Retcon: Live Updates for Embedded Event-Driven Applications

Jean-Luc Watson
University of California, Berkeley
jlw@berkeley.edu

Saharsh Agrawal
University of California, Berkeley
saharshagrawal@berkeley.edu

Ryan Tsang
University of California, Berkeley
r_tsang@berkeley.edu

Sherry Luo
University of California, Berkeley
sherry.z.luo123@berkeley.edu

Raluca Ada Popa
University of California, Berkeley
raluca.popa@berkeley.edu

Prabal Dutta
University of California, Berkeley
prabal@berkeley.edu

Abstract

Embedded systems are deeply integrated into critical applications but, despite their importance, lack an effective means to apply over-the-air software patches without significant downtime. Standard mechanisms for firmware updates require device reboots that wipe important in-memory state. Prior efforts have proposed "live" updates to address this problem, applying patches to an embedded application without a reset, but they tackle a limited set of applications or propose a clean-slate design. In this paper, we present Retcon, a live update toolchain for embedded systems that supports a familiar event-driven programming model and does not require application code changes. Retcon leverages static analysis at compile time to determine when it will be safe to update a device. To find safe update points in the presence of complex asynchronous behavior, we define a novel system state, *asynchronous quiescence*, in which an update can be applied. We evaluate Retcon on a set of embedded event-driven applications – a dual-chamber pacemaker model, a programmable logic controller runtime, an artificial pancreas system, and a sensing node – and demonstrate Retcon's ability to make low-overhead updates in less than one millisecond.

1 Introduction

Low-power embedded systems are used in a variety of high-availability applications, particularly in the medical and industrial automation fields [52, 59]. Although embedded systems often perform simple tasks, they form a quickly expanding (and aging) infrastructure that requires frequent software updates to continue executing securely and safely. In particular, increased connectivity of embedded devices has exposed them to attack vectors that require urgent security patches. For example, a 465,000-strong pacemaker recall [14] was required after a vulnerability in their communication stack was found, and a similar attack affecting pacemakers using Bluetooth Low Energy [58] occured 2020. Breaches can also affect critical infrastructure: the Indian power grid saw compromises of their operations networks in early 2021 through Internet-connected cameras [16].

Downtime from applying patches in highly-deployed systems can have significant impact [35]. Existing device firmware update (DFU) mechanisms require a device to reboot into a new firmware version after an update [2, 6, 26, 51]. Programmable logic controllers (PLCs), used for low-level control in automated industrial equipment [8], must therefore shut down the physical systems (e.g. electric motors or manipulators) they control [5, 39], while medical devices often require an in-person visit for urgent security patches [14, 29]. After a reboot, the update has wiped any useful in-memory state accumulated by the previous version.

A growing body of research has focused on providing a *live update* capability, in which a device is updated seamlessly from one patch to the next, without incurring any downtime or state loss. Existing approaches remain limited: frameworks for general-purpose computing rely on expensive runtime monitoring [11, 17, 36, 43], and some prior efforts for embedded systems avoid state transfer entirely [38, 41]. In particular, applications are often restricted to simple cyclic execution [15, 47, 56] where updates always occur at the start of a read-compute-write cycle. While this simplifies update analysis, it also restricts updates to a small application subset.

A much broader set of embedded applications operate in an *event-driven* model, executing in response to asynchronous events (e.g. a timer expiration, or reception of network data). Event-driven code is in widespread use in a slew of embedded operating systems (OSes) developed over the past two decades [2, 6, 7, 13, 31, 33]. Unfortunately, no current live update mechanisms can operate in this setting without a clean-slate OS design [13, 18, 21, 56] – a serious barrier to adoption, where patches on legacy code are needed most of all. In this paper, we explore the question: *how do we design a live update mechanism that supports event-driven applications, without requiring code modification, while enabling near-instantaneous updates on constrained embedded platforms?*

To answer this question, we introduce Retcon, a live update toolchain for embedded systems that supports unmodified event-driven applications. Retcon consists of two major components: a static analysis toolchain that generates update instructions at compile time, and a lightweight runtime executing on the target embedded device. Retcon determines both *when* it is safe to apply a patch without disrupting the device and *how* to migrate state to the new version. In doing so, it must support event-based applications that are significantly more complex than envisioned in prior work (Section 2), requiring new approaches in this setting to support efficient live updates, discussed below.

Control-flow static analysis. The targeted event-based applications consist of many handlers that execute in response to specific events. They will also routinely create asynchronous requests to the device OS that remain outstanding even while the application is idle. Handlers might receive events in any order, with no particular guarantees about when "in-flight" asynchronous operations will be completed. In larger applications, reasoning about event interleaving is error-prone at best, even for the original developer. Thus, a programmatic mechanism for detecting safe update points is necessary. Retcon solves this by contributing a more powerful static analysis model that forms a control-flow graph (CFG) of application behavior based on the application source code. It tracks

979-8-3503-6201-5/24/$31.00 ©2024 IEEE
DOI 10.1109/IPSN61024.2024.00015

relevant system state, including hardware configuration, and active asynchronous operations. To patch applications with minimal interruption, Retcon's primary design philosophy is to limit runtime overhead by moving analysis to a compile time phase that can reason over this CFG, which we discuss in more detail in Section 4.

Asynchronous quiescence. Prior work commonly assumes that when an application is "quiescent" and the processor is idle, the system can be safely updated [46]. Unfortunately, this is no longer the case in an event-driven system, since pending asynchronous operations might generate unexpected events during or after the update. For example, a running timer started by the original application might expire and trigger an actuator after an update, causing real-world injury. Retcon introduces a novel notion of *asynchronous quiescence* (Section 4.1): live updates are applied only when no tasks are executing *and* we know that existing in-flight asynchronous operations will not trigger events during or after the update. The latter condition can be satisfied if, for example, the new application would immediately cancel or reset any outstanding operations. We compare the CFGs of the original and patched applications to identify the desired behavior.

Detecting update points at runtime. Detecting asynchronous quiescence, and confirming that a system has reached a valid update point at runtime, requires significant inspection of application state that cannot be performed on the embedded target. To address this challenge, Retcon's toolchain calculates a set of simple *predicates* that the embedded runtime can evaluate over application state to verify a correct update point, without performing any analysis itself. Once a valid update point is detected, the runtime uses a matching set of *transfer instructions* to synchronize memory state between the application versions. We demonstrate that even for applications with many such predicates, they are processed with minimal overhead.

