PASan: Detecting Peripheral Access Concurrency Bugs within Bare-Metal Embedded Applications

Taegyu Kim, *Purdue University;* Vireshwar Kumar, *Indian Institute of Technology, Delhi;* Junghwan Rhee, *University of Central Oklahoma;* Jizhou Chen and Kyungtae Kim, *Purdue University;* Chung Hwan Kim, *University of Texas at Dallas;* Dongyan Xu and Dave (Jing) Tian, *Purdue University*

https://www.usenix.org/conference/usenixsecurity21/presentation/kim

This paper is included in the Proceedings of the 30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the 30th USENIX Security Symposium is sponsored by USENIX.

PASAN: Detecting Peripheral Access Concurrency Bugs within Bare-Metal Embedded Applications

Taegyu Kim[†], Vireshwar Kumar[*], Junghwan Rhee[§], Jizhou Chen[†]
Kyungtae Kim[†], Chung Hwan Kim[¶], Dongyan Xu[†], Dave (Jing) Tian[†]
[†]*Purdue University, {tgkim, chen2731, kim1798, dxu, daveti}@purdue.edu*
[*]*Indian Institute of Technology, Delhi, viresh@cse.iitd.ac.in*
[§]*University of Central Oklahoma, jhrhee@gmail.com*
[¶]*University of Texas at Dallas, chungkim@utdallas.edu*

Abstract

Concurrency bugs might be one of the most challenging software defects to detect and debug due to their non-deterministic triggers caused by task scheduling and interrupt handling. While different tools have been proposed to address concurrency issues, protecting peripherals in embedded systems from concurrent accesses impose unique challenges. A naïve lock protection on a certain memory-mapped I/O (MMIO) address still allows concurrent accesses to other MMIO addresses of a peripheral. Meanwhile, embedded peripherals such as sensors often employ some internal state machines to achieve certain functionalities. As a result, improper locking can lead to the corruption of peripherals' on-going jobs (we call *transaction corruption*) thus corrupted sensor values or failed jobs.

In this paper, we propose a static analysis tool namely PASAN to detect peripheral access concurrency issues for embedded systems. PASAN automatically finds the MMIO address range of each peripheral device using the parser-ready memory layout documents, extracts the peripheral's internal state machines using the corresponding device drivers, and detects concurrency bugs of peripheral accesses automatically. We evaluate PASAN on seven different embedded platforms, including multiple real time operating systems (RTOSes) and robotic aerial vehicles (RAVs). PASAN found 17 true positive concurrency bugs in total from three different platforms with the bug detection rates ranging from 40% to 100%. We have reported all our findings to the corresponding parties. To the best of our knowledge, PASAN is the first static analysis tool detecting the intrinsic problems in concurrent peripheral accesses for embedded systems.

1 Introduction

Concurrency bugs might be one of the most challenging software defects to detect and debug due to their non-deterministic triggers caused by task scheduling and interrupt handling. They not only lead to intermittent unexpected system behaviors but also contribute to attack surfaces. For instance, the Dirty Cow [1] vulnerability caused by a race condition in the memory subsystem enables privilege escalations within the Linux kernel. The race condition bug in VMware Tools on Windows 10 [17] causes privilege escalations in the virtual machines. The most recent privilege escalation vulnerability [16] in Android was caused by a race condition in the binder. Another race condition within BIND [9] allows a remote attacker to carry out Denial-of-Service of DNS servers. In fact, a simple keyword search for "race condition" in the CVE database shows 862 entries [10].

Multiple approaches have been proposed to address concurrency issues including static analysis [33,40,50,79], dynamic analysis [83,84], and hybrid analysis [54,55,62,73]. However, protecting peripheral devices[1] in embedded systems from concurrent accesses imposes unique challenges. A naïve lock protection on a certain memory-mapped I/O (MMIO) address still allows concurrent accesses to other MMIO addresses of a peripheral. In other words, unless there is a global lock for this peripheral and every MMIO access to the peripheral is protected by the same lock, race conditions still can exist on the peripheral.

Meanwhile, embedded peripherals often employ some internal state machine transitions to achieve a functionality. For instance, a sensor might need to go through different internal states[2] to accomplish one sensor read operation. We define such a specific sequence of internal state machine transitions as a *transaction*. Accordingly, the device driver often needs to access different MMIO addresses of the peripheral and even sleep in between to follow the peripheral's internal state machine transition. Note that unlike typical critical section protection, where sleep is excluded or even forbidden (e.g., spinlocks), the sleep here gives the embedded peripheral time to finish its job and corresponds to the part of the internal state machines (e.g., *wait*).

As a result, an effective concurrent peripheral access protection means the protection (locking) of both the MMIO

[1] We will also use simply *peripherals* in this paper interchangeably.

[2] e.g., *receive_cmd*: receiving a command, *wait*: waiting for an ongoing job completion, and *return_res*: returning the job result.

address range and the internal state machine transition of the peripheral for embedded systems. Unfortunately, none of the existing tools mentioned above acknowledges this unique concurrent protection requirement of embedded peripherals, and fails to detect potential concurrency bugs. Improper locking finally leads to the corruption of peripheral's on-going jobs, thus corrupting sensor values or failing jobs. We call such corruption of jobs as a *transaction corruption*.

In this paper, we propose PASAN (short for Peripheral Access SANitizer), a static analysis tool to detect peripheral access concurrency bugs for embedded systems. PASAN learns the MMIO address range of each peripheral device automatically using the memory layout documents. To gain the knowledge of the internal state machines, PASAN analyzes different device drivers to extract state machine models based on the correlation between device drivers and target peripherals. Leveraging the MMIO address ranges and internal state machines, PASAN finally detects the potential concurrent peripheral accesses and generate bug reports automatically.

We have evaluated PASAN on seven embedded platforms, including multiple real time operating systems (RTOSes) and robotic aerial vehicles (RAVs). PASAN has found 17 true positive concurrency bugs in total among three platforms with the bug detection rates ranging from 40% to 100%. We have reported all of our findings to the corresponding parties. To the best of our knowledge, PASAN is the first static analysis tool detecting the intrinsic problems in concurrent peripheral accesses for embedded systems. We summarize our contributions as follows.

- We analyze the unique challenges in concurrent peripheral access protection in embedded systems and define the correct protection to consider both of the MMIO address range and the internal state machines of peripherals at the same time.
- We design and implement PASAN, a static analysis tool to detect potential concurrency bugs for peripheral accesses in embedded systems. PASAN parses memory layout documents to extract MMIO address ranges automatically, learns the internal state machines by analyzing device drivers, and detects concurrency bugs by combining multiple underlying techniques of the MMIO address range identification, transaction abstraction, points-to analysis, and lockset analysis.
- We validate the capabilities of PASAN by evaluating its effectiveness on real-world embedded platforms, and discovering a total of 17 concurrency bugs in three different platforms.

2 Background and Motivation

Concurrency protection for peripheral accesses is a general practice for device driver writers on general-purpose operating systems such as Linux. For instance, in a Multi-Function Device (MFD) driver, a write function is protected via a *mutex* preventing concurrent accesses to the device as shown in Listing 1. Unfortunately, these simple concurrency protections fail on embedded systems due to the intrinsic states of bus types and embedded peripherals. Take Figure 1 as an example, where an `LIS3DH` sensor and an SD card controller are attached to an Serial Peripheral Interface (SPI) bus. A naïve concurrency protection for any operations on these peripherals or the bus does not protect the internal *state machines* of these devices, leading to a job failure, data loss, etc. We note that these internal state machines exist on both embedded buses and peripherals. We define a complete transition of these bus- and peripheral-level internal state machines as a transaction to reflect its atomic requirement. Once such unprotected states and corresponding transactions are identified, attackers may exploit this attack surface and trigger unexpected bus- or peripheral-level state machine transition (e.g., via network interface) to cause security or safety critical issues.

```
1  int retu_write(struct retu_dev *rdev, u8 reg, u16 data)
2  {
3          int ret;
4
5          mutex_lock(&rdev->mutex);
6          ret = regmap_write(rdev->regmap, reg, data);
7          mutex_unlock(&rdev->mutex);
8
9          return ret;
10 }
```

Listing 1: A MFD device write function within the Linux kernel 5.4 protected by a mutex.

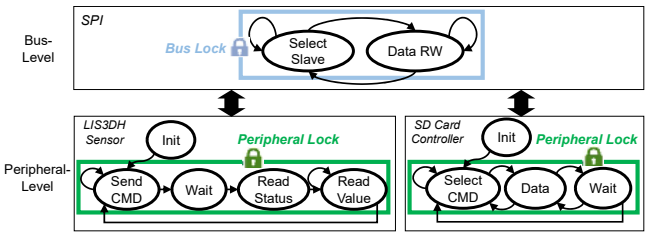

Figure 1: Simplified motivating example of state machines with SPI and attached peripherals.

Bus-Level State Machines. The SPI bus in Figure 1 is an I/O port controlling two attached peripheral devices. To communicate with any device, the bus needs to: (i) select the slave device and (ii) read/write data from/to the device. These two steps represent the internal state machines of this bus. Now imagine step (i) is done by thread A, which is going to send a command to the `LIS3DH` sensor. Simultaneously, thread B makes the SPI bus choose another slave device, i.e., the SD card controller. In this case, thread A's command will then be redirected to another slave device due to the transaction corruption of SPI caused by concurrent bus accesses. As a result, thread A never gets the response from the sensor because the transaction corruption leads to an erroneous redirection

Table 1: Comparison of concurrency bug detection approaches. The "Hybrid" analysis approach is based on both static and dynamic analysis; the "Algorithmic" indicates a theoretical approach without actual implementation; and the "Manual" approach requires manual efforts to detect (or prevent) concurrency bugs.

Work	Analysis Approach	Automatic Detection	Memory Objects	Address Range Aware	Transaction Aware
Lamport timestamps [60]	Algorithmic		✓		
Vector clock [66]	Algorithmic		✓		
Esterel [36]	Manual		✓		
Rust [65]	Manual		✓		
VCC [42]	Manual		✓		
VeriFast [32]	Manual		✓		
RacerX [50]	Static	✓	✓		
RELAY [79]	Static	✓	✓		
Vojdani et al. [78]	Static	✓	✓		
Chen et al. [40]	Static	✓	✓		
DSAC [33]	Static	✓	✓		
Polyspace [24]	Static	✓	✓		
Separation logic [69]	Static	✓	✓		
Mthread [20]	Static	✓	✓		
Coverity [15]	Static	✓	✓		
Infer [21]	Static	✓	✓		
Flawfinder [19]	Static	✓	✓		
CodeSonar [13]	Static	✓	✓		
ProRace [84]	Dynamic	✓	✓		
Cruizer [83]	Dynamic	✓	✓		
Hellgrind [67]	Dynamic	✓	✓		
ThreadSanitizer [73]	Hybrid	✓	✓		
RaceMob [55]	Hybrid	✓	✓		
LockDoc [62]	Hybrid	✓	✓		
Razzer [54]	Hybrid	✓	✓		
PASAN	Static	✓	✓	✓	✓

of the requested job. Note that putting a lock only on the step (i) will not eliminate the concurrency bug. To guarantee the exclusive access to the SPI bus, we need to protect the bus-level state machines, as denoted as *Bus Lock* (i.e., a blue box) in Figure 1.

