use physical hardware to execute firmware. Therefore, the security properties of the firmware are evaluated in a realistic environment, ensuring that discovered vulnerabilities are unlikely to be false positives.

Our work. In comparison, GDMA is agnostic of the firmware under emulation, requiring no prior knowledge, hardware, specifications, or source code of the target.

8 Discussion

Exhaustiveness of DMA Mechanisms. Our evaluation of reference manuals shows that six types capture the mechanisms used by a wide range of silicon vendors. However, we stress that hardware designers, in principle, may freely design these DMA mechanisms. To detect arbitrary DMA mechanisms outside of our evaluation, the DMA configuration synthesis described in Section 4 would have to be extended. While we cannot predict future hardware manufacturer design decisions, we design GDMA's components in a generic way. For example, we allow the GDMA model configuration format to express DMA descriptors of arbitrary compound types instead of tying it to a particular type of DMA descriptor. This is designed to ease the integration of future DMA mechanisms.

Non-ARM Targets. We focus on ARM MCUs, which dominate the embedded sector, especially in the 32-bit market [2]. DICE included two PIC32 MCUs in its evaluation. A review of the respective reference manuals [40, 41] reveals that these PIC32 MCUs employ only M1 descriptors. As such, these samples would add little to our evaluation, as the types represented in our dataset already include the single DMA mechanism used in these PIC32 MCUs. However, as our implementation builds upon Fuzzware (which does not

provide support for PIC32 MCUs), supporting other architectures would require extending the architecture support of Fuzzware. We consider adding multi-architecture support to Fuzzware outside the scope of this work.

Potential Future Work. A key feature of GDMA is the filtering mechanism introduced by our MMIO isolation and type inference (see Section 4). This step narrows down the potential DMA configuration candidates. Based on this pre-filtering, more heavy-weight program analysis techniques such as symbolic execution could be employed to further refine the modeling process or address challenges that stem from previously unaddressed DMA mechanisms. This is why we believe that GDMA may be leveraged as a basis for future fully automated DMA modeling techniques.

In addition to the DMA modeling process itself, GDMA could be used to test targets outside of embedded firmware. For example, a possible direction to pursue is the testing of operating system driver implementations.

9 Conclusion

In this work, we addressed the so far largely untouched challenge of fully automated and generic DMA rehosting. We introduced a fully reproducible data set to enable the evaluation of DMA rehosting of all of the six DMA configuration mechanisms that we observed. GDMA successfully handled all of these mechanisms, which is 6x the number of mechanisms supported by the previous state of the art. In addition, barring fuzzer limitations, GDMA is the first to cover all targets of the DICE DMA dataset. Finally, we have shown that GDMA improved fuzzing effectiveness drastically. GDMA increases the code coverage by between 3.5% and 152.6% on DMA-enabled firmware targets and identified 6 new bugs, each of which was assigned a CVE.

Acknowledgments

We thank our anonymous reviewers for their valuable feedback. This work was funded by the European Research Council (ERC) under the consolidator grant RS^3 (101045669) and the Federal Ministry of Research, Technology and Space (BMFTR) under the grants CPSec (16KIS1899), ASRIOT (16KIS1901), and Startup Secure (16KIS2362). This work was also supported by the German Federal Office for Information Security (FKZ: Pentest-5GSec - 01MO23025B).

Ethics Considerations

Our paper presents a method for more effectively identifying software faults in firmware images. We are confident that we adhere to the high ethical standards and best practices of the research community. To minimize potential harm, we promptly disclosed the identified vulnerabilities to the respective projects shortly after discovering them, following the community's established coordinated disclosure practices. Moreover, we provided assistance during the analysis and patching of the vulnerabilities and supported the developers in mitigating the bugs we identified. In total, 6 CVEs have been assigned to our findings so far.

Open Science

In accordance with the principles of open science, we open source our artifacts at https://www.github.com/fuzzware-fuzzer/gdma-experiments and at https://doi.org/10.5281/zenodo.15600641.