Implementation. We implement Retcon's static analysis toolchain and embedded runtime for the Zephyr RTOS [2], targeting an nRF9160 SiP embedded platform, and evaluate Retcon's ability to efficiently and safely perform live updates on four event-driven applications: a pacemaker controller, an Artificial Pancreas System (APS), a simple PLC runtime, and a sense-and-send application integrating a third-party MQTT library [9] (Section 6). Compared to standard update mechanisms, which require over 4 seconds for a full firmware flash and reboot, and 1.9 ms for an OS-based application swap, Retcon can apply updates live in less than a millisecond.

Such a capability enables a new generation of embedded systems to truly offer zero downtime by embracing a safe, continuous, and live update function.

2 Related Work

Systems for performing live updates have been heavily investigated as a way to avoid system downtime. A key early result by Gupta et al. [20] showed that determining the validity of a live update for general programs is adjacent to solving the halting problem, and thus undecidable. As a result, avoiding full generality by specifying additional constraints on the supported programming model, or requiring additional external information from applications, is a common feature of any live update mechanism [17, 43].

2.1 General-purpose live updates

One of the earliest dynamic update systems, Ksplice [4], targets small code-only security patches for Linux, installing code trampolines to jump into a patch from the original binary. Critically, this requires freezing execution, something which could affect the safety of embedded applications with real-world sensing and actuation tasks. Similarly, kGraft [50] routes application processes to original or updated code based on when they entered the kernel. Kup [28] uses a checkpoint mechanism to restore processes after an update, but incurs a restart.

Live update systems targeted at powerful server-class applications require significant runtime processing, such as stack unrolling [19, 27, 36, 42], although this allows applications to update at any point, rather than waiting for a quiescent state without active stack frames. ProteOS [17] relies on runtime memory tagging to track and transfer state, synchronously updating while suspending the system, resulting in a significant memory overhead (approximately 35%) ill-suited for a memory-constrained embedded device. Mvedsua [43], DynAMOS [37], POLUS [11], and Mx [23] each rely on multiversion execution, running the updated application in parallel during a swap-over period. With limited memory and compute capability on embedded devices, this approach would change application resource availability during an update; in contrast, we target near-instantaneous switch between versions without overlap. Ginseng [40] and Javelus [19] rely on lazy state transfer, interposing on accesses to memory in the updated process to copy state. However, this results in an indeterminate overhead after an update. A complex runtime increases system memory footprint and the interruption time during an update; Retcon pushes as much computation as possible to offline analysis, providing a device with a straightforward set of simple instructions at runtime.

Systems like Kitsune [22] or ProteOS [17] simplify the process of detecting update points by asking application developers to identify update points. In the event-driven model, however, this approach does not scale well. Event-triggered tasks can interleave in a complex manner based on when events arrive, resulting in a state machine that is difficult to completely explore manually, even for the code's author. Instead, we show that static analysis can automatically find events and system states that indicate valid update points in these applications.

A number of live update-adjacent systems have been proposed. Several virtual machines designed specifically for constrained embedded platforms [30, 44] can make application updates significantly easier, although still requiring an application restart. Neutron [12] reboots individual system components suffering from hardware faults while protecting precious memory from reinitialization, but doesn't apply updated software or identify safe update points. Retcon is complimentary to efforts efficiently distributing updates across wireless networks [24, 32, 49], focusing on the update once it arrives on-device.

2.2 Embedded live update systems

The need for software updates on high-availability embedded systems has motivated a significant amount of work on live updates. Table 1 compares Retcon's capabilities and programming model against prior work in the area.

Table 1: Summary of prior work in live update systems for embedded platforms. While existing systems rely on simplified cyclic programming models or require clean-slate component-based designs, Retcon supports whole-application updates in a broader event-driven programming model, and identifies correct, safe update points to begin executing a new version.

System	Unit of update	State transfer	Programming model	Condition for safe update point
DURTS [38], HERA [41]	Hotpatch	✗	Task	✗
Seifzadeh et al. [47]	Application	✗	Cyclic	Cycle end
Fischmeister et al. [15]	Component	✗	Cyclic	Cycle end
SOS [21]	Component	✗	Event-driven	✗
Contiki [13]	Component	✓	Event-driven	✗
FASA [53–56]	Component	✓	Cyclic	Cycle end
ELUS [18]	Component	✓	Task/Cyclic	Task quiescence
Retcon (this paper)	**Application**	✓	**Event-driven**	**Asynchronous quiescence**

Many existing systems have considered updating portions of applications, which constrains the scope of possible updates. DURTS [38] modifies function pointers to redirect applications to updated functionality, cannot perform more substantial updates that require changes to control flow. Similarly, HERA [41] uses ARM debugging hardware to dynamically insert patches, but updates are limited to "small, isolated, and featureless" [3] segments, and do not survive a reboot if it were to occur. Dynamic operating systems with a clean-slate modular designs, like SOS [21] or Contiki [13], support dynamic reconfiguration by loading individual application components, which are linked together and managed by the runtime. However, this requires rewriting applications to fit into these new frameworks. In contrast, Retcon focuses on whole application updates, allowing for significant internal changes like modified control flow, removing the risk of component incompatibilities, and allowing live updates to be performed on commonly-used RTOSes without a component-based architecture [2, 7, 31].

The ability for an application to transfer and retain important state during an update is critical for a live update capability. However, a significant number of prior systems ([15, 21, 38, 41, 47]) support only code changes without considering state transfer. Retcon leverages compile-time static analysis to compare application versions and determine which state to transfer at runtime.

Furthermore, prior work has focused on simple programming models that do not translate to more complex applications. For example, Seifzadeh et al. [47] and Fischmeister et al. [15] focus on a cyclic programming model, in which inputs are read at the beginning of each cycle, a static set of tasks is executed, and outputs are written. Updates in these systems without state transfer are completed simply by updating the tasks between cycles. In the event-driven programs that Retcon targets, there is no "end-of-cycle" event that triggers an update, and safe update points are instead application-dependent and require much more analysis.