Peripheral-Level State Machines. Embedded peripherals are often memory mapped within an embedded system and have their own internal state machines. As shown in Figure 1, the `LIS3DH` sensor (accelerometer) contains four states besides the *init* state. To read a value from the sensor, a thread A starts with a read command via writing into a memory mapped I/O (MMIO) address, which puts the sensor into the *read cmd* state. The sensor's internal state machine then transits to the *wait* state since the command processing takes some time depending on the sensor's working frequency (e.g., 50Hz). Now imagine another thread B sends a command to the sensor during the *wait* state. Due to such a transaction corruption, the sensor might produce an unexpected result, e.g., corrupted three-axis acceleration values, leading to an accident if it is used by a robotic vehicle. Similarly, putting a lock only on the state sending a command to the sensor cannot eliminate this concurrency bug. To achieve an exclusive access to a peripheral, we need to protect the peripheral-level state machines, e.g., all the four states of the sensor and all the three states of the SD card controller, as denoted as *Peripheral Lock* in Figure 1.

There have been a large body of the prior approaches for detecting concurrency bugs [13, 15, 19–21, 24, 32, 33, 36, 40, 42, 50, 54, 55, 60, 62, 65–67, 69, 73, 79, 83, 84]. As summarized in Table 1, most of them (classified as "Static", "Dynamic" and "Hybrid" in the analysis approach column) have not considered the concurrency issues caused by the race conditions in the internal state machines within bus and peripheral levels [13, 15, 19–21, 24, 33, 40, 50, 54, 55, 62, 67, 69, 73, 79, 83, 84]. Furthermore, other works (classified as "Manual" in Table 1) require to manually modify source code [36, 65] or insert annotations for analysis [32, 42] while relying on users to fully understand peripheral device operations. The other works (classified as "Algorithmic") even require the redesigning of the entire code base [60, 66]. This paper aims to detect a new class of concurrency bugs caused by the transaction corruption that has never been considered before PASAN – an *address-range-aware* and *transaction-aware* concurrency detection tool for embedded systems. In PASAN, we solve three main challenges:

- **C1: How to find a peripheral's MMIO address range automatically?** We note that a naïve protection on a single MMIO address operation is not enough due to the intrinsic behavior of the internal state machines. We need to know the whole MMIO address range given a peripheral, and lock the whole range to protect a transaction of the peripheral.
- **C2: How to find a peripheral's transaction scope automatically?** Recall that a transaction is essentially a complete transition of the internal states. To protect a transaction, we need to know where the transaction starts and ends within the code, and lock the whole transaction to protect the internal state machines.
- **C3: How to use MMIO addresses and transaction scopes to find bugs automatically?** With the above knowledge, we have an opportunity to detect concurrency issues of peripherals. We need a way to explore as many concurrency sources as possible while reducing false positives.

Usage Scenarios and Required Expertise. PASAN is an automatic tool that detects not just typical concurrency bugs, but specific concurrency bugs with transaction corruptions. Therefore, it does not assume users to have certain expertise. However, we expect the developers who respond to PASAN's bug report to have knowledge about in (1) embedded system device driver programming, (2) multi-threading, (3) race condition (e.g., lock/unlock usage), and (4) peripheral device data sheets. The aforementioned background is essential to understand the bugs and fix the race conditions.

We believe PASAN is particularly useful when appropriate dynamic device driver concurrency analysis tools are not available. This is quite common in the domain of embedded systems because of either the inability to instrument the related hardware devices (e.g., the peripheral device or the target board) for analysis or the unavailability of corresponding dynamic analysis frameworks. For instance, Hellgrind (part

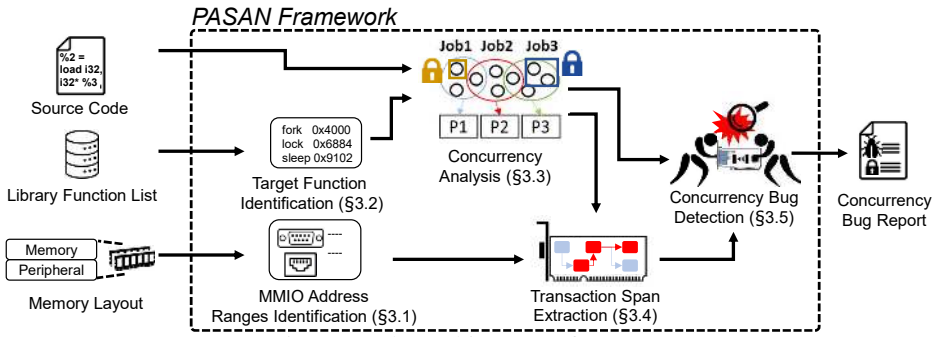

Figure 2: The architecture of PASAN.

of Valgrind [67]) cannot run on an RTOS, and QEMU [35] supports only a few boards and peripherals. Moreover, even state-of-the-art dynamic analysis tools have limited analysis coverage; it is hard for them to uncover concurrency bugs due to their intrinsic triggering conditions [55]. More importantly, they cannot find concurrency bugs with transaction corruptions.

3 Design

Concurrent memory accesses which do not consider the internal processing states of peripherals can lead to concurrency bugs. These bugs result in undefined behavior due to the generation of incorrect results or operation failures. We propose PASAN which provides a device-agnostic framework to detect such concurrency bugs. Different from the detection techniques in the prior art (that focuses on preventing concurrent accesses to certain program variables [33, 40, 50, 54, 55, 62, 73, 79, 83, 84]), PASAN takes a *transaction-aware* and *address-range-aware* concurrency bug detection approach which has resulted in the discovery of novel concurrency bugs in peripheral device transactions.

Figure 2 presents the overall architecture of PASAN framework. PASAN takes three inputs: (1) the source code of the host firmware which will compile into the LLVM bitcode [61], (2) the host firmware's memory layout including MMIO address ranges, and (3) the list of the library functions utilized by the host firmware. Then PASAN proceeds through the following steps to generate the concurrency bug report as the output automatically without requiring any user intervention and expertise. This report contains: (i) MMIO access instructions causing concurrency bugs, (ii) inferred transaction spans, and (iii) lock objects and their spans if they are enforced. For developing the rectified device driver, PASAN requires an expert to deal with false positives and fix bugs as discussed in Section 2.

1. **MMIO Address Range Identification (Section 3.1)**: First, PASAN parses the memory layout documents to identify the address ranges of MMIOs through which peripheral devices are attached to the host. By enabling the automated mapping of the accessed addresses to the corresponding MMIOs, this step plays an important role (in Step 4) in identifying the instructions belonging to the same transaction. As such, this step addresses the first aforementioned challenge (**C1** in Section 2).
2. **Target Function Identification (Section 3.2)**: Then, by analyzing the target LLVM bitcode, PASAN identifies the functions (e.g., multi-process, multi-thread, lock, and interrupt management functions) which are relevant for analyzing concurrently executable functions.
3. **Concurrency Analysis (Section 3.3)**: In this step, PASAN first identifies the instructions which can be executed concurrently. Out of those instructions, PASAN identifies the *existing* locked instructions (which are executed exclusively) via the context-sensitive lockset analysis [79]. Unlike the prior art, PASAN also considers the operations of interrupt handlers.
4. **Transaction Span Extraction (Section 3.4)**: Next, PASAN identifies all of the *transaction spans*, i.e., start and end pair of instructions belonging to one complete transaction of a peripheral device, by developing a set of span extraction heuristics. This novel technique to extract the proper *lock spans* enables PASAN to determine transaction-aware access patterns of peripheral devices, and addresses the second aforementioned challenge (**C2** in Section 2). We note that the complete transaction should *ideally* be locked (i.e., executed exclusively) to avoid concurrency bugs.
5. **Concurrency Bug Detection (Section 3.5)**: Finally, PASAN verifies whether the determined transaction span (obtained in Step 4) is correctly covered by the existing lock objects (obtained in Step 3). This addresses the last aforementioned challenge (**C3** in Section 2) and enables the detection of concurrency bugs by automatically checking whether an MMIO address can be concurrently accessed in the absence of a proper lock span.

We describe the details of each step of PASAN in the following sections.

3.1 MMIO Address Range Identification

MMIO enables the interaction between a host and peripheral devices by assigning a unique and fixed range of memory addresses for each peripheral. For example, a Universal

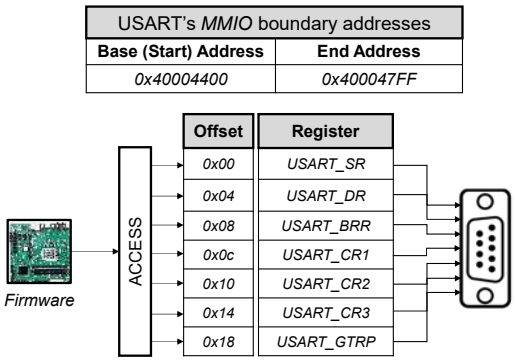

Figure 3: An MMIO address range corresponding to a Universal Synchronous/Asynchronous Receiver/Transmitter (USART).

Synchronous/Asynchronous Receiver/Transmitter (`USART`) is mapped to an address range used to control, receive and transfer data as illustrated in Figure 3. Therefore, in this step, to detect potential concurrent accesses to the same peripheral, PASAN identifies the MMIO address range allocated to each peripheral.

To identify these address ranges, PASAN utilizes the *memory layout* documents for the host including either the system view description (SVD) file [8] or the host-specific development tool libraries. We note that SVD is preferred because of the following reasons: (1) SVD contains the formally defined and accurate description of the memory layout of all MMIO address ranges; (2) SVD can be easily parsed thanks to its well-defined structure based on the Extensible Markup Language (XML) format; and (3) SVDs are available for a majority of hosts equipped with ARM architecture-based processors (e.g., Cortex-A and Cortex-M).

If an SVD file is not available, PASAN identifies the MMIO address ranges using the hard-coded base addresses in host-specific development tool libraries (e.g., header files). In this case, PASAN utilizes two common observations in embedded domains: (1) each peripheral is mapped to a unique address range, and (2) each peripheral is accessed by loading a hard-coded base address. Exploiting these observations, PASAN determines the MMIO address range for a peripheral starting with the base address for the peripheral and ending with the address right before the base address of the closest next peripheral. For example, as shown in Figure 3, the MMIO address range of `USART` spans from `0x40004400` to `0x400047FF`.

3.2 Target Function Identification

In this step, PASAN identifies the functions related to potential concurrent MMIO accesses and lockings that are essential to identify concurrently executable code. Specifically, PASAN handles four types of functions: (1) thread or process management functions (e.g., `pthread_create` and `pthread_join`) which are used to analyze the control flow of execution, (2) interrupt handler functions (e.g., `I2C_IRQHandler`) representing the starts of interrupt processes, (3) interrupt disable/enable functions (e.g., `enable_irq`) utilized to check whether interrupt handlers can execute concurrently, and (4) the functions related to locks and unlocks (e.g., `mutex_lock`) identifying the locked instructions and objects. It is important to consider interrupts because they can start a new transaction with a peripheral, thus corrupting the ongoing transaction of the peripheral. If none of relevant functions is found from the source file, PASAN looks for architecture-specific assembly instructions related to interrupts. For example, Cortex-M series architecture employs `cpsid` and `cpsie` instructions for disabling and enabling interrupts, respectively.