Our research artifact includes the following five different components:

1. The firmware data set outlined in Table 2, which consists of 15 new firmware samples, spanning 10 hardware platforms and 6 different DMA mechanisms. The data set is fully reproducible; i. e., each sample is provided with its sources and a docker container to build the sample.
2. The second firmware data set (see Figure 6), which includes 10 different firmware based on vendor examples. For this sample set, we also provide a docker-based build reproduction set-up.
3. The ground truth metadata that we created for the DICE dataset discussed in Section 6.4.
4. A reproducible, docker-based reproduction environment for DICE and its non-DMA baseline P2IM.
5. The source code to GDMA, which enables Fuzzware [31] to effectively rehost DMA.

References

[1] Markuze Alex, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. Characterizing, exploiting, and detecting DMA code injection vulnerabilities in the presence of an IOMMU. In *European Conference on Computer Systems (EuroSys)*, 2021.

[2] AspenCore. The Current State of Embedded Development. https://www.embedded.com/wp-content/uploads/2023/05/Embedded-Market-Study-For-Webinar-Recording-April-2023.pdf (Online; Accessed 2025-01-15), 2023.

[3] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A security architecture with CUstomizable and resilient enclaves. In *USENIX Security Symposium*, 2021.

[4] Jia-Ju Bai, Tuo Li, Kangjie Lu, and Shi-Min Hu. Static detection of unsafe DMA accesses in device drivers. In *USENIX Security Symposium*, 2021.

[5] Alexander Bulekov, Bandan Das, Stefan Hajnoczi, and Manuel Egele. Morphuzz: Bending (input) space to fuzz virtual devices. In *USENIX Security Symposium*, 2022.

[6] Alexander Bulekov, Qiang Liu, Manuel Egele, and Mathias Payer. HYPERPILL: Fuzzing for hypervisor-bugs by leveraging the hardware virtualization interface. In *USENIX Security Symposium*, 2024.

[7] Zitai Chen, Sam L Thomas, and Flavio D Garcia. Metaemu: An architecture agnostic rehosting framework for automotive firmware. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[8] Michael Chesser, Surya Nepal, and Damith C. Ranasinghe. Icicle: A re-designed emulator for grey-box firmware fuzzing. In *International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery, 2023.

[9] Michael Chesser, Surya Nepal, and Damith C Ranasinghe. Multifuzz: A multi-stream fuzzer for testing monolithic firmware. In *USENIX Security Symposium*, 2024.

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

[11] CounterCycle. Oscilloscope test binary is broken. https://github.com/RiS3-Lab/DICE-DMA-Emulation/issues/9. Accessed: 2025-01-22.

[12] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. Fuzzing embedded systems using debug interfaces. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2023.

[13] Guy Farrelly, Michael Chesser, and Damith C Ranasinghe. Ember-IO: effective firmware fuzzing with model-free memory mapped IO. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2023.

[14] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, et al. Sok: Enabling security analyses of embedded systems via rehosting. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2021.

[15] Bo Feng, Alejandro Mera, and Long Lu. P2IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *USENIX Security Symposium*, 2020.

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

[17] Taehun Kim, Hyeongjin Park, Seokmin Lee, Seunghee Shin, Junbeom Hur, and Youngjoo Shin. Devious: Device-driven side-channel attacks on the IOMMU. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[19] Youngwoo Lee, Juhwan Kim, Jihyeon Yu, and Joobeom Yun. Embedded firmware rehosting system through automatic peripheral modeling. *IEEE Access*, 2023.

[20] Chongqing Lei, Zhen Ling, Yue Zhang, Yan Yang, Junzhou Luo, and Xinwen Fu. A Friend's Eye is A Good Mirror: Synthesizing MCU Peripheral Models from Peripheral Drivers. In *USENIX Security Symposium*, 2024.

[21] Wenqiang Li, Le Guan, Jingqiang Lin, Jiameng Shi, and Fengjun Li. From library portability to para-rehosting: Natively executing microcontroller software on commodity hardware. *Symposium on Network and Distributed System Security (NDSS)*, 2021.