In that vein, prior work can be readily distinguished from Retcon by the mechanisms used to determine when a safe update point has been reached. Some systems ignore update safety entirely – Contiki allows application developers to define an initialization function that can consume the prior version's state, but no analysis is made to determine a safe update point. The safety of an update is not guaranteed and developers must defensively program around cases where functionality may suddenly disappear from the system.

Most similar to Retcon are Wahler et al.'s works on FASA [53–56] and ELUS [18], supporting individual component-based updates. However, their safe update point analysis is limited to simple task quiescence. In the case of FASA, this point is at the end of a cycle, while ELUS simply waits for all tasks to be idle. However, as discussed in Section 1, this analysis is not sufficient when performing live updates for event-driven applications – ongoing asynchronous operations can persist across an update at an idle point and wreck havoc in the updated application. This is Retcon's primary contribution: defining a novel condition for detecting valid live-update points in event-driven code, *asynchronous quiescence*, that prior work has not addressed. Section 4 describes how we determine system states that satisfy this condition.

3 System Overview

Retcon's live update operates in two stages, detailed in Figure 1. The first stage creates an update payload, executing in parallel the application's normal build process. Payloads contain the patched binary and instructions for when and how to apply the update. The payload is then sent to Retcon's second stage, an update runtime executing within the target device's embedded OS. When the runtime detects a valid update point, the application is updated, state is transferred, and execution continues in the new application version.

Update payload generation. (Section 4) At compile-time, the updated code passes through a set of static analysis stages, where update instructions are generated from the program's behavior:

- Application source code is passed to a static analysis tool, which generates traces of program execution. We use the `clang` Static Analyzer [1], as it can operate directly on application source code without modification. When generating these execution traces, the analyzer generates symbolic constraints on application memory, allowing Retcon to identify specific state values that lead to different events.
- We combine the individual generated execution paths into a control flow graph (CFG) of the application (Section 4.2), where edges between basic blocks of application code are defined by the constraints on application state. For example, if a handler checks for a null input value, the CFG edge to the rest of the handler will require the input to be non-zero.

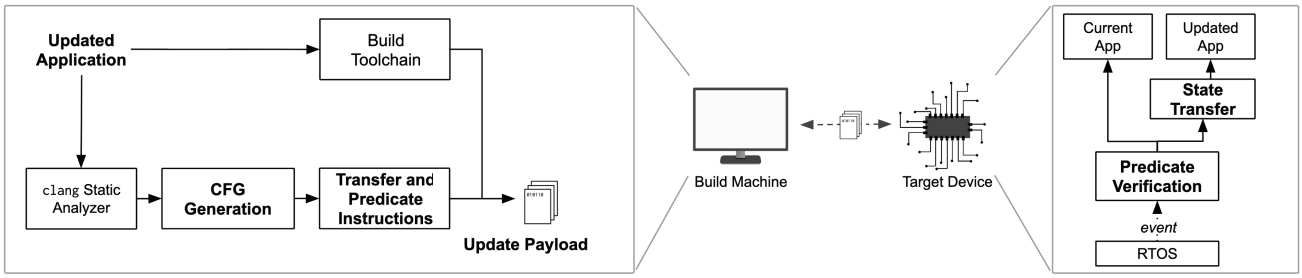

Figure 1: At compile time, Retcon's static program analysis generates predicates for safe update points and state transfer instructions, parallel to a normal build process. The update runtime on the target device, upon receiving the update payload, begins evaluating each predicate over system state. When the device reaches a safe update point that satisfies a predicate, relevant state is transferred and execution passes to the updated application.

- A set of update predicates, which the runtime uses to verify that the device has reached a safe update point, is generated by comparing the CFGs of the original and updated application versions (Section 4.4).
- Finally, a set of state transfer instructions is paired with each predicate, detailing how application state is transferred between versions. These instructions are generated automatically, but in the case of new unknown state, developer input is needed at compile time to define their values (Section 4.3).

Embedded update runtime. (Section 5) Once the update payload is received by the target device, Retcon's runtime loads the new application binary into an isolated memory segment in the background while the current application continues executing in the foreground. The runtime begins searching for a suitable update point, evaluating each of the provided predicates against system state. Once a predicate matches, the system can safely update. The runtime executes the state transfer instructions associated with the successful predicate, and hands control to the new version. Finally, the runtime signals the deployment infrastructure that the update is complete and idles until the next patch is received.

3.1 Supported Applications

Retcon supports a broad set of event-driven embedded C applications. In particular, we target applications that consist of a set of event handlers that run to completion and yield back to the kernel upon completion – updates can modify existing handlers, add their own, and even remove existing ones, and do the same with any state. Retcon supports applications that statically allocate their global application state (e.g. a timer, flag, or other data). These properties are common in many embedded programs: the vast majority are written in C, avoid dynamic memory allocation for fear of unpredictable memory shortages and device instability, and rely on external and software interrupts to manage control flow instead of long-running processes. Together, they allow Retcon to perform static analysis at compile-time rather than expensive on-device analysis at runtime to identify valid live update points.

Testing Retcon Updates. It is important to note that Retcon's static analysis cannot reason about program *semantics*. For example, the halting problem prevents us from determining whether an application will actually reach a valid update point or remain stuck

```
(1)
(2)
(3)
(4)
(5)
(6)
(7)
(8)  void ventricle_beat() {
(9)
(10)
(11)
(12)   timer_start(&t1);
(13)   // process event
(14)
(15) }
```

Original

```
struct timer vrp_timer;
bool vrp_flag;

void vrp_timer_expire_callback() {
  vrp_flag = false;
}

void ventricle_beat() {
  if (!vrp_flag) {
    vrp_flag = true;
    timer_start(&vrp_timer);
    timer_start(&t1);
    // process event
  }
}
```

Updated

Figure 2: Pacemaker VRP component update. Adds a flag and additional timer (lines 1-2), and modifies the `ventricle_beat` event handler (lines 9-11) to ignore spurious ventricle sensor readings when they occur in sequence too quickly to indicate a real arrhythmia. At the end of the VRP period, `vrp_timer` expires and resets the flag (lines 4-6).

in an infinite loop [20]. Similarly, an application developer might modify the meaning of some program state without changing its name, which cannot be automatically detected. However, these issues are readily apparent as incorrect application behavior if an update is tested. Given that an update payload does not change between individual devices applying the same update, Retcon allows application developers to test the behavior of their update before widely deploying it to the bulk of targeted devices, and adjust the payload to ensure, for example, an update point is reached in practice.