3.3 Concurrency Analysis

In this step, PASAN identifies which code can potentially be executed concurrently by tracking the code's starting/stopping threads and checking the enabling/disabling code of interrupt handlers. Next, by leveraging lockset analysis, PASAN identifies which code are not properly locked allowing concurrent execution of unlocked code by leveraging lockset analysis. Specifically, by analyzing the LLVM bitcodes and the list of the relevant library and interrupt handler functions (identified in Section 3.2), PASAN first identifies the *executable* processes, threads and interrupt handlers. Next, PASAN analyzes them to identify the *concurrently* executable instructions. Finally, PASAN performs lockset analysis to identify the instructions that are "locked" to prevent concurrent access. We provide the technical details of this analysis below, and describe them through an example shown in Figure 4 and Figure 5.

Executable Processes, Threads, and Interrupt Handlers. To infer this information, PASAN generates the call graph via points-to analysis [76], which is an established static analysis technique for identifying which memory locations the pointer variables can reference. Then, PASAN gathers the list of *entry* functions of processes, threads, and interrupt handlers. Starting from the *entry* function of the main process, PASAN finds instructions which call process and thread creation functions. Next, PASAN finds newly created functions from the arguments of these function call instructions. If such arguments are variables, PASAN finds the possible functions pointed by those variables via points-to analysis. One example is `main` function calling `pthread_create` with `IOThreadEntry` (the entry function) as the argument.

Concurrently Executable Code. PASAN identifies the concurrently executable code by analyzing the instructions corresponding to different processes and threads [47]. In this analysis, PASAN first discovers the life span of each process/thread by tracking its identifier via points-to analysis. A life span usually starts with the identifier initialized by the process/thread creation function, and ends when the identifier is passed back to the function after the process/thread termination function. For instance, the functions `waitpid` and

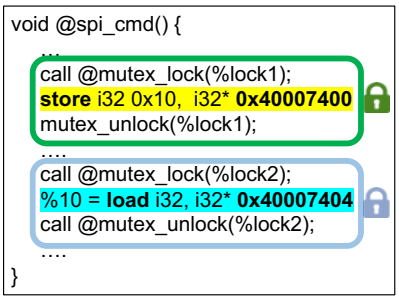

Figure 4: Code snippets for locked MMIO access instructions.

MMIO Access Instructions		store i32 0x10, i32* 0x40007400	%10 = load i32, i32* 0x40007404
Thread Call Stack List	Thread1 Call Stack	spi_cmd, csId: 1233 sd_write, csId: 798 log_z, csId: 622 update_z, csId: 210 **main, csId: -** *Call Stack*	spi_cmd, csId: 1233 sd_write, csId: 798 log_z, csId: 622 update_z, csId: 210 **main, csId: -**
	Thread2 Call Stack	spi_cmd, csId: 1233 sd_write, csId: 999 log_cmd, csId: 633 exe_cmd, csId: 30 **IOThreadEntry, csId: -**	spi_cmd, csId: 1233 sd_write, csId: 999 log_cmd, csId: 633 exe_cmd, csId: 30 **IOThreadEntry, csId: -**
	...		

Figure 5: Locked MMIO access instructions in different thread call stacks.

`pthread_join` may denote the end of a process and thread respectively. We obtain the concurrently executable code by detecting the overlap of the life spans of different processes and threads. For instance, in Figure 4, we observe that the `store` and `load` instructions are executed whenever the `spi_cmd` function is executed. In Figure 5, we consider that the two overlapping threads (i.e., Thread 1 corresponding to the `main` function, and Thread 2 corresponding to the `IOThreadEntry` function) call the `spi_cmd` function. Then, PASAN reports both `load` and `store` instructions (that are parts of a single transaction that must be atomically executed) as concurrently executable when those threads run simultaneously.

Lockset Analysis. After analyzing the lock/unlock and interrupt enable/disable functions (identified in Section 3.2), and the list of concurrently executable instructions (obtained above), PASAN identifies the *lock objects* used to lock instructions, and the *lock span* of each lock object, i.e., the start (using a `lock` function) and the end (using an `unlock` function) of the lock object.

For example, in Figure 4, the `store` instruction is placed between the `mutex_lock` and `mutex_unlock` functions with a lock object `lock1`. Similarly, a lock object called `lock2` is used for the `load` instruction. However, in spite of these locks, different threads (i.e., Threads 1 and 2 in Figure 5) can concurrently execute these locked instructions because different locks are used for the two instructions. To detect such cases, PASAN performs context-sensitive analysis of the complete *call stack*. Such call stack shows (1) the called functions on the stack and (2) the call instruction's unique identifier (*csId*) of its callee in a bottom-up fashion. As shown in Figure 5, these different call stacks help identify the potential threads which can execute concurrently.

In addition to detecting typical lock objects, PASAN also takes enabling and disabling of interrupts into account by considering them as lock and unlock functions respectively. In fact, the interrupt control flag can be considered as a *virtual* global lock object preventing interrupts from concurrent executions. PASAN also identifies recursive function calls, and avoids the analysis of duplicate functions in a loop. To identify such recursive function calls, we use the strongly connected component algorithm [68] employed in other static analysis systems as well, such as the points-to analysis framework employed by us [76].

3.4 Transaction Span Extraction

To find concurrency bugs for peripheral devices, PASAN must consider whether the concurrency can occur for transactions rather than for individual MMIO accesses (discussed in Section 2). As such, before the concurrency bug detection, PASAN must identify *transaction spans* that are the ranges of instructions representing transactions.

Specifically, as shown in Figure 5, the usage of different locks leads to peripheral access concurrency bugs. More importantly, even if the same lock was used, we still could not guarantee that both the `store` and the `load` come from the same thread. It might be Thread 1 `store` + Thread 2 `load` or Thread 2 `store` + Thread 1 `load`. In either case, neither Thread 1 nor Thread 2 would have the correct response from the peripheral due to the corruption of each thread's transaction with the peripheral.

Consequently, we need to detect each transaction initiated by different threads that can potentially interleave with each other and cause a transaction corruption. As the first step, we extract *all of* the transaction spans in advance. We argue that although drivers might lack proper locking, their implementations have to follow the operation instruction of the peripherals (aka, transaction) to make them work. Otherwise, these drivers simply would not work, which would be caught during the development or testing. More importantly, the extracted transaction spans need to be context-sensitive and MMIO-address-range-aware. The former provides call stacks with lock information (if exists); the later tells potential concurrent peripheral accesses from different MMIO addresses but within the same MMIO address range.

Finding the Peripheral-Access Instructions. PASAN identifies the peripheral-access instructions by following the occurrences of the `store` and `load` instructions, whose pointer argument might represent an MMIO access. We take the following approach to resolve possible address values of a given pointer variable: PASAN first performs points-to analysis to find the list of alias variables of the pointer variable. It then strives to find the *constant* MMIO address values propagated to such alias variables. This can be done by checking the

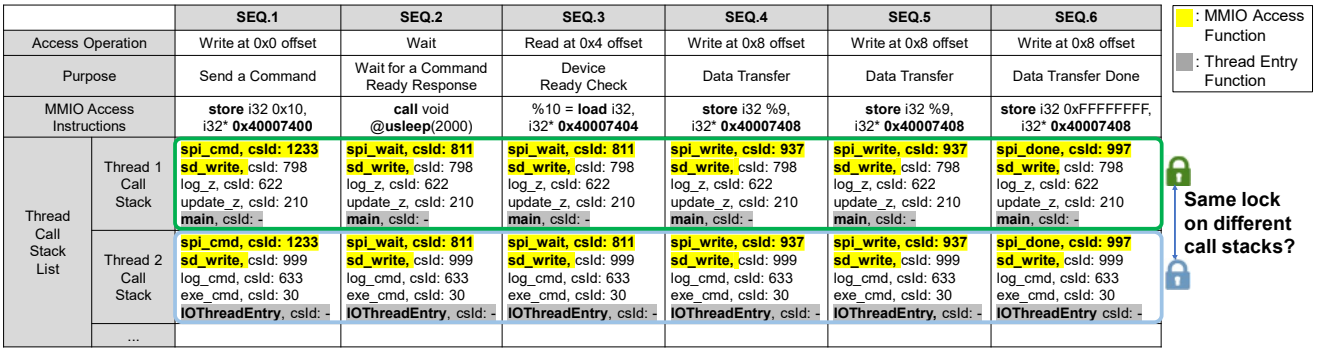

Figure 6: An example of a simple transaction for a peripheral device. The high-level operations are described in the four top columns. Then, we show transaction spans which should be covered by a respective ideal lock span identified by PASAN at the bottom with threads' call stacks. Here, Thread 1 is executed with `main` as an entry function, and Thread 2 is executed with `IOThreadEntry` as an entry. Both can be executable concurrently.

Algorithm 1 [T4] Transaction Span Extraction Per MMIO Address Range.

Input: Intermediate representation codes (IR), Target MMIO access instruction set (MI), Entry and interrupt handler bottom functions (E), Threshold values between device access instructions (Thr)
Output: Extracted transaction spans (L)

1: **function** TRANSACTIONSPANEXTRACTION(IR, Thr, MI, E) ▷ Main Function
2: *Initialize* L;
3: **for** $e \in E$ **do** ▷ Iterate all entry functions
4: *Initialize* cs;
5: cs.PUSH($\{e, NULL\}$) ▷ Initialize call stack (cs)
6: $L' \leftarrow$ RECURSIVEEXTRACTION(cs, Thr, L, IR, MI, e)
7: $L \leftarrow L \cup$ DISCONNSPAN(cs, L')
8: **return** L
9: **function** RECURSIVEEXTRACTION(cs, Thr, L, IR, MI, F)
10: $C \leftarrow$ GETINSTRUCTIONS(IR, F) ▷ F is target analysis function
11: **for** $c \in C$ **do** ▷ Analyze each instruction in F
12: **if** ISINSTBELONGSTOTRANSACTIONSPAN(cs, c, MI) **then**
13: $L \leftarrow$ EXTENDSPAN(cs, Thr, MI, c, L)
14: **else if** ISTOOLARGEDISTANCE(L, Thr) **then**
15: $L \leftarrow$ DISCONNSPAN(cs, L) ▷ Disconnect a too long transaction span
16: **else if** ISCALLINST(cs, c, IR) **then**
17: $callees \leftarrow$ GETNONREPEATEDRECURSIVECALLEES(cs, c, IR)
18: $L' \leftarrow L$
19: **for** $ce \in callees$ **do** ▷ Iterate non-repeated recursive callees
20: $L_{in} \leftarrow L'$ ▷ To keep the currently analyzed a transaction span
21: cs.PUSH($\{ce, c\}$) ▷ Update cs with a callee (ce) and call instruction (c)
22: $L_{in} \leftarrow$ RECURSIVEEXTRACTION($cs, Thr, L_{in}, IR, MI, F$)
23: cs.POP() ▷ Restore cs
24: $L \leftarrow$ UPDATELOCKSPAN(L_{in}, L)
25: **return** L

sequence of updates in each alias variable and backtracking relevant instructions, i.e., `store` and `load` instructions and value-updating instructions on the constant MMIO address values (e.g., `add` and `or` operations). During the backtracking, when PASAN finds a constant value for an alias variable, it maps an MMIO address range covering this constant value into that alias variable. Finally, the set of (potentially accessible) MMIO address ranges are mapped to each of the `store` and `load` instructions.