[22] Wenqiang Li, Jiameng Shi, Fengjun Li, Jingqiang Lin, Wei Wang, and Le Guan. muafl: non-intrusive feedback-driven fuzzing for microcontroller firmware. In *International Conference on Software Engineering (ICSE)*, 2022.

[23] Changming Liu, Alejandro Mera, Engin Kirda, Meng Xu, and Long Lu. CO3: Concolic Co-execution for Firmware. In *USENIX Security Symposium*, 2024.

[24] Yuwei Liu, Siqi Chen, Yuchong Xie, Yanhao Wang, Libo Chen, Bin Wang, Yingming Zeng, Zhi Xue, and Purui Su. Vd-guard: Dma guided fuzzing for hypervisor virtual device. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2023.

[25] Zheyu Ma, Qiang Liu, Zheming Li, Tingting Yin, Wende Tan, Chao Zhang, and Mathias Payer. Truman: Constructing device behavior models from os drivers to fuzz virtual devices. In *Symposium on Network and Distributed System Security (NDSS)*, 2025. To appear.

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

[27] Alejandro Mera, Bo Feng, Long Lu, and Engin Kirda. Dice: Automatic emulation of dma input channels for dynamic firmware analysis. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[28] Alejandro Mera, Changming Liu, Ruimin Sun, Engin Kirda, and Long Lu. SHiFT: Semi-hosted Fuzz Testing for Embedded Applications. In *USENIX Security Symposium*, 2024.

[29] Masanori Misono, Toshiki Hatanaka, and Takahiro Shinagawa. Dmafv: testing device drivers against dma faults. In *ACM Symposium on Applied Computing (SAC)*, 2022.

[30] Nicolas Nino, Ruibo Lu, Wei Zhou, Kyu Hyung Lee, Ziming Zhao, and Le Guan. Unveiling IoT Security in Reality: A Firmware-Centric Journey. In *USENIX Security Symposium*, 2024.

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

[32] Tobias Scharnowski, Simon Woerner, Felix Buchmann, Nils Bars, Moritz Schloegel, and Thorsten Holz. Hoedur: Embedded Firmware Fuzzing using Multi-Stream Inputs. In *USENIX Security Symposium*, 2023.

[33] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. SoK: Prudent Evaluation Practices for Fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2024.

[34] Sergej Schumilo, Cornelius Aschermann, Ali Abbasi, Simon Wörner, and Thorsten Holz. Hyper-cube: High-dimensional hypervisor fuzzing. In *Symposium on Network and Distributed System Security (NDSS)*, 2020.

[35] Lukas Seidel, Dominik Christian Maier, and Marius Muench. Forming faster firmware fuzzers. In *USENIX Security Symposium*, 2023.

[36] NXP Semiconductors. LPC18xx ARM Cortex-M3 microcontroller. https://community.nxp.com/pwmxy87654/attachments/pwmxy87654/lpc/55570/1/UM10430%20(1).pdf (Online; Accessed 2025-01-15), 2019.

[37] Zekun Shen, Ritik Roongta, and Brendan Dolan-Gavitt. Drifuzz: Harvesting bugs in device drivers from golden seeds. In *USENIX Security Symposium*, 2022.

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

[39] Statista. Leading microcontroller unit (MCU) manufacturers worldwide in 2021, by market share. https://www.statista.com/statistics/1327509/top-mcu-suppliers-worldwide/ (Online; Accessed 2025-05-09), 2021.

[40] Microchip Technology. PIC32MX5XX/6XX/7XX Family Data Sheet. http://ww1.microchip.com/downloads/en/DeviceDoc/60001156J.pdf (Online; Accessed 2025-01-15), 2016.

[41] Microchip Technology. PIC32MZ Embedded Connectivity with Floating Point Unit (EF) Family. https://ww1.microchip.com/downloads/aemDocuments/documents/MCU32/ProductDocuments/DataSheets/PIC32MZ-Embedded-Connectivity-with-Floating-Point-Unit-Family-Data-Sheet-DS60001320H.pdf (Online; Accessed 2025-01-15), 2021.