Running Example: Updating a Pacemaker Controller. In the following sections, we describe Retcon's static analysis approach and runtime behavior. To practically demonstrate the process, we consider a small update to a pacemaker controller as an example. The update code is shown in Figure 2; in a few lines of code, we implement a Ventricular Refractory Period (VRP) [25],

which addresses noisy sensor readings that might erroneously report a heart beating much faster than desired, triggering undesired pacing stimulus. The update itself adds a flag indicating that additional beats should be temporarily ignored, a timer to toggle the flag after a valid amount of time has passed, and code to manage this interplay. We analyse this update throughout the rest of the paper and evaluate in Section 6.

Updating in the midst of changing event handlers highlights the importance of verifying asynchronous quiescence as an update condition as introduced in Section 1. If Retcon were to blindly update when a ventricular beat was detected, the new version of `ventricle_beat` might see timer expirations from timers set by the *original* application set. Luckily, in this case, the new handler function resets the timers, completely removing this danger. In Section 4, we show how Retcon performs this analysis automatically, and for many more handlers and asynchronous events.

4 Static Analysis

Retcon's static analysis serves two main purposes. First, it must identify specific application states that correspond with safe update points (states that are asynchronously quiescent) and ensure that the update will correctly move execution to a valid state in the new version. Second, Retcon generates state transfer instructions for the updated application to ensure that precious information is not lost during the update. As we expect the application build process to execute on a host with substantially greater processing capability, memory space, and execution time available than the target platform, Retcon performs the entirety of the necessary analysis statically, at compile time, simplifying the runtime component.

Section 4.1 describes how we model application state, including hardware configuration and active operations. Section 4.2 details how we extract an application's control-flow graph (CFG), where every node represents a different application state. Section 4.3 shows how Retcon generates state transfer instructions for use by its update runtime. Finally, we compare the original and updated CFGs to yield simple predicates identifying the valid update points.

4.1 Modeling Application State

In Retcon, we focus on modeling application state that persists across tasks or event handler invocations. At every potential update point, an application's state consists of three things:

- A name-value map of all static application memory state.
- The current state of the device's hardware configuration.
- A set of active asynchronous operations (e.g. timers or I/O).

In addition, each program has a list of event handler entry points (e.g. `ventricle_beat` in our running example), which are executed by the program in response to external interrupts or as callbacks for asynchronous operations completed by the kernel.

Retcon tracks the application's memory and hardware state to inform the state transfer process, because they influence the specific transfer instructions used in an update. Values in application memory with matching labels, such as the state of timer `t1` in the VRP update, or any shared hardware configuration, like an interrupt configuration, must also be seen by the updated application. In the case of differing hardware requirements, such as an additional sensor, Retcon must track the current hardware status in order to correctly reconfigure the device.

Tracking *asynchronous* quiescence. To ensure a safe update, we must also track the presence of active asynchronous operations, such as un-expired application timers. They are particularly important because applying a live update while they are outstanding could yield incorrect behavior after the update. A simple strategy – waiting for system states where the list of active asynchronous operations is empty – is an appealing approach because it's unlikely that an application would prefer to be updated while, for example, performing a radio transmission or waiting for a DMA write to complete. However, the practical structure of event-driven application creates a central problem. Commonly, asynchronous operations are stopped, reset, or started in an event handler triggered by an event from another asynchronous operation. This behavior is widespread in a complete pacemaker controller (Section 6), which manages up to five concurrently-active timers to detect different arrhythmias.

In this scenario, there is no point at which we can check for a completely quiescent system. Checking just *before* an event handler is executed would fail because active operations may be removed or reset only when the handler executes, such as timer `t1` when `ventricle_beat` is triggered. The system is not quiescent *during* the event handler's execution even if those operations are removed, and *after* the event handler completes, new or restarted operations will thwart an update even though carefully inspecting the application's semantics would reveal a safe update point.

While there might be a couple of states for a program where the set of active operations is truly empty, we leverage static program analysis to generate a broader set of update points that includes handlers where, upon receiving an event, *its execution guarantees that every pending asynchronous operation will be stopped or restarted.* Intuitively, this marks a "reset" point in the application where Retcon knows that any nominally active operations will not survive and the system is actually asynchronously quiescent. Considering our example, we can determine that a `ventricle_beat` event will likely put the system into a valid state for the VRP update because both implementations fully reset timer `t1`, ensuring that a prior timer started by the original version will not survive the update.

Given the relatively limited scope of embedded event-driven programs (compared to general cloud server applications), we can gain enough information to identify these safe update points without requiring exacting manual code annotations from the application developer. However, identifying the right handler invocations at which to trigger an update requires understanding how each individual handler modifies the program's set of active asynchronous operations. We describe extracting this behavior as a CFG below.

4.2 Generating Application CFGs

Retcon represents application logic as a set of directed, acyclic control-flow graphs, with one graph rooted at every application event handler. Graph nodes indicates operations – read or write to application memory, or a relevant system call – that modifies the tracked application state described in Section 4.1. CFG edges indicates program flow to the next operation node to be executed, along with any matching constraints on application state. A specific execution trace of an event handler is represented by a path through

```
{ "event": "call_timer",
  "call": "timer_start(&t1)",
  "caller": "ventricle_beat",
  "previousID": "79995",
  "currentID": "74519",
  ...
  "source_range": "<./src/main.c:174:5, col:59>",
  "constraints": [
       {"symbol":"reg_$0<enum state_t s>","range": "{ [1, 1] }"},
       {"symbol":"reg_$1<_Bool b1>","range": "{ [1, 1] }"},
  ],
  ... }
```

Figure 3: `clang`-derived symbolic execution node, capturing an important timer API system call to start one of the pacemaker timers. Retcon uses the node to identify the top-level event handler, link the call to previous memory stores and loads or system calls, and extracts range-based constraints on application state.

its CFG until reaching a node where no outgoing edge constraints are satisfied by the current state.

We extract these CFGs by first performing a pass on the application source code using the `clang` StaticAnalyzer code analysis tool [1], which generates a tree of branching symbolic execution traces through the application, starting at each top-level entry point. We implemented a custom "checker" that evaluates every path and yields a sequence of relevant operations, including reads, writes, and calls to kernel APIs that modify platform hardware or affect asynchronous operations. In addition, symbolic path constraints are generated by the analyzer on branching logic, from which we extract range-based constraints on application memory.