Determining Boundaries of a Transaction. Utilizing the list of instructions and their accessed MMIO address ranges, PASAN pursues the intuitive algorithm shown in Algorithm 1 to detect the boundaries (start and end) of a transaction. For each target instruction, PASAN computes a metric called "access distance" which is defined as the number of instructions between the target instruction and the next related instruction which access the *same* MMIO address range. We note that a large access distance indicates that the peripheral device's driver code are not executed for a large number of instructions, which may indicate the end of a transaction between the host and the peripheral (Line 14-15). As such, PASAN considers the target instruction and the next related instruction to belong to the same transaction if the access distance between them is smaller than a threshold denoted by Thr (Line 12-13). For example, in Figure 6, the instructions `SEQ.1,3-6` (as shown in the "MMIO Access Instructions" row) are determined to be part of the same transaction. We note that PASAN also collects thread call stacks as shown in the "Thread Call Stack List" in Figure 6 to check whether each transaction can be executed concurrently by different threads. We will explain how to use call stack information in Section 3.5.

Specifically, PASAN determines whether an MMIO access instruction belongs to a transaction span (Line 12) if the following three cases are satisfied.

- *Case-1: Peripheral MMIO Wait Pattern:* The host employs a `wait` instruction (e.g., the `sleep` function call) when it needs to wait for the completion of a job requested to the peripheral. In other words, a `wait` instruction is a part of the state machine of an ongoing transaction. Hence, in Figure 6, PASAN considers the instructions `SEQ.1` and `SEQ.2` to belong to the same transaction (part of Line 12-13).
- *Case-2: Different Access Distance Thresholds:* PASAN can encounter a mix of instructions accessing different peripherals with different drivers. In such cases, PASAN utilizes different threshold values for different peripherals. For example, to transfer a large amount of data to an Ethernet card, a device driver may delegate the data copy job to a direct memory access (DMA) unit. In this case, since it is usually a temporary, small job, PASAN selects a smaller threshold value (Thr_d) for the access distance instead of the default longer threshold value (Thr_i) (part of Line 12-13). We will further demonstrate the impact of this threshold in Section 4 and their effectiveness in Section 5.2.

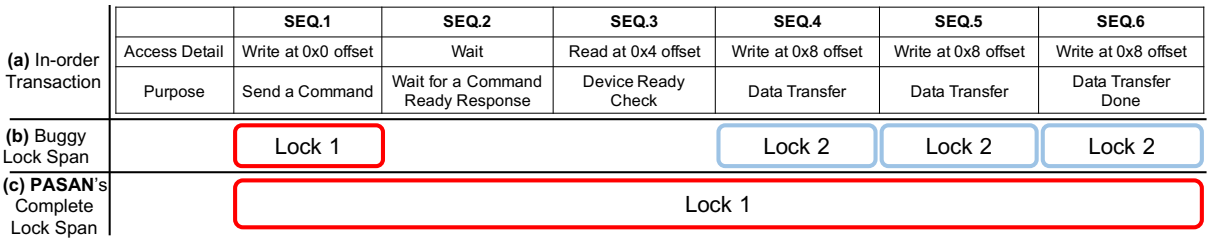

Figure 7: An example of a simple transaction for a peripheral device. The high-level operations are described in (a). Then, we show the example buggy lock enforcement in (b). Finally, we present the transaction span which should be covered by the proper lock span in (c).

- *Case-3: Write Access Inclusion:* PASAN considers an extracted transaction to be a potential target for a concurrency bug only if the transaction contains at least one `write` instruction. We note that the host can perform a `read` instruction (on a register that an MMIO address is mapped into) without interacting with any peripheral. In most cases, it usually does not affect the state machine transition of a peripheral. However, if we include read-only transactions, it would cause a large number of false positives because the status of some peripherals (e.g., timer and USART) are not volatile, and hence not vulnerable to unprotected concurrent reads as they maintain their own internal states. Therefore, we chose to use "write-access-inclusion" heuristic to reduce the false positive rate of concurrency bug detection in the next step (Section 3.5).

Handling Call Instruction. Once PASAN starts to analyze a call instruction, it recursively handles that call instruction first. During this step, PASAN keeps tracking the call stack to abide by *context-sensitivity* (Line 4-5, 21, 23). Other than that, PASAN needs to handle two challenges: recursive calls and indirect calls. To prevent repeated recursive function call analysis, PASAN generates a non-repeated callee list (Line 17) [68]. To handle the case of indirect calls, PASAN first retrieves the list of callees. If that is a direct call instruction, there is only one callee in the list. Otherwise, there can be multiple callees with different call stacks. For that, PASAN makes copies (corresponding to the number of such callees) of the transaction under analysis (Line 17-24). Note that these copied transactions are processed independently to determine their boundaries.

3.5 Concurrency Bug Detection

Now we describe how PASAN detects concurrency bugs caused by concurrent transactions of a peripheral. PASAN takes the following inputs from the previous steps: (1) concurrently executable instructions, (2) ranges of instructions locked by certain lock objects, and (3) transaction spans. Then, PASAN detects which parts of transactions can be concurrently executed even with the enforced locks.

We notice that these transaction concurrency bugs prevail in embedded systems because it is challenging for developers to correctly enforce every lock span to cover the complete transaction (we demonstrate two real-world examples in Sections 5.5 and 5.6). As such, we observe three common characteristics of a buggy lock span as demonstrated in Figure 7: (1) instructions (e.g., `SEQ.1` and `SEQ.4`) which access different MMIO addresses are locked separately; (2) an instruction (e.g., `SEQ.2`) accessing no MMIO address is not considered for locking; and (3) a load instruction (e.g., `SEQ.3`) performing a read-only access is not locked. In contrast, PASAN takes a novel approach combining the following two strategies: the *address-range-aware* strategy and *transaction-aware* strategy guided by the extracted complete transaction spans. We note that these transaction spans are obtained using Algorithm 1 in Section 3.4, e.g., the transaction span shown in Figure 7(c). Without the guidance of the extracted transaction spans, traditional concurrency bug detectors would either consider it unnecessary to protect some instructions or protect them with *different locks* and *separate lock spans* as shown in Figure 7(b). Next, we elaborate on how and why the transaction spans are related to the concurrency bug detection.

- *Address-range-aware strategy:* PASAN must check whether two accessed MMIO addresses are accessed by the same peripheral. For example, in Figure 7(a), `SEQ.1` accesses the memory at an offset of `0x0` from the base address, `SEQ.3` accesses the memory at offset of `0x4`, and `SEQ.4-SEQ.6` access the memory at an offset of `0x8`. With a naïve strategy, only `SEQ.4-6` will be considered as accesses by the same peripheral, and the resulting disconnected locks may cause concurrency issues. Hence, by employing an address-range aware strategy, PASAN detects `SEQ.1`, `SEQ.3` and `SEQ.4-6` can be accessed by the same peripheral.
- *Transaction-aware strategy:* PASAN must also check whether a sequence of instructions belonging to the same transaction is protected by a single lock span. For example, in Figure 7, PASAN detects that `SEQ.1` and `SEQ.4-SEQ.6` belong to the same transaction. Note that this strategy also helps to cover sequences `SEQ.2` (i.e., the wait pattern that was not considered as a part of a transaction) and `SEQ.3` (detected by the address-range-aware strategy), which are not normally considered as protection targets in spite of them being parts of the same transaction spanning from SEQ.1 to SEQ.6.

Algorithm 2 Concurrency Bug Detection Per MMIO Address Range.

Input: Mapping an instruction into a set of the possible contexts (M_{inst}), Mapping a MMIO into a set of transaction spans (M_t), Mapping an instruction into alias lock objects (M_{lock})
Output: Concurrency Bug Report (CR)

1: **function** ConcurrencyBugDetection(M_{inst}, M_t, M_{lock}) ▷ Main Function
2: *Initialize* CR;
3: **for** $T_i \in M_t$ **do** ▷ Get one transaction set
4: **for** $T_j \in M_t$ **do** ▷ Get another transaction to make a comparison pair
5: $CR \leftarrow CR \cup$ ConcurrencyBugAnalysis($T_i, T_j, M_{inst}, M_{lock}$)
6: **return** CR ▷ Get one lock span
7: **function** ConcurrencyBugAnalysis($T_i, T_j, M_{inst}, M_{lock}$)
8: *Initialize* cr;
9: **for** $t_{cs_i} \in T_i$ **do** ▷ Get one transaction with a call stack. cs is a call stack
10: **for** $t_{cs_j} \in T_j$ **do** ▷ Get another transaction with a call stack
11: **if** IsConcurrentlyExecutable($t_{cxt_i}, t_{cxt_j}, M_{inst}$) **then**
12: $s_i \leftarrow$ GetLockSpan(t_{cs_i}, M_{lock}) ▷ Get locks and their spans in t_{cs_i}
13: $s_j \leftarrow$ GetLockSpan(t_{cs_j}, M_{lock}) ▷ Get locks and their spans in t_{cs_j}
14: **if** CheckLockSpanAndObj($s_i, s_j, t_{cs_i}, t_{cs_j}$) == *False* **then**
▷ Check whether a lock protects both transactions
15: $cr \leftarrow cr \cup \{t_{cs_i}, t_{cs_j}\}$
16: **return** cr ▷ Return the concurrency bug result for this pair

Algorithm. To detect concurrency bugs, PASAN first identifies the transactions by combining both *address-range-aware* and *transaction-aware* strategies. Then PASAN analyzes concurrently executable instructions (obtained in Section 3.3) to check whether the proper lock objects have been employed to cover the transactions (extracted in Section 3.4).

Algorithm 2 shows the pseudo code of the concurrency detection mechanism. PASAN first takes two transactions (denoted as T_i and T_j) accessing the same MMIO address range from the transaction list (Line 3-4). Then, PASAN checks whether T_i and T_j can be executable concurrently (Line 7-16). We note that both transactions can be "identical" (i.e., $T_i = T_j$) when they are concurrently executed in two different threads. For example, two transactions shown in Figure 6 execute the same MMIO access functions (i.e., `sd_write` and its callee functions, such as `spi_cmd`, as indicated by the same call site identifier `csId`). However, those transactions can be executed concurrently because Thread 1 and Thread 2 (whose entry functions are `main` and `IOThreadEntry`) concurrently execute the same transaction in different call stacks and call sites as described in the "Thread Call Stack List" row. As such, PASAN must consider them for concurrency bugs if the locks are not identical between different call stacks or they do not cover `SEQ.1-6`.

As such, PASAN obtains the call stacks from the transaction (denoted as T_{cs_i} and T_{cs_j} in Line 9-11). If the call stacks are different, PASAN needs to check if their threads and their locksets are different. To determine if their threads are different, PASAN first checks the entry functions of T_{cs_i} and T_{cs_j} (Line 11). If that is true, PASAN obtains (i) lock spans and (ii) lock objects for MMIO access instructions of T_{cs_i} and T_{cs_j} (Line 12-13). Then, PASAN checks whether there is a concurrency issue between T_{cs_i} and T_{cs_j} (Line 14). Essentially, if the existing locks do not cover either T_{cs_i} or T_{cs_j}, each of them has a concurrency bug. Next, if the lock spans cover each of T_{cs_i} and T_{cs_j}, PASAN checks whether both of them are locked by the identical lock objects. If this is not true, PASAN considers this transaction pair can be executable concurrently, which means they have concurrency bugs. Once T_{cs_i} and T_{cs_j} are determined to have a concurrency bug, the result is updated in the generated concurrency bug report (Line 5 and 15).