[42] András Vargha and Harold D Delaney. A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[43] Xingkai Wang, Wenbo Shen, Yujie Bu, Jinmeng Zhou, and Yajin Zhou. DMAAUTH: A lightweight pointer integrity-based secure architecture to defeat DMA attacks. In *USENIX Security Symposium*, 2024.

[44] Christopher Wright, William A Moeglein, Saurabh Bagchi, Milind Kulkarni, and Abraham A Clements. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys (CSUR)*, 2021.

[45] Yilun Wu, Tong Zhang, Changhee Jung, and Dongyoon Lee. Devfuzz: automatic device model-guided device driver fuzzing. In *IEEE Symposium on Security and Privacy (S&P)*, 2023.

[46] Wei Xie, Yikun Jiang, Yong Tang, Ning Ding, and Yuanming Gao. Vulnerability detection in iot firmware: A survey. In *International Conference on Parallel and Distributed Systems (ICPADS)*, 2017.

[47] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. {FIRM-AFL}:{High-Throughput} greybox fuzzing of {IoT} firmware via augmented process emulation. In *USENIX Security Symposium*, 2019.

[48] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Automatic Firmware Emulation through Invalidity-guided Knowledge Inference. In *USENIX Security Symposium*, 2021.

[49] Wei Zhou, Shandian Shen, and Peng Liu. IoT Firmware Emulation and Its Security Application in Fuzzing: A Critical Revisit. *Future Internet*, 17(1), 2025.

[50] Wei Zhou, Lan Zhang, Le Guan, Peng Liu, and Yuqing Zhang. What your firmware tells you is not how you should emulate it: A specification-guided approach for firmware emulation. In *ACM Conference on Computer and Communications Security (CCS)*, 2022.

[51] Wei Zhou, Lan Zhang, Le Guan, Peng Liu, and Yuqing Zhang. SEmu-Fuzz implementation. https://github.com/IoTS-P/SEmu-Fuzz/blob/b6e381d8a9ffc2738450ec53b72effee85a9cb4d/semu_fuzz/emulate/semu/rule.py#L536C51-L536C62 (Online; Accessed 2025-01-15), 2023.

A Data Type Merging

Table 6: Overview of data type transitions: While merging the types of two fields of a node within a type tree, the following type merging rules are applied. Types are merged from weaker to more dominant types in the order `Zero`, `Pointer`, and `HighEntropy`.

	Zero	Pointer	HighEntropy
Zero	Zero		
Pointer	Pointer	Pointer	
HighEntropy	HighEntropy	HighEntropy	HighEntropy

B Detailed DICE Unit Test Overview

As described in Section 6.4, there are different types of misclassifications. We describe the exact distribution of misclassifications in Table 7. As mentioned in Section 6.4, we attribute the misclassifications to two different sources: DICE heuristic mismatch and wrong transfer direction classification. We note that only DICE suffers from these misclassifications, as GDMA correctly identifies the DMA configuration if the fuzzer covers both DMA configuration as well as DMA usage.

A missing DMA detection occurs if the DICE heuristics do not apply. The DICE data set includes six samples that use DMA mechanism M2 (all 6 nRF samples) and one sample that uses DMA mechanism M3 (the `SAM3X ADC_PDC.ino`). These DMA mechanisms are not detected. DICE mentions the nRF samples as a limitation but does not mention that the second type of SAM3X DMA configuration is not supported. However, the reference manual reveals that the source and destination pointer in the second type of DMA are non-adjacent. Consequently, this is an M3 DMA configuration mechanism and is unsupported by the DICE heuristic.

A transfer direction misclassification occurs due to another DICE heuristic: If the firmware reads from a buffer that is referred to by an MMIO register, DICE classifies the DMA transfer as a receive transfer. A transfer direction misclassification occurs if a transmit buffer is read from. DICE incurs one such misclassification.