As an example, the snippet in Figure 3 shows the raw program state associated with the VRP update `t1` timer reset, and the associated constraints that must be satisfied to reach it.

Using the CFG, given a triggered event handler and the application's memory, hardware, and active asynchronous operation state, we can identify the sequence of operations to be executed under those constraints, and how they will modify asynchronous requests. Retcon determines whether they are reset or completely removed, which indicates that the device will achieve asynchronous quiescence during which an update can be performed.

While this type of path-based analysis scales exponentially with the complexity of the application, presenting an issue for live updates on more general applications [17], for small embedded applications with limited branching during execution, this effect is relatively minimal. In addition, the analysis for an update only needs to be completed once at compile time on a powerful processor, before the result is deployed to individual devices.

4.3 Transfer Instruction Generation

To ensure that the application keeps working as designed after a live update, Retcon must determine how to migrate application state to the updated version. It generates a sequence of transfer instructions necessary to synchronize the application memory and

```
(1)  def custom_state_transfer(symbol, predicate, constraints):
(2)          if symbol == 'vrp_flag':
(3)            return [0, 0]
(4)          else:
(5)            return None
```

Figure 4: Simple developer-specified state transfer for new state, written in Python and executed offline when preparing the update payload. In our example, the new `vrp_flag` cannot be automatically transferred, so it is initialized to 0 regardless of the specific predicate or constraints on pre-existing state. Retcon warns developers when they are missing a needed transfer implementation for new global state.

device hardware for the new version, which are executed by the embedded runtime described below in Section 5.

Application memory. Names present in the application memory of both the original and updated applications, such as timer `t1`, correspond to state that must be transferred during the update process. These are encoded as (source address, destination address, size) triples for the runtime to copy. As a result, the device runtime does not need to interpret any of the static analysis outputs, simply executing the transfer blindly, as all analysis has already taken place at compile time. Memory maintained by the original application but not the update is entirely ignored.

New state. Retcon handles new application state with no existing counterpart in the original version using a small amount of external input from the application developer. For example, Retcon does not know what the value of the newly-added `vrp_flag` in the VRP update (Figure 2) should be, because it cannot understand the updated version's semantics. Rather than implementing an imperfect technique to yield the correct initialization values, Retcon asks the developer at compile-time to specify custom logic that yields the correct initialization value for a specific name as a function of the predicate for that specific update point. Figure 4 shows how we could specify a simple default value of 0 for the new `vrp_flag` state in Python. The runtime will automatically initialize (but not start) the added `vrp_timer` without developer intervention by interacting with the operating system.

Hardware initialization. Retcon also generates transfer instructions to perform the update's expected hardware initialization. We assume for simplicity that this initialization is performed when the application starts up. Since the update CFG already contains all the relevant system calls known to initialize hardware resources (like timers and GPIO pins), Retcon determines the correct hardware configuration for the update by walking through the CFG for the application entry point. Given the current hardware state, a sequence of (system call pointer, arg1, arg2, ...) tuples is generated based on knowledge of the specific platform hardware subsystem APIs. For example, the analysis process for the VRP update automatically generates instructions to initialize the new versions of the application timers and external inputs (Figure 5). If the original and updated applications share the same set of hardware, non-configuration hardware registers are not modified by default, and are automatically accessed by the updated application version. If

```
{"initialization": [
  ["gpio_config", "gpio_dev", 10],
  ...
  ["gpio_pin_interrupt_configure", "gpio_dev", 10, RISING_EDGE],
  ["gpio_init_callback", "ventricle_beat"],
  ...
  ["timer_init", "&t1", "t1_expire_callback"],
  ["timer_init", "&vrp_timer", "vrp_timer_expire_callback"],
  ...
]}
```

Figure 5: Generated hardware initialization instructions for the VRP update. The ventricle beats are measured through GPIO pin 10 and a rising-edge interrupt triggers `ventricle_beat` execution. The relevant timers are initialized so that future expirations will route to the update.

```
{
  'event_name': 'ventricle_beat',
  'constraints': [{'range': [[1, 1]], 'symbol': 's1'}],
  'inactive_ops': ['t1'],
}
```

Figure 6: A predicate to determine when the pacemaker is in a valid state for the VRP update (edited for clarity). To trigger an update with this predicate, the system must be about to invoke the `ventricle_beat` handler, state `s1` must be equal to 1, and timer `t1` must not be running.

desired, Retcon allows application developers to implement a custom transfer routine that modifies the updated hardware state as a function of the current predicate, which is executed by the runtime during the update.

4.4 Generating Update Predicates

Given the CFGs of the original and updated application versions and a method for generating state transfer instructions, Retcon translates safe update states it has detected into simple "should the device update here?" predicates that succeed if the application has reached a valid update point. Each predicate is evaluated just before execution is routed to an application event handler, so they first check that the expected handler is going to be executed. Since different application states may result in an event handler executing differently, each handler execution path through the CFG generates a different predicate. The accumulated constraints on the path edges are collected and included as part of the resulting predicates to verify the desired memory and hardware state.

The predicate must also check for asynchronous quiescence: that the currently active asynchronous operations will stopped or restarted as a result of executing the event handler. Each operation, identified in an API-dependent manner (e.g. an active timer address), is placed into a list of allowable pending operations.

The enumeration of CFG paths and predicates is repeated for the same event handlers in the updated application, given the appropriate transferred state. If an execution path does not show up in the update (e.g. if the control flow was changed), that potential update point and associated predicate is abandoned as non-viable.

The constraints on the update's execution path are combined with the constraints from the original application path to ensure we have identified a shared update point in *both* application versions. Likewise, the system's currently active asynchronous operations should be a subset of those reset by both the original and updated handler. Intuitively, this tends to identify update points at recurring locations in the program that are stable between versions, such as the expiration of long-running timers, and excludes most one-off operations like radio transmissions, which could vary widely between versions. For example, the `ventricle_beat` handler fits this category, and a matching sample predicate is shown in Figure 6.