Table 2: Target embedded platforms. NT: the number of threads; NI: the number of interrupt handlers; and ND: the number of compiled device drivers.

Platform	OS	Version	Lines of Compiled Code	Lines of All Codes	NT	NI	ND
ArduPilot [11]	ChibiOS	3.6.10	116,815	2,220,042	11	54	42
RaceFlight [26]	Bare-metal	06ef4c2*	46,683	206,888	1	36	17
RIOT [28]	RIOT	201907	17,378	1,542,403	3	17	33
Contiki [14]	Contiki	4.4	12,762	553,596	6	15	5
TS100 [31]	FreeRTOS	2.05	20,291	185,126	5	19	8
grbl [2]	Baremetal	0.8	5,857	52,777	1	11	5
rusEFI [29]	ChibiOS	e33798c*	89,405	2,302,209	14	54	4
Total	-	-	309,191	7,063,041	41	208	114

* When there is no proper version (e.g., when the developers have updated the codes, but have not tagged its version), we provide the commit number from the github repository.

Table 3: The number of peripheral devices attached to respective MMIOs in each target firmware.

Platform	SPI	I2C	UART	USB	GPIO	IRQ	Flash	ADC	DMA
ArduPilot	11	10	13	1	2	1	2	1	2
RaceFlight	2	5	2	1	2	1	2	1	2
RIOT	5	19	1	0	4	1	2	1	1
Contiki	0	0	1	0	2	1	1	0	0
TS100	0	1	1	0	1	2	1	1	1
grbl	0	0	1	0	1	2	1	0	0
rusEFI	0	0	0	1	1	1	1	0	0
Total	18	35	19	3	13	9	10	4	6

4 Implementation

PASAN mainly targets embedded systems and is designed to use only static analysis. We use LLVM 7.0 [61] and SVF 1.6 [76] as the base for our analysis. Peripheral device address memory layout is extracted from the SVD [8] or development tool libraries. Overall, our implementation is composed of over 7K lines of C++ code and various miscellaneous Python scripts for automation. After the evaluation of seven target embedded platforms (introduced in Section 5), we selected the parameters to extract lock spans for transactions (Section 3.4) with the empirical values, Thr_i as 5,000 and Thr_d as 2,000, yielding the highest lock span accuracy on average as discussed in Section 5.2.

5 Evaluation

We first introduce the target testing platforms (Section 5.1), and focus our evaluation on answering the questions below:

- **Q1**: How accurate is the transaction span inference?
- **Q2**: How effective is PASAN's concurrency bug detection?
- **Q3**: How effective is PASAN compared to the existing approaches?
- **Q4**: What real-world concurrency bugs are detected?

Table 4: Summarized results of transaction span extraction.

Platform	# of Transaction Spans		Accuracy (%)	# of Incorrectly Inferred Transaction Spans		
	Extracted	Correct		Subset	Superset	Mixed
ArduPilot	60	41	68.33	5	6	8
RaceFlight	30	26	86.67	2	2	0
RIOT	41	34	82.93	2	5	0
Contiki	9	8	88.89	0	1	0
TS100	12	11	91.67	0	1	0
grbl	13	8	61.54	4	0	1
rusEFI	18	13	72.22	0	5	0
Total	183	141	77.05	13	20	9

5.1 Evaluation Targets

Table 2 summarizes the information about our evaluation targets of 7 open-source embedded platforms. We selected this set of platforms with the following criteria: (i) different running environments (e.g., different RTOSes), and (ii) different peripheral devices (e.g., different sensors). The first two platforms (i.e., ArduPilot and RaceFlight) are for robotic aerial vehicles (RAVs), and RIOT and Contiki are RTOSes. We evaluated RIOT by putting all testing device drivers together to generate one bitcode file. We evaluated Contiki with the blink-hello application running multiple threads with MMIO accesses. TS100 is a soldering iron platform; grbl is for computer numerical control (CNC) milling controllers; and rusEFI is used for internal combustion engine control units. Each platform has lines of compiled code ranging from 5,857 to 116,815, with total lines ranging from 52,777 to 2,302,209, the number of threads ranging from 1 to 11, the number of interrupt handlers ranging from 11 to 54, and multiple peripherals ranging from 4 to 42. We note that most of interrupt handlers execute the simple tasks such as infinite loop execution (without doing anything), immediate acknowledgement of the interrupt, or a common interrupt handler call (e.g., a kernel panic handler).

Table 3 shows the types of device drivers used in our evaluation. We note that some device drivers can support different buses (e.g., SPI and I2C). Furthermore, GPIO can sometimes act as SPI or I2C according to the configuration. In either case, we count the number of device drivers individually.

5.2 Transaction Span Extraction Accuracy

As one of the critical steps in the concurrency bug detection, PASAN identifies the possible transaction spans based on the extraction approach (Section 3.4) focusing on the instructions of transactions which can be executed concurrently (Section 3.3). The details of extraction accuracy are presented in Table 4 showing the following information for each target platform: (1) the number of the extracted transaction spans, (2) the number of the correctly extracted transaction spans, (3) the accuracy of the extracted transaction spans, and (4) incorrectly inferred transactions (e.g., subset, superset and mixed transaction spans).

To identify the ground truth, we manually inspected source code for every transaction span. For example, we look into the function(s) accessing a target device with a sequence of instructions for a specific purpose (e.g., `sdcard_spi_read_blocks` to read data from an SD card). Such functions can be called by the external non-driver functions rather than device drivers. Overall the accuracy of PASAN's lock span extraction is 77.05% on average ranging from 61.54% to 91.67%. Several target platforms, RaceFlight, RIOT, Contiki, and TS100, achieve high accuracy, i.e., over 80%. Other platforms such as ArduPilot, grbl, and rusEFI show a reasonable accuracy ranging from 60% to 80%.

In terms of incorrectly inferred transaction spans, there are three categories of partial inferences, which might still be useful for concurrency bug analysis.

1. **Subset transaction span:** A subset transaction span may contain a subset of the complete device access instructions, which can cause false negatives and/or additional inaccurate transaction span generation. The number of this type of incorrectly inferred spans range from 0 to 5 in Table 4. However, PASAN can still utilize it to detect concurrency bugs because MMIO access instructions in each subset transaction span should also be executed atomically.
2. **Superset transaction span:** A superset transaction span includes potential bug cases along with other instructions. As PASAN detects concurrency bugs in device access instructions for any bug case within the span, some of the superset transaction spans may lead to false positives. The number of this type of incorrectly inferred spans range from 0 to 6 in Table 4.
3. **Mixed transaction span:** This involves both subset and superset transaction spans. Therefore, it may lead PASAN to detect concurrency bugs with false positives and negatives. The number of this type of incorrectly inferred spans is from 0 to 8 in Table 4.

There are a couple of reasons why we could not achieve higher extraction accuracy according to our ground truth study. In the case with the lowest accuracy, execution of the application code (e.g., controller computation or sensor value conversion code in robotic vehicles) and the peripheral device management code frequently interleave. This causes our heuristic distances (discussed in Section 3.4 and 4) to be suboptimal because the different level of mixture with application code varies the optimal distance thresholds leading to incorrect transaction span extraction. Another main reason is that some platforms continue the device initialization steps whose access patterns are intensive and complex, even after threads or child processes have started. The initialization steps configure the device and its I/O setting, during which the platforms interact with diverse peripheral devices and I/Os rather than running application code. Consequently, our device access distance threshold values (i.e., the values of Thr_i and Thr_d mentioned in Section 4) are not optimal in those steps. For example, we found that ArduPilot hands over certain initial-

Table 5: Summary of concurrency bugs.

Platform	# of Bugs	# of False Positive Bugs	False Positive Rates	Bug Detection Rates	# of Affected Device Drivers
ArduPilot	20	12	60.0%	40.0%	7
RaceFlight	0	0	-	-	0
RIOT	9	1	11.11%	88.89%	8
Contiki	0	0	-	0	
TS100	1	0	0.0%	100.0%	1
grbl	0	0	-	0	
rusEFI	6	6	100.0%	0.0%	0
Total	36	19	-	-	16

ization steps to threads and processes communicating with the dedicated devices during the early execution stages. Finally, indirect calls to support multiple different I/Os also lead to low extraction accuracy. e.g., in ArduPilot.

5.3 Concurrency Bug Detection Effectiveness

Ground Truth Study Experiment. We find patches in RIOT related to the bus-level concurrency bugs in I2C[3] and SPI[4]. Before those patches, there were no locks at all, and hence any peripheral device attached to either I2C or SPI bus had concurrency bugs in RIOT. We use those patches as the ground truth by removing this patch in our RIOT testing. PASAN found all the 28 concurrency bugs fixed by the patch with 0% false positive rates. We apply the removed patch again for the following RIOT testing.

Bug Detection. As shown in Table 5, we evaluate each target platform on: (1) the number of concurrency bugs, (2) the number of false positives cases, (3) the bug detection rates, (4) the false positive rates in the bug detection, and (5) the number of potentially affected devices. In total, PASAN reported 36 bugs from ArduPilot, RIOT, TS100, and rusEFI platforms. After verification, we found that 17 out of 36 reported bugs are true positives, and the rest 19 cases are false positives. Among the 17 true positive cases, 8 cases are from RIOT. While the patch mentioned earlier fixed some bus-level concurrency bugs in RIOT, these 8 are new peripheral-level concurrency bugs. After we found aforementioned bugs in RIOT, we checked patch histories and found that ten peripheral devices had concurrency bugs with transaction corruptions[5]. However, RIOT developers did not consistently apply the similar patches to the other peripheral device drivers containing concurrency bugs. We reported our findings to RIOT developers, and they acknowledged our findings as bugs[6]. All the bugs found in ArduPilot are peripheral-level concurrency bugs. TS100's case is a generic concurrency bug on MMIO accesses caused by interrupt handling. Overall, PASAN achieves bug detection rates from 40.0% to 100.0%.

[3] https://github.com/RIOT-OS/RIOT/pull/2323/commits for three boards before the patch.

[4] https://github.com/RIOT-OS/RIOT/pull/2317/commits for nine boards before the patch.

[5] https://github.com/RIOT-OS/RIOT/pull/2326/commits.

[6] https://github.com/RIOT-OS/RIOT/issues/13444

False Positives. Due to the limitations of static analysis, PASAN reported 12, 1, and 6 false positive bugs in ArduPilot, RIOT, and rusEFI, respectively. rusEFI has six transactions reported as concurrently executable code because employed points-to analysis [76] treats their locks to be different. In fact, these locks are the alias of the same lock. For ArduPilot and RIOT, PASAN reported two and one incorrect concurrency bugs, respectively, due to inaccurate transaction span extractions. We also found that one false positive case was reported because it did not require waiting for the job completion after device initialization. Specifically, `LSM9DS0`, a magnetometer of ArduPilot reads sensor values iteratively without requesting a processing job in the device driver. `LSM9DS0` was mistakenly reported due to I2C attached requiring writing accesses to control the I2C bus. In this case, the peripheral's internal state machine is tolerant to potentially buggy concurrent accesses, although PASAN correctly reports this as potential concurrency bugs based on our detection algorithm. Our manual verification did not reveal any more false alarms. We discuss about factors causing false positives in Section 6.