Finally, both DICE and GDMA suffer from limitations of the underlying fuzzing engines, P2IM and Fuzzware, respectively. If the fuzzing engine does not cover DMA usage, no transfer can be detected. We stress that this is a limitation of neither approach; instead, it is a limitation of P2IM and Fuzzware. P2IM fails to cover the use of DMA in 10 cases, Fuzzware in 2 cases.

We conclude that GDMA is the first tool to successfully rehost DMA for all samples, barring fuzzer limitations.

Table 7: Overview of GDMA and DICE performance on DICE unit tests.

Platform	Sample name	DICE	GDMA
F103	ADC_SingleConversion_TriggerSW_DMA	✓	✓
F103	ADC_SingleConversion_TriggerTimer_DMA	✓	✓
F103	ChibiOS_ADC_slider	✓	✓
F103	ChibiOS_SPI	✓	✓
F103	ChibiOS_UART	✓	✓
F103	ChibiOS_acelerometer	(✓)	✓
F103	I2C_OneBoard_AdvCommunication_DMAAndIT	✓	✓
F103	I2C_OneBoard_Communication_DMAAndIT	✓	✓
F103	I2C_TwoBoards_ComDMA	✓	✓
F103	I2C_TwoBoards_MasterTx_SlaveRx_DMA	(✓)	✓
F103	SPI_FullDuplex_ComDMA	✓	✓
F103	SPI_OneBoard_HalfDuplex_DMA	(✗)	✓
F103	SPI_OneBoard_HalfDuplex_DMA_Init	✓	✓
F103	SPI_TwoBoards_FullDuplex_DMA	✓	✓
F103	UART_HyperTerminal_DMA	(✓)	✓
F103	UART_TwoBoards_ComDMA	(✓)	✓
F103	USART_Communication_TxRx_DMA	✓	✓
F103	USART_SyncCommunication_FullDuplex_DMA	(✓)	✓
F429	ChibiOS_ADC_slider	(✓)	✓
L1521	ChibiOS_SPI	✓	✓
L1521	ChibiOS_UART	✓	✓
LPC1837	PDMA_memory	✓	✓
nRF51822	SPI_slave	✗	✓
nRF51822	console_bleprph	✗	(✓)
nRF52832	SPI_master	✗	✓
nRF52832	SPI_slave	✗	✓
nRF52832	console_bleprph	✗	(✓)
nRF52832	uart	✗	✓
NUC123	PDMA_memory	(✓)	✓
NUC123	PDMA_usart	(✓)	✓
SAM3X	ADC_PDC.ino	✗	✓
SAM3X	spi_spi_dmac_slave_example_flash	(✓)	✓
SAM3X	usart_dmac_example	(✓)	✓
✓: correct		15/33	31/33
(✓): fuzzer limitation (DMA usage not reached)		10/33	2/33
(✗): wrong transfer direction (read instead of transmit)		1/33	0/33
✗: undetected descriptor (DICE heuristic mismatch)		7/33	0/33

C Detailed DICE Fuzzing Tests Analysis

We provide a detailed comparison of DICE and GDMA performance. We note that we do not compare coverage directly, as the underlying fuzzing engines, P2IM and Fuzzware, have a big impact on coverage.

Figure 7 shows the coverage graphs of DICE and GDMA with their respective baselines, P2IM and Fuzzware. As expected, Fuzzware outperforms P2IM. This validates our approach to compare improvements over the baseline instead of absolute coverage numbers.

Our observations are in line with the results reported by DICE [27]. The `Soldering-Station` and `Guitar-Pedal` targets only marginally benefit from DMA. Upon manual inspection, we found that there are only marginal increases because the firmware performs computations on the DMA input, but this DMA input does not impact the control flow much. Consequently, the firmware can perform its computation on null bytes if no DMA input is provided. This is in contrast to targets that pass DMA input to parsers, such as