If Retcon does not find any valid asynchronous quiescence states that satisfy these conditions, it outputs an error to application developers at compile time. The developers then have the opportunity to modify program code to yield additional quiescent periods, before resubmitting the modified update to Retcon's analysis. However, in practice, we do not expect such cases to arise often, as we target embedded systems that have natural periodic processing cycles like sensing interrupts or power management routines that remain relatively stable between updates, and valid quiescent update points are likely to be discovered when these periodic events occur.

Finally, Retcon ensures that the updated event handler does not stop any operation not stopped by the original handler. This is a practical requirement, in order to support cases where the embedded operating system underlying Retcon executes additional application logic conditioned on whether the operation was active (e.g. attempting to stop a timer that has already been stopped). If attempting to stop such an operation results in the same behavior regardless of whether it was active or not, this step can be ignored.

5 Runtime

In comparison to the compile-time analysis pipeline, the update runtime is by design much more lightweight. Executing as a subsystem inside the embedded RTOS on-device, the runtime acquires and loads the updated application binary as the lowest priority scheduled task. This gives absolute latitude to application functionality to continue executing during this process; the update makes progress when the system overall is idle. Retcon takes advantage of the fact that, even while facing tight deadlines, embedded devices commonly exhibit periods of no activity (e.g. while waiting for asynchronous operations). While the device could instead be stopped for the entire duration of the update load, performing as much of the update process as possible in the background ensures application availability, at the cost of a slightly longer end-to-end update duration after deployment of the updated binary begins.

Safety in the face of errors during the update process is maintained by completely isolating the different application binaries once they are on the board. No modifications are made to the active application: during the update process, all necessary state is copied to the update's memory segment. If the runtime encounters an error while handling the update, it can stop and the old application version will continue running.

5.1 Update Payload

The update payload sent to the runtime combines the generated predicate and state transfer instructions extracted by the static analysis toolchain in Section 4, and contains:

- The unmodified, updated application binary.
- Additional metadata useful to the update runtime: application load address, location of application segments, addresses of update-related flags, etc.
- A list of *update predicates* identifying valid update points.
- A list of automatically-generated state transfer instructions.
- A list of hardware initialization instructions.

5.2 Predicate Verification

Once the runtime loads the updated application binary, it begins inspecting the system state to determine when a safe update state has been reached. Each generated predicate is tied to a specific application event handler (see Figure 6), so the runtime hooks some system events in the kernel – interrupts, timer expirations, etc. – that would return execution to the original application. When an event occurs, each predicate is verified in turn. If the system meets the predicate's application memory, hardware, and asynchronous operation constraints, the remaining predicates are ignored and the update process moves to the state transfer phase. If no predicates indicate the current state is safe for an update, the update is postponed and the original application handler executes.

5.3 State Transfer

When a predicate is satisfied, the matching set of state transfer instructions are fetched from the update payload (Section 5.1) and executed. Application memory is copied or initialized in the updated binary, with the same process occurring for the platform hardware. At this point, the application is suspended until the update completes, as there are limited hardware resources that cannot be shared by the application versions. Transfer instructions will not change after being generated during the compilation process, so pre-deployment testing is possible to mitigate the risk of deployment errors. Following the update, the pending event is routed to the updated handler and execution continues.

5.4 Error Handling

Retcon isolates update actions from the currently-running application until the update is complete. If an error occurs, control can be immediately handed back to the original application by routing the current event to the existing event handler and halting the update runtime. Update code and data are written into separate Flash and RAM memory regions without modifying the existing application binary. If a reboot were to occur during an update, the boot process would simply restart the unmodified original application. Before an update is applied, the binary text segment is written into non-volatile memory to provide persistence after the update completes, if a reboot were to occur.

6 Evaluation

We evaluate Retcon's ability to perform embedded live updates while maintaining application safety. Specifically, we demonstrate its ability to automatically identify safe update points and ensure continuity between applications by efficiently performing state transfer. In this section, we apply Retcon's toolchain to four applications: the dual-chamber pacemaker introduced in Section 3.1, an Artificial Pancreas System (APS) control algorithm, a simple PLC runtime, and a sense-and-send application integrating a third-party MQTT library [9]. Table 2 summarizes each application's memory and complexity characteristics.

Each application is evaluated on an nRF9160 SiP with a 64 MHz ARM Cortex-M33 microprocessor, 1 MB of Flash, and 256 kB of RAM [48]. The update runtime is implemented as a 1,152 SLOC subsystem for the Zephyr RTOS [2], on which each application executes. The compiled RTOS, including the Retcon subsystem, requires 86.15 kB of Flash and 32.08 kB RAM at runtime, compared to 81.06 kB and 3.98 kB, respectively, for the OS without live updates enabled. As a result, Retcon requires 5.09 kB for code and 28.1 kB of memory to operate. The largest use of this memory (85%) is in statically allocating a 24 kB buffer for receiving update payloads. This allocation can be adjusted to reduce memory overhead by limiting maximum payload size, which could be achieved by limiting the number of predicates contained in each payload.

6.1 Applications

Pacemaker controller. Embedded control logic is a natural target for a live-update mechanism that can maintain continuity over an update, and pacemaker controllers in particular may require urgent updates [14]. We implement the formal dual-chamber pacemaker control algorithm model verified by Jiang et al. [25], based on functional descriptions of pacemaker operation from Boston Scientific. Our implementation executes as a collection of asynchronous state machines controlled by four independent timers. Another microcontroller, acting as a synthetic heart, provides two external "sense" inputs to mark each part of the beat and receives two "pace" actuation signals if the controller detects an arrhythmia. While the memory footprint of the controller is minimal, the constantly-interacting state machines mean that detecting an appropriate update point is much more difficult compared to an application with a simple duty cycle. As previously discussed, we update the pacemaker with a VRP component to ignore spurious signal inputs.

APS. We implement an artificial pancreas system (APS) based on the OpenAPS reference design [34], representing a simple embedded sensing application with a significant amount of cached historical data that is critical to providing accurate care. An APS is a personal, DIY solution that closes the cycle between a blood glucose measurement device and an insulin pump. As additional blood measurements are taken, the controller updates a running internal estimate of a person's blood glucose levels, and, periodically, temporarily increases the patient's insulin delivery rate to maintain safe conditions. We update the APS controller with a modification described in the OpenAPS reference [34] to safely adjust to unexpected blood glucose deviations.