5.4 Concurrency Bug Detection Capability Comparison

We compare PASAN with the existing concurrency bug detection tools to show its effectiveness as summarized in Table 6. Our selection of the existing tools was guided by the following criteria. First, we focus on the comparison with static analysis tools. This is because dynamic analysis-based approaches [54, 55, 62, 67, 73, 83, 84] require dynamic analysis frameworks, which are not generically applicable to embedded systems except for only a few boards [41, 51, 53]. Second, we do not consider the tools requiring non-trivial manual efforts such as theoretical algorithms [60, 66] or manual code instrumentation [36, 65]. Finally, we consider the static analysis tools that are available to use for uncovering concurrency bugs with transaction corruption[7]. As such, we chose Flawfinder [20], Polyspace [24], and Coverity [15]. Flawfinder performs concurrency analysis for generic C/C++ code independent of compilers and target boards. Polyspace claims that they cover various real embedded systems such as Nissan car and aircraft autopilot [24, 25]. Coverity also claims to support automotive embedded systems while supporting embedded system compilers [30].

Table 6 shows the number of true concurrency bugs *only with transaction corruption*, and the number of any types of concurrency bugs reported by each tool. We found that Flawfinder, Polyspace, and Coverity cannot find any concurrency bug with transaction corruption. More specifically, Flawfinder found 265 *conventional* concurrency bugs (e.g.,

[7] For example, a trial version of CodeSonar [13] does not support academic evaluation; Mthread add-on is working on porting to its recent main framework [20]; Infer [21] does not support embedded system code since it ignores compilation commands for embedded systems.

Table 6: Summary of the concurrency bug detection performance of PASAN in comparison with existing works. T: # of true concurrency bugs *only with* transaction corruption, A: # of all reported concurrency bugs of any types without manually verifying their correctness.

Target	PASAN		Flawfinder [19]		Polyspace [24]		Coverity [15]	
Firmware	T	A	T	A	T	A	T	A
ArduPilot	8	20	0	247	0	0	0	0
RaceFlight	0	0	0	0	0	0	0	0
RIOT	8	9	0	9	0	1	0	0
Contiki	0	0	0	3	0	0	0	0
TS100	1	1	0	0	0	0	0	0
grbl	0	0	0	0	0	0	0	0
rusEFI	0	6	0	6	0	0	0	0

concurrent file object accesses); Polyspace found one concurrency bug caused by a global variable in RIOT; Coverity found zero concurrency bug although Coverity found the other types of bugs (e.g., integer overflow). Overall, as shown in Table 6, unlike PASAN, the existing tools cannot detect concurrency bugs caused by peripheral access transactions.

5.5 Case Study I: SD Card Data Corruption

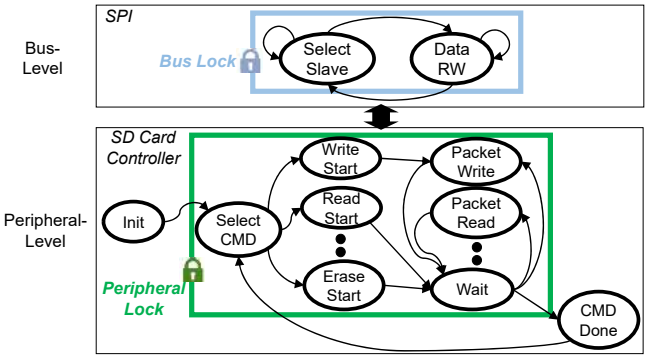

Figure 8: Simplified example of two-layered state machines of SPI and SD card controller.

RIOT [28] supports a variety of peripherals on diverse embedded systems. One of the supported peripherals is an SD card controller. Due to the limited number of I/O ports in embedded systems, an SD card controller is frequently attached to an SPI bus which may already be connected with other peripherals. We note that RIOT is designed to be a generic RTOS with a variety of interface options. Unfortunately, because of design flaws in the exclusive access protection, it is possible to exploit the control interface and access the controller directly/indirectly. As a result, a concurrency bug could potentially lead to data loss or corruption such as a missing SD card access and undesired data transfer to the SD card.

PASAN's analysis of the existing lock objects and lock spans corresponding to the controller has revealed two issues: there is no bus lock for protecting the state machine of SPI, and there is no peripheral lock spanning the whole transaction with the controller.

Missing Bus Lock on an SPI. As shown in the *bus-level* box of Figure 8, the SPI takes two states for the data transfer: (1) select a slave device among the attached peripherals, and (2) perform data read/write operations with the peripheral. As such, a concurrency bug can be found by checking whether there is a lock spanning from (1) to (2). Missing locks can cause the transferred data to be corrupted or data to be transferred to different devices unless both (1) and (2) are performed atomically. In our analysis, PASAN did not find a lock in either of the two states of the tested embedded platform revealing its vulnerability to potential attacks.

Missing Peripheral Lock for an SD Card Controller. The embedded system needs to perform a set of transactions with the controller to operate correctly. Such transactions are represented through a state machine shown in the *peripheral-level* box of Figure 8. We note that each transaction starts from `Select CMD` and ends at `CMD Done`. Hence, to guarantee the correct operation of the controller, the state machine transitions from `Select CMD` to `CMD Done` must be secured atomically by a lock. However, we found no lock spanning the state machine transitions. This means that concurrent accesses to the SD card controller may cause unexpected problems (e.g., data loss or corruption). Recently RIOT developers have applied a patch to enforce a *Bus Lock* as shown in Figure 8. However, the concurrency bug cannot be eliminated completely without enforcing the *Peripheral Lock* along with the *Bus Lock*.

Real-World Attack Scenario. Embedded systems used in IoT/CPS devices store various critical information including secret keys (e.g., passwords) and data logs (e.g., object approaching detection and mission execution orders facilitating movement between two waypoints). However, our experiments show that the concurrency bugs at both bus and peripheral levels can result into corruption of such information. To exploit these concurrency bugs, we configured our experimental embedded system on a BluePill [12] board with an SD card adapter connected through an SPI interface [22] to run four threads recording secret data (that is set as `PASSWORD`) continuously. When a concurrency bug was triggered in the middle of a data store operation by enabling concurrent accesses of multiple threads to the single SPI, we observed two cases with exploitable patterns. In the first case, one or more characters out of the eight characters of `PASSWORD` would be missing resulting into words such as `ASSWORD`. In the second case, the words from different threads would interleave with each other resulting into words such as `PAPASSWORDSSWORD`. We note that while the first case happens only when SPI bus-level locks are missing, the second case happens when any of the bus-level or peripheral-level locks are missing. Once such data corruption or loss happens, legitimate users may be prevented from accessing their embedded systems. In another example, the corruption may damage or even lose evidence for investigation if the entered data is log/forensic data.

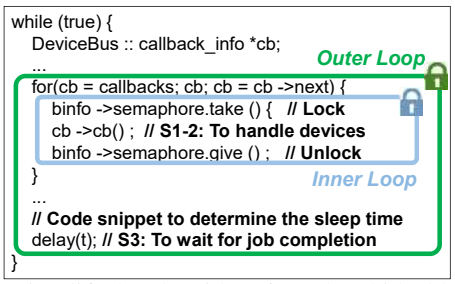

Figure 9: Simplified code with enforced and ideal lock spans for multiple devices.

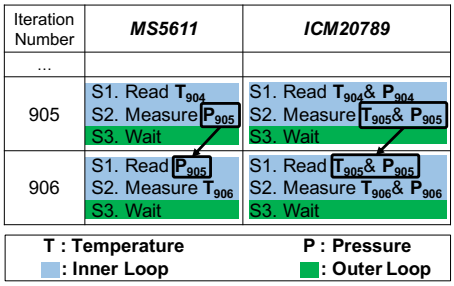

Figure 10: Iterative state machine transitions and operations for both sensors.

5.6 Case Study II: Sensor Value Corruption

An RAV is controlled by a remote control interface such as MAVLink [5]. This interface is known to be insecure [59, 71] because it does not employ fundamental network security features of encryption and authentication due to its computational constraints and limited hardware resources. Surprisingly, we found that this remote interface also allows direct access to I2C. As a result, anyone can potentially send instructions to any peripheral attached to I2C via MAVLink [18]. In fact, an RAV platform employs multiple sensors to accurately measure the physical state which is critical for its safe operation. Specifically, for controlling movements along vertical axis, an RAV employing ArduPilot [11] measures various physical states including the altitude (measured by barometers such as `MS5611`) and the three dimensional angles and accelerations (measured by an inertia sensor such as `MPU6000`). Hence, the corrupted altitude or angle values can cause sudden vertical movements or loss of the angular control of the vehicle, which may eventually lead to a crash. Here, we focus on the altitude corruption case.

Figure 9 shows a pseudo code corresponding to the device driver of a peripheral. This code has two-layered nested loops denoted as inner and outer loops. Those loops (outer + inner) are iteratively executed with the following three states as described in Figure 10.

- S1 (`read`): In this state, read the sensor value whose measurement was scheduled in the previous iteration (e.g., a sensor value from `MS5611` is read at Iteration 905. This value was scheduled to be measured at Iteration 904).
- S2 (`measure`): In this state, schedule a command to measure sensor value(s) for the next iteration (e.g., a sensor value from `MS5611` scheduled to be measured at Iteration 905 will be read at Iteration 906).
- S3 (`wait`): In this state, sleep to wait for job completion before the next iteration.

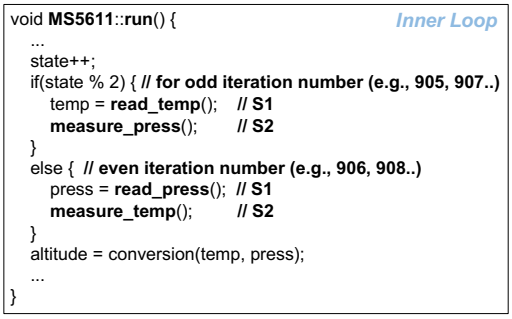

Figure 11: Simplified `MS5611` device handler.

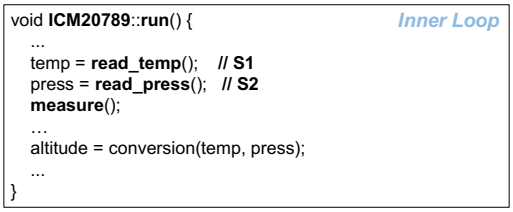

Figure 12: Simplified `ICM20789` device handler.

While the operations corresponding to the `read` and `measure` states are performed in the inner loop, those corresponding to the `wait` state are performed in the outer loop. For example, in ArduPilot [11], we found two barometers, `MS5611` and `ICM20789`, attached to I2C. These barometers are widely used to calculate the altitude using the pressure and temperature measurements. The code for `MS5611` and `ICM20789` are presented in Figure 11 and Figure 12 respectively. Specifically and interestingly, `ICM20789` reads both pressure and temperature values at each iteration and schedules their measurements for the next iteration. In contrast, `MS5611` reads one of the pressure and temperature measurements in one iteration, and the other one in the next iteration.