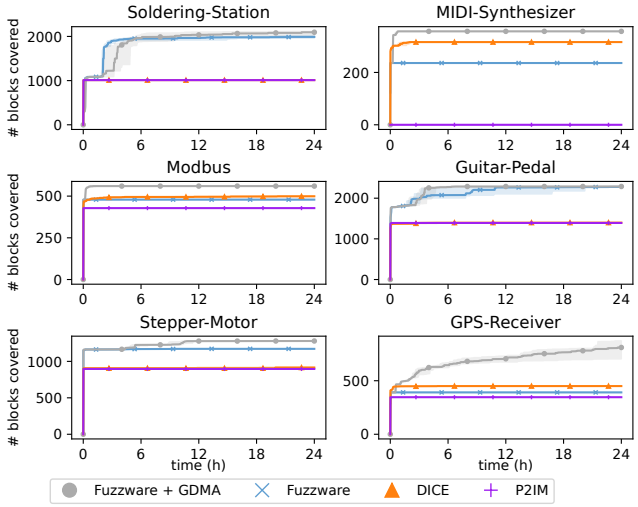

Figure 7: Coverage of DICE fuzzing tests. The plots show the median and the 25 and 75 percentile over 10 runs.

`GPS-Receiver` or `Modbus`. Here, the parser will always fail to parse early if no DMA input is provided. Supplying DMA thus allows for large coverage gains, as the fuzzer can now cover the parsing logic.

Overall, GDMA performs much better than DICE, but Fuzzware already significantly outperforms P2IM. This makes numerical comparisons of the two approaches hard. Instead, we analyze the coverage improvement over the respective baseline. The detailed results can be found in Table 8. First, we observe that our experiments show similar coverage improvements of DICE over P2IM as reported in DICE [27]. The only exception is the `Stepper-Motor` target. We assume this stems from the difference in experiment setup. Our experiments all run 24 hours, while DICE employs experiments over 48 hours. Indeed, manual inspection of the target reveals that DICE supplies DMA input to the parser of the motor commands, but does not progress far into the parsing logic. We assume that in 48 hours, P2IM cannot achieve additional coverage while DICE continues to explore the parser, leading to the difference in reported numbers in DICE.

In addition, we find that that Fuzzware (without DMA support) outperforms P2IM on all targets and outperforms DICE (with DMA support) on 50% of the firmware, proving the impact of the underlying fuzzing engine. We note that GDMA performance is always at least similar to DICE. In the `Guitar-Pedal` and `Modbus` targets, performance gains are similar between DICE and GDMA. In contrast, in the `GPS-Receiver`, `Soldering-Station` and `Stepper-Motor` targets, GDMA shows up to 79.1% more coverage improvement than DICE.

We conclude that GDMA detects DMA in every sample where DICE detects DMA. GDMA outperforms the state of the art in coverage improvements in most targets (**RQ3**).

Table 8: DICE and GDMA average improvements (in basic blocks) over P2IM and Fuzzware, respectively.

	#Total	#P2IM	#DICE	Δ DICE	#Fuzzware	#GDMA	Δ GDMA
GPS Receiver	3004	346	450	30.1%	391.0	819.5	109.6%
Guitar Pedal	8211	1391	1398.5	0.5%	2277.0	2289.6	0.1%
MIDI Synthesizer	703	0	316	–	236.0	357.0	51.3%
Modbus	811	428	498	16.4%	479.0	560.0	16.9%
Soldering Station	3308	1011	1011	0%	1966.9	2052.6	4.4%
Stepper Motor	4007	896	915.6	2.2%	1172.0	1288.3	9.9%

Table 9: List of CVEs assigned to bugs that were found by GDMA. The CVE identifiers are blinded for submission.

Software	CVE Identifier	Description
Contiki-NG	CVE-2023-48229	Out of bounds write while reading an IEEE 802.15.4 frame.
Mbed-OS	CVE-2024-48981	Out of bounds write due to missing size validation of HCI frames.
Mbed-OS	CVE-2024-48982	Integer wrap around leading to out of bounds write while reading HCI frames.
Mbed-OS	CVE-2024-48983	Integer overflow leading to out of bounds write while allocating memory.
Mbed-OS	CVE-2024-48985	Missing error handling after failed allocation leading to out of bounds write.
Mbed-OS	CVE-2024-48986	Out of bounds write on copy with illegal size.