PLC Runtime. We implement a PLC runtime with a 3-state ladder-logic state machine. As the PLC application repeats frequently and periodically, updates can occur from one scan cycle to the next. We emulate a PLC runtime on-device to execute ladder logic programs, a widespread visual PLC programming method,

Table 2: The size and complexity of each evaluated application. At static analysis, we measure the size of the resulting CFG and analysis duration. The resulting update payload is transmitted to the Retcon runtime on-device. † Due to evaluation platform memory constraints, in updates with a high predicate count, we empirically limit the number of predicates sent to the runtime to 50. This maintains system safety but may result in a longer end-to-end updates as some valid predicates are not evaluated.

App	SLOC	RAM Used (bytes)	Event Handlers	CFG Size (nodes)	Analysis Time (s)	Update Payload (kB)	Predicates	**Update Time** (μs)
Pacemaker	209	240	7	169	54	3.03	12	353.97 ± 3.96
APS	268	2644	2	5798	742	4.50	26	396.98 ± 3.98
PLC	319	88	4	9982	1666	12.45†	136†	244.36 ± 1.53
MQTT Client	1334	4332	12	14324	2271	8.95	41	638.28 ± 1.03

with LDmicro [57], an open source compiler that automatically generates C code from arbitrary ladder logic programs. While its memory footprint is the lowest of the evaluated applications, the bulky machine-generated code yields a large control flow graph of almost ten thousand nodes, slowing analysis and generating a large number of predicates (Table 2. Our update includes generated ladder logic adding two additional application-level states to the execution schedule, controlling a separate output.

Networked sensor. Finally, we evaluate a sense-and-send MQTT client built using the MQTT-C library [9] to perform lightweight communication with a remote server. For simplicity, we proxy published MQTT packets from the device over a serial link before they are relayed to a remote MQTT server, and the embedded client periodically queries for the resulting responses. Importantly, the MQTT client demonstrates Retcon's ability to support embedded software libraries that were not developed to explicitly support a live update capability; we use Retcon to update the system on-the-fly to publish on different topics. Table 2 shows that while there are disadvantages to updating a complex application like the MQTT client – it generates the largest control flow graph and requires almost forty minutes of offline analysis – the update overhead at runtime remains negligible.

6.2 Pacemaker Update Process

The pacemaker trace in Figure 7 explores Retcon's update process in detail. The goal of the updated controller is to implement a VRP function against spurious signals sensed from the heart. In our test setup, the simulated "heart" has a functional ventricle but requires the atrium to be continually paced by the controller. Periodically, the original controller is given two ventricle inputs in quick succession, which results in incorrectly delaying the next atrial pace by 50 ms. To fix the issue, we generate a small patch that adds a VRP state machine [25] to ignore these repeated signals and yields the expected 950 ms interval between each pace.

The update process starts at approximately $t = 21$s by transferring the appropriate update payload to the device, but remains in the background, processing only when the system is idle and in small chunks before serving application interrupts to avoid interfering with application functionality. Remaining in the background, the update payload is fully received at approximately $t = 28$ and Retcon's runtime begins checking system state against the payload's predicates whenever control is about to be handed to the application. While the next two incoming ventricle events immediately following do not satisfy any of the predicates, the following timer expiration triggers an atrial pace that satisfies a predicate. Retcon, detecting that the system is in a safe update state, begins the quick blocking update phase.

As discussed in Section 5, the entire platform now performs state transfer and initialization before routing control flow to the *updated* controller's atrial pace handler with minimal delay. The newly-added VRP component now begins to ignore duplicate sensor inputs, resulting in the expected pacing behavior. We evaluate the performance of this process in Figure 8 and Figure 10.

6.3 Update Runtime Performance

Baselines. Our baseline comparisons for timely updates are the industry-standard flash-and-reboot approach to upgrading system firmware, and an application-swapping approach that uses Retcon to deliver a smaller application-only binary, but still requires a system reset. For the flash-and-reboot method, the resulting update time consists primarily of the cost to erase the device's persistent flash memory and copy the entire operating system, packaged with the application, while the application swap only measures system reinitialization. This process, assuming a maximum application payload size, requires approximately 4 seconds on our platform to flash a new binary, while an application swap requires 1.93 ms. For comparison, each application update we apply using *Retcon successfully performs the update live in less than a millisecond.*

Update latency. Figure 8 shows the distribution of update latencies (the amount of time the application is blocked from executing in order to finalize the update) for each application. For the PLC, which has very little state to transfer and no new kernel state to initialize (e.g. a new timer), the interruption is minimal at an average of 244 μs. However, updates that require additional initialization (the pacemaker controller) or must transfer large buffers (the APS and MQTT client) yield a two- or three-times greater latency.

However, the interrupt latency caused by a triggered update is not necessarily the worst-case scenario for an application using Retcon. While the analysis model in Section 4 allows us to identify safe program points for an update, it cannot guarantee that we will ever reach those points during a normal run of the program, potentially never completing an update. In Figure 9 we evaluate the performance impact of an application that evaluates each of its predicates without detecting a safe update point.

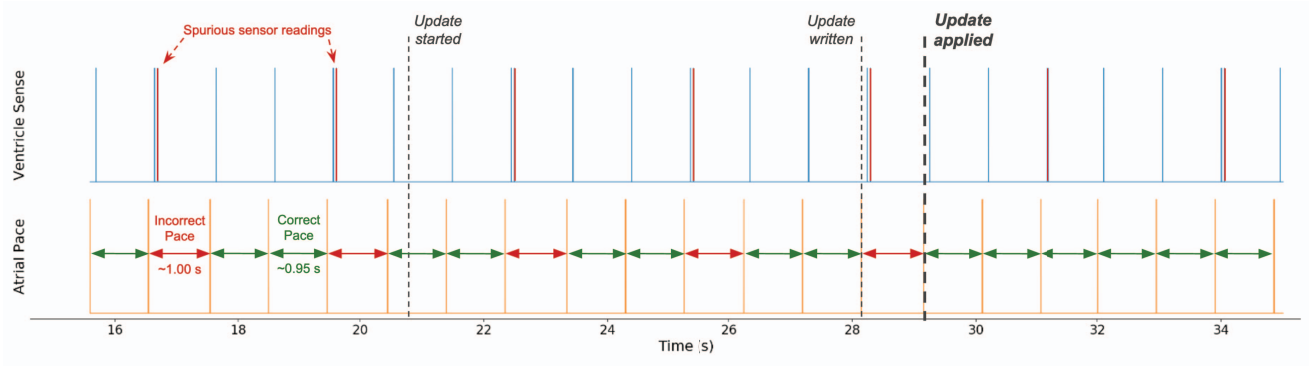

Figure 7: Overview of the pacemaker update that implements a VRP, accounting for spurious duplicate signal readings from the heart's ventricle (Section 6.2). Prior to the update, the controller is periodically incorrectly delaying a required pace to the heart's atrium because additional impulses (highlighted in red) are detected from the ventricle. After the update, the controller begins to pace optimally without missing a beat.