In this case, PASAN found that while *Bus Lock* is enforced, *Peripheral Lock* is only *partially* enforced. Specifically, as shown in Figure 9, since the existing lock does not cover the code corresponding to the `wait` state, a different transaction can execute in a different thread during the *wait* state of the ongoing transaction. As such, both barometers might map the temperature measurement to the pressure variable and vice versa, or have sensor values corrupted due to the concurrent access to these sensors from the remote control interface, e.g., MAVLink. As a fix, each driver should employ its own lock to protect the transaction with its sensor, and the remote control interface needs to respect these peripheral locks too.

Real-world Attack Scenario. As we mentioned earlier, the remote communication interface (i.e., MAVLink in this case) is insecure, but allows interfaced users to directly access I2C or SPI. As such, if an attacker abuses the insecure remote

interface and exploits this concurrency bug, it can corrupt the measured sensor values. We experimented with two different sensors measuring different states and attached them to I2C: `MS5611` and `MPU6000` attached to Pixhawk 1 [23] that is part of the 3DR IRIS+ RAV [3]. In the case of `MS5611` (a barometer), we launched a denial-of-service-like attack via MAVLink by alternately issuing temperature and pressure reading commands while the device was in the middle of executing one of measurement transactions. When a concurrency bug is triggered, `MS5611` fails to complete the ongoing measuring transaction; consequently, `MS5611` reports an abnormal altitude value. For example, if the current altitude is five meters, it generates a corrupted value (ranging from -3,200 to 3,200 meters) and records it in the flight log. In our experiments, the absolute values of the corrupted measurements were always larger than 200 meters. Hence, this attack led to corrupted altitude measurements and caused mission failures by triggering "safe landing" at an unexpected location.

We also carried out a similar attack targeting `MPU6000` (which is used as an accelerometer and gyroscope). The concurrency bug exploitation causes `MPU6000` to generate corrupted acceleration and gyro values. In our experiments, `MPU6000` produced the three-axe acceleration values in the range between -120 and -160 m/s/s, where the normal values should have been between -10.0 and 10.0 m/s/s. Due to abnormally large acceleration values, this exploitation caused the RAV to trigger safe landing or even crash due to severe control instability.

We believe that the demonstrated concurrency bug exploitation is a meaningful attack vector because of its stealthiness into ArduPilot (and other autonomous vehicle control software) and RTOSes. ArduPilot is one of the most popular RAV control programs [56, 57]. As such, its source code is widely adopted by various RAV vendors, such as Intel Aero [4], Parrot [6] and 3DR [3]. To support debugging and crash investigation, ArduPilot also provides plentiful logging information including those corresponding to sensor and control states, and mission tasks. However, ArduPilot does not provide any meaningful network system logging that requires the support of full-fledged operating systems (e.g., Linux). Instead, ArduPilot uses a lightweight RTOS (i.e., ChibiOS) without such features. Furthermore, ArduPilot's logging system does not record any information on MMIO accesses including I2C and SPI. Besides, due to their nondeterminism, concurrency bugs are tricky to debug even in the environments with powerful debugging tools [55]. Overall, due to the absence of MMIO access and network traces, and difficulty in concurrency bug debugging, concurrency bug exploitation is a meaningful attack vector. It will remain an attractive attack vector (from attackers' perspective) – even more so after the improvement of the MAVLink protocol security in the (near) future.

Why peripheral access concurrency bugs are complex? While PASAN detects the missing *Peripheral Lock*, cautious readers might have found out that while a peripheral lock within `ICM20789` driver protects its transaction to the sensor, a similar peripheral lock within `MS5611` driver still fails to protect its transaction. Due to the unique code structure within the `MS5611` driver, its de facto transaction with its sensor spans into two iterations within the outer loop, e.g., calling the driver twice, which is the only way to get both temperature and pressure measurements to fulfill the computation of altitude.

Currently, PASAN extracts transaction spans covered by a single lock span. If one transaction involves two outer iterations of the loop as in `MS5611`'s transaction (i.e., subset transaction span case introduced in Section 5.2), PASAN *partially* covers one outer iteration and could not extend to multiple iterations because the driver itself does not implement the whole transaction but relies on callees to accomplish it.

6 Discussion

Limitations Inherited from Existing Static Analysis Employed. PASAN requires call graphs to generate possible thread call stacks (e.g., "Thread 1 and 2 Call Stack" in Figure 6). It also needs to identify aliases of function pointers for indirect calls, lock objects, and accessed MMIO addresses. As such, PASAN utilizes points-to analysis [76] for identifying call graphs (including indirect function calls) and alias variables. The current tools that PASAN relies on have two well-known limitations in tracking aliases, which can cause inaccuracy in our concurrency bug detection.

One of the common limitations of points-to analysis is to over-approximately resolve possible pointers [37] by encompassing infeasible function calls or aliases. This may result in false positives in identifying aliases. Specifically, points-to analysis may mistakenly identify different MMIO access variables as identical aliases (causing false positives in concurrency bug detection), different lock object variables as identical aliases (causing false negatives), and infeasible indirect call targets (causing false positives). We did not observe such inaccurate results in our experiments.

The other common limitation of points-to analysis is failure in tracking aliases to mitigate state explosion of points-to analysis [52, 64]. Specifically, points-to analysis can fail to identify the aliases of MMIO access variables (causing false negatives in concurrency bug detection) and aliases of lock object variables (causing false positives). Furthermore, missing indirect call targets (e.g., device drivers) can cause PASAN to miss transaction spans (causing false negatives).

Moreover, lockset analysis cannot take into consideration the timeout locks that are automatically unlocked after a given time at run time to prevent deadlocks. However, it is challenging for static analyses to estimate the lock spans affected by the timing behavior of the timeout locks. Hence, PASAN conservatively considers the timeout locks as typical locks. This might cause false negatives in concurrency bug detection

although we did not observe any in our experiments.

To alleviate the above limitations, we could employ either (1) more advanced static analysis works [63, 86] that could reduce false positives in alias identification or improve points-to analysis algorithm to reduce false negatives in alias identification as DR. CHECKER [64] pointed out, or (2) dynamic analysis with peripheral modeling as proposed in the prior work [41, 51]. Especially, dynamic analysis can overcome limitations in handling special lock operations (e.g., timeout locking) with emulated boards [41, 51]. However, dynamic approaches may not be directly applicable because they cannot model various peripheral devices. Furthermore, they suffer from a limited analysis coverage as they can only analyze executed code. Further improvement in this direction will be our future work.

Using Incorrectly Inferred Transaction Spans. Achieving perfect accuracy on the inference of transaction spans is not the main goal for our project. However, we point out that even the incorrectly inferred transaction spans can be useful. There are three categories of such transaction spans: subset, superset, and mixed, as explained in Section 5.2. Thanks to these transaction spans, PASAN did *not* miss the concurrency bugs in the `MS5611` case (Section 5.6). On the flip side, we did have several false positive cases caused by inaccurate transaction extraction.

Validity of Protection for All Peripheral Devices. We cannot ascertain whether a peripheral device is tolerant to buggy concurrent accesses without manual verification due to its black-box nature. However, we observe that device drivers often perform read-only accesses to the concurrency-tolerant peripherals. Based on this observation, we employ "write-access-inclusion" heuristic in PASAN to exclude those read-only accesses, which helps remove (false-positive) transactions of those concurrency-tolerant peripherals. As a result, we observed only one false-positive case with `LSM9DS0` (detailes in Section 5.3) due to the concurrency bug-tolerant peripheral.

Validity of "Write-Access-Inclusion" Heuristic. PASAN analyzes all transactions involving at least one write access to an MMIO address, which is the most common case based on our experience. We found that including read-only transactions would cause many false positives because the status of some peripherals (e.g., timer and USART) are concurrency-tolerant and hence not vulnerable to unprotected concurrent reads as they maintain their own internal states. Instead, this heuristic can introduce false negatives by missing read-only transactions that are not tolerant to concurrency bugs.

Limitation in Handling Individual Interrupts. PASAN does not support individual interrupt requests (IRQ) as it would require non-trivial manual efforts to map into IRQs and their corresponding bit masks which enable/disable individual interrupts. Also, one mask can be related to multiple IRQs [51]. Furthermore, some interrupts are enabled/disabled dynamically. These challenges can only be addressed through a dynamic analysis tool with access to the target device. PASAN, as a static analysis tool, cannot support individual IRQs, and may lead to false-positives. Fortunately, we have manually confirmed that, in our evaluation, no false positive was caused by individual IRQs.

Binary Firmware Support. While we evaluate PASAN on the source code of firmware in this paper, the fundamental mechanism may become applicable to binary firmware, after addressing the following technical challenges. We identify two specific challenges in obtaining necessary inputs from binary firmware: (1) A binary firmware needs to be lifted into compatible LLVM bitcode. Although there are multiple approaches to doing this [27, 48, 80, 81], their lifting results are either incompatible or immature for embedded systems. For example, the results for ARM 32bit architecture (which is dominant on embedded systems) are not mature enough[8]. (2) PASAN must identify key functions, such as locks, multi-threads and multi-process management functions. If a firmware is stripped, this information needs to be supplemented by other sources such as pattern-based function identification [34], and binary-based code similarity search [49] to identify these key functions.

Automatic Lock Enhancement. Since PASAN detects invalid concurrency lock behavior, it is a promising idea to use this information to correct or enhance locks automatically. Such an automated approach demands very high accuracy on the extracted lock spans. Otherwise, it may introduce unstable behavior. We reserve this direction as our future work after we achieve higher accuracy in lock span extraction.

7 Related Work

Concurrency Bug Detection. The concurrency detection techniques in the prior art can be broadly classified based on their analysis methodologies which include static [13, 15, 19–21, 24, 32, 33, 40, 50, 69, 79], dynamic [67, 83, 84], and hybrid (static and dynamic) analysis [54, 55, 62, 67, 73]. There are also some algorithmic [60, 66] and manual detection techniques [32, 36, 42, 65].

Prior static analysis-based schemes are limited to analyzing single memory objects without considering transactions for MMIO accesses. Hence, unlike PASAN, they cannot discover transaction- and address-range-aware concurrency bugs. The dynamic analysis-based approaches are applicable to binary-only programs, they require the aid of specialized hardware, and they handle only limited types of concurrency bugs. Researchers have also proposed hybrid analysis approaches to perform dynamic analysis on top of the static analysis results.

[8] Out of the four cited tools, only RetDec [27] and mctoll [81] support ARM 32bit architecture. In our experience, RetDec generates severely incorrect control flow results and mctoll generates empty bitcode.

However, these hybrid analysis approaches require direct access to the target peripheral devices. We note that it is not practical to find concurrency bugs individually in each embedded platform. Development of theoretical algorithms and manual techniques require non-trivial efforts and instrumentation in identifying transactions of peripherals. In summary, unlike PASAN which discovers concurrency issues for peripherals, the scope of the approaches in the prior art is limited to memory object-level concurrency bugs.

Device Driver Vulnerability Detection. Vulnerabilities hidden in the device driver have been discovered statically [58, 64, 70] as well as dynamically [7, 44, 72, 75, 77, 85]. Traditionally, static analysis relied on symbolic execution [58, 70] to find bugs and vulnerabilities. In a more recent work, DR.CHECKER [64] leveraged compiler-level program analysis (e.g., points-to analysis and data flow analysis) to find bugs. Moreover, Charm [77] carried out dynamic analysis of device drivers in mobile systems. PeriScope [75] wisely hooked into the page fault handler in the kernel to detect vulnerabilities while fuzzing the Wi-Fi drivers. While vUSBf [72] fuzzed the USB device drivers, Syzkaller [7] integrated multiple kernel fuzzing systems (such as DIFUZE [44]) to fuzz the kernel functionality including kernel drivers. However, none of these vulnerability detection approaches can discover bus- and peripheral-level concurrency issues.

Embedded Firmware Analysis Framework. Both, static [43,45,46,74] and dynamic [35,38,39,41,51,53,82,87], approaches have been employed for analyzing embedded platforms. Following the static analysis approach, Costin et al. [45], Firmalice [74], and PIE [43] found several network security vulnerabilities and imperfect API implementations. FIE [46] was specifically designed to find memory corruption bugs. To discover bugs such as memory corruption or program crash, IOTFuzzer [39] was designed to fuzz the bare metal Internet of Things (IoT) devices. To enable instrumentation and monitoring, schemes in the existing literature rely on device emulators [35] or specific hardware interfaces. Moreover, researchers have also analyzed a limited number of platforms (e.g., Linux-based platforms) on the emulated environments which are already well-developed in emulator development communities [38, 87]. To overcome full emulation requirements, some recent works have been proposed [41, 51, 53]. While Pretender [53] still requires the original hardware to record the MMIO's activity, both P2IM [51] and Halucinator [41] cannot correctly handle some hardware devices such as DMA. Finally, none of these studies emulated any device attached to the I/O which limits the coverage of the analysis results. Overall, unlike PASAN, the dynamic analysis approaches in the prior art are limited by the requirement of significant engineering efforts in generating analysis environments with actual boards and specialized hardware (e.g., GDB, Bluetooth, or client devices). Furthermore, both static and dynamic analysis approaches focus on program crash, memory corruption and known security threats.

8 Conclusion

Concurrency bugs in embedded platforms (e.g., RAVs) may cause a variety of safety and security issues (i.e., from physical system failure to security critical data corruption). Unfortunately, detection of concurrency bugs is especially challenging in embedded platforms due to the intricate interplay of the bus-level and peripheral-level state machines. In this paper, we propose PASAN, a device-agnostic static analysis-based approach which addresses this challenge. PASAN detects peripheral access concurrency bugs automatically by pursuing a *transaction-aware* and *address-range-aware* strategy. We validate the capabilities of PASAN by evaluating it on seven real-world embedded platforms, and discover a total of 17 concurrency bugs in three different platforms. We have reported these findings to the corresponding parties.

Acknowledgment

We thank the anonymous reviewers for their valuable comments. This work was supported in part by ONR under Grants N00014-20-1-2128 and N00014-17-1-2045. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

References

[1] Dirty cow (cve-2016-5195), 2016. https://dirtycow.ninja.

[2] *grbl — An open source, embedded, high performance g-code-parser and CNC milling controller ported to stm32f4*, 2016. https://github.com/deadsy/grbl_stm32f4.

[3] *3DR IRIS+*, 2018. https://3dr.com/support/articles/iris.

[4] *Intel Aero*, 2018. https://software.intel.com/en-us/aero.

[5] *MAVLink Micro Air Vehicle Communication Protocol*, 2018. https://mavlink.io.

[6] *Parrot Bebop2*, 2018. https://www.parrot.com/global/drones/parrot-bebop-2.

[7] *syzkaller - linux syscall fuzzer*, 2018. https://github.com/google/syzkaller.

[8] *CMSIS System View Description*, 2019. http://www.keil.com/pack/doc/CMSIS/SVD/html/index.html.

[9] Cve-2019-6471, 2019. https://kb.isc.org/docs/cve-2019-6471.

[10] List of 862 race conditions in the cve database, 2019. http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=race+condition.

[11] *ArduPilot*, 2020. http://ardupilot.org.

[12] *Blue Pill — STM32F103C8T6*, 2020. https://stm32-base.org/boards/STM32F103C8T6-Blue-Pill.html.

[13] Codesonar c/c++ - sast when safety and security matter, 2020. https://www.grammatech.com/codesonar-cc.

[14] *Contiki-NG: The OS for Next Generation IoT Devices*, 2020. https://github.com/contiki-ng/contiki-ng.

[15] Coverity scan - static analysis, 2020. https://scan.coverity.com.

[16] Cve-2020-0030, 2020. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-0030.

[17] Cve-2020-3941, 2020. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-3941.

[18] *Direct Comms with SPI/I2C bus*, 2020. https://ardupilot.github.io/MAVProxy/html/modules/devop.html.

[19] Flawfinder, 2020. https://dwheeler.com/flawfinder.

[20] frama-c: Software analyzers, 2020. https://frama-c.com.

[21] Infer - a static analysis tool for java, c++, objective-c, and c., 2020. https://fbinfer.com.

[22] *Micro SD Card Module Mini TF Card Adapter with SPI Interface Driver Module*, 2020. https://www.amazon.com/Geekstory-Module-Adapter-Interface-Arduino/dp/B07X478BPL.

[23] *Pixhawk 1 Flight Controller*, 2020. https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk.html.

[24] Polyspace: Automated static code analysis using formal methods for c/c++ and ada, 2020. https://www.mathworks.com/products/polyspace.

[25] Polyspace bug finder reference, 2020. https://www.mathworks.com/help/pdf_doc/bugfinder/bugfinder_ref.pdf.

[26] *RaceFlight — Performance, stability and ease of use for STM32F4 and more*, 2020. https://github.com/rs2k/raceflight.

[27] *RetDec: a retargetable machine-code decompiler based on LLVM*, 2020. https://github.com/avast/retdec.

[28] *RIOT — The friendly OS for IoT*, 2020. https://www.riot-os.org.

[29] *rusEFI — a GPL open source engine control unit*, 2020. https://rusefi.com.

[30] Sast-coverity-datasheet, 2020. https://www.synopsys.com/content/dam/synopsys/sig-assets/datasheets/SAST-Coverity-datasheet.pdf.

[31] *TS100 — soldering iron firmware*, 2020. https://github.com/Ralim/ts100.

[32] Verifast, 2020. https://github.com/verifast/verifast.

[33] Jia-Ju Bai, Yu-Ping Wang, Julia Lawall, and Shi-Min Hu. Dsac: Effective static analysis of sleep-in-atomic-context bugs in kernel modules. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, 2018.

[34] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. Byteweight: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security)*, 2014.

[35] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track (ATC)*, 2005.

[36] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[37] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[38] Daming D Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for linux-based embedded firmware. In *Proceedings of the 25th Annual Symposium on Network and Distributed System Security (NDSS)*, 2016.

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

[40] Qiu-Liang Chen, Jia-Ju Bai, Zu-Ming Jiang, Julia Lawall, and Shi-Min Hu. Detecting data races caused by inconsistent lock protection in device drivers. In *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019.

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

[42] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *Proceedings of the International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2009.

[43] Lucian Cojocar, Jonas Zaddach, Roel Verdult, Herbert Bos, Aurélien Francillon, and Davide Balzarotti. Pie: parser identification in embedded systems. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, 2015.

[44] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

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

[46] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of the 22nd USENIX Security Symposium (USENIX Security)*, 2013.

[47] Peng Di and Yulei Sui. Accelerating dynamic data race detection using static thread interference analysis. In *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*, 2016.

[48] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (IEEE S&P)*, 2020.

[49] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019.

[50] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

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

[52] Eléonore Goblé. Taint analysis for automotive safety using the llvm compiler infrastructure. 2019.

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

[54] Dae R Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (IEEE S&P)*, 2019.

[55] Baris Kasikci, Cristian Zamfir, and George Candea. Racemob: Crowdsourced data race detection. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[56] Taegyu Kim, Chung Hwan Kim, Altay Ozen, Fan Fei, Zhan Tu, Xiangyu Zhang, Xinyan Deng, Dave (Jing) Tian, , and Dongyan Xu. From control model to program: Investigating robotic aerial vehicle accidents with mayday. In *Proceedings of 29th USENIX Security Symposium (USENIX Security)*, 2020.

[57] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. Rvfuzzer: Finding input validation bugs in robotic vehicles through control-guided testing. In *Proceedings of 28th USENIX Security Symposium (USENIX Security)*, 2019.

[58] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with ddt. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*, 2010.

[59] Y. Kwon, J. Yu, B. Cho, Y. Eun, and K. Park. Empirical analysis of mavlink protocol vulnerability for attacking unmanned aerial vehicles. *IEEE Access*, 6:43203–43212, 2018.

[60] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. 2019.

[61] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.

[62] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. Lockdoc: Trace-based analysis of locking in the linux kernel. In *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019.

[63] Kangjie Lu and Hong Hu. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[64] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. Dr.checker: A soundy analysis for linux kernel drivers. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security)*, 2017.

[65] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[66] Friedemann Mattern et al. Virtual time and global states of distributed systems. In *Proceedings of the Workshop on Parallel and Distributed Algorithms*, 1988.

[67] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[68] Esko Nuutila and Eljas Soisalon-Soininen. On finding the strongly connected components in a directed graph. *Information Processing Letters*, 49(1):9–14, 1994.

[69] Peter O'Hearn. Separation logic. *Communications of the ACM*, 62(2):86–95, 2019.

[70] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. Symdrive: testing drivers without devices. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[71] Nils Rodday. Hacking a professional drone. *Blackhat ASIA*, 2016.

[72] Sergej Schumilo, Ralf Spenneberg, and Hendrik Schwartke. Don't trust your usb! how to find bugs in usb device drivers. *Blackhat Europe*, 2014.

[73] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications (WBIA)*, 2009.

[74] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSS)*, 2015.

[75] 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 *Proceedings of the 28th Annual Symposium on Network and Distributed System Security (NDSS)*, 2019.

[76] Yulei Sui and Jingling Xue. SVF: Interprocedural Static Value-flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*, 2016.

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

[78] V. Vojdani, K. Apinis, V. Rõtov, H. Seidl, V. Vene, and R. Vogler. Static race detection for device drivers: The goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[79] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Relay: static race detection on millions of lines of code. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, 2007.

[80] Shuai Wang, Pei Wang, and Dinghao Wu. Reassembleable disassembling. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, 2015.

[81] S Bharadwaj Yadavalli and Aaron Smith. Raising binaries to llvm ir with mctoll. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, 2019.

[82] Jonas Zaddach, Luca Bruno, Aurelien Francillon, Davide Balzarotti, et al. Avatar: A framework to support dynamic security analysis of embedded systems' firmwares. In *Proceedings of the 23rd Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[83] Qiang Zeng, Dinghao Wu, and Peng Liu. Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[84] Tong Zhang, Changhee Jung, and Dongyoon Lee. Prorace: Practical data race detection for production use. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

[85] Tong Zhang, Dongyoon Lee, and Changhee Jung. Txrace: Efficient data race detection using commodity hardware transactional memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[86] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. Pex: A permission check analysis framework for linux kernel. In *Proceedings of 28th USENIX Security Symposium (USENIX Security)*, 2019.

[87] 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 *Proceedings of the 28th USENIX Security Symposium (USENIX Security)*, 2019.