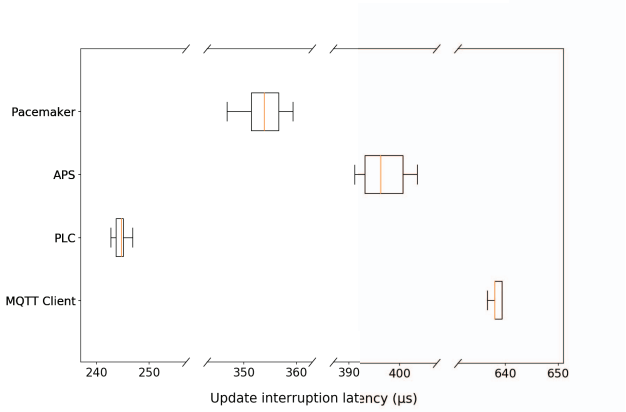

Figure 8: Interrupt latency when the Retcon runtime detects a safe update state and performs an update before returning execution to the new program version. For each application, the variance of update latency is minimally variable and in even the slowest case requires only approximately 650 μs.

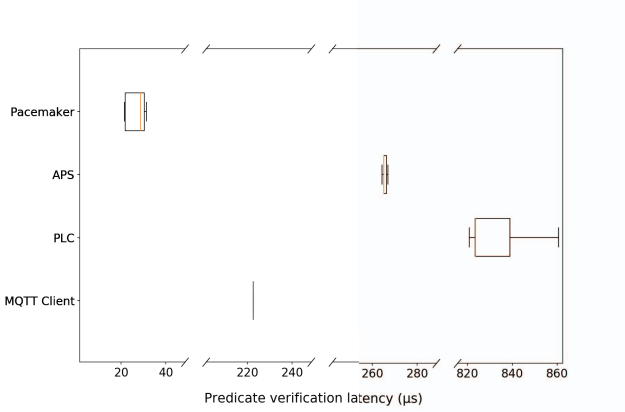

Figure 9: Interrupt latency when the Retcon runtime does not detect a suitable safe system state to trigger an update and as a result, evaluates every predicate provided by the update. In general, cost is dominated by the number of predicates that must be checked.

Applications with few predicates incur small evaluation overheads, while the opposite is true for those with many predicates. The PLC, even with a limited set of predicates (see Table 2), requires almost 900 μs to evaluate each of its predicates in the worst case, far larger than its one-time update interruption latency of about 250 μs. The pacemaker can check each of its 12 predicates in less than 40 μs, but exhibits higher update latency when a safe state is found due to increased state initialization.

Application-specific factors. The breakdown of contributing factors to update latency, shown in Figure 10, clearly splits the evaluated applications into two groups. The time required to update the APS and MQTT client implementations is dominated by the cost of state transfer, as both must copy over 2 kB in data buffers of historical measurements (in the case of the APS) or sending and receiving message queues (in the case of the MQTT client). On the other hand, the PLC and Pacemaker implementations require much less state transfer, so latency is primarily dominated by new state initialization. In the end, the total delay in performing an update is determined by application-specific factors, while the overhead of the runtime, as well as the cost of evaluating predicates until a valid update point is detected, remains small in every case.

7 Discussion

We briefly analyze some broader implications, limitations, and potential future directions suggested by our experience designing and evaluating Retcon. While Retcon targets a large number of event-driven embedded applications, it does not consider some more complex use cases, such as multi-threaded or multi-core applications – expanding Retcon to coordinate distributed updates between multiple microprocessors on a device (such as a security or signal processing chip) is an interesting avenue for future work. Similarly, applications that currently use dynamic memory allocation cannot be updated by Retcon. However, since applications may have asynchronous quiescent update points that are not dependent

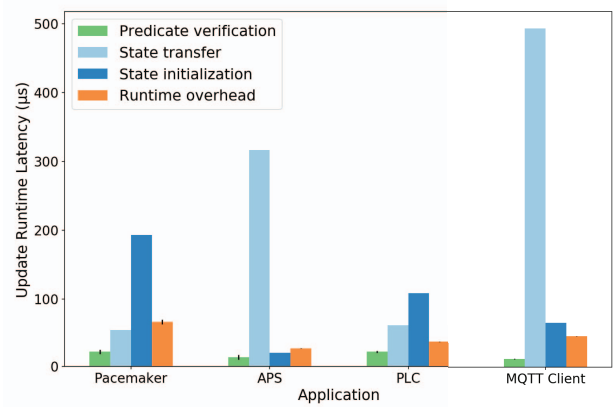

Figure 10: Breakdown of individual Retcon runtime component contributions to the total update latencies in Figure 8. We verify that the system state satisfies a given predicate, then transfer program state to the new application and initialize new state. Inducing the RTOS to pass control to the update incurs some overhead.

on heap state, we envision an extension to Retcon that tracks the actions of a simple memory allocator in our runtime subsystem to perform dynamic state transfer.

The scalability of Retcon's static analysis is dictated by the efficiency of the underlying symbolic execution. In Section 6, we successfully analyze a PLC runtime consisting largely of automatically-generated C code, but the process requires almost an hour of analysis. This is a byproduct of the branching execution tree-style analysis that the `clang` static analysis engine uses, which is primarily used to generate compiler warnings for developers. Retcon has to track any store/load global state modifications, as well as any relevant system calls, and update constraints on program branching, yielding an exponentially-expanding search space. The number of predicates could also scale similarly. Of the 12 predicates we generate for the pacemaker controller in Section 6, 6 are manually verified as impossible to observe during normal operation due to incompatibilities between state constraints and expected running timers. However, since the analysis process is performed only once offline, and the results are broadcast to target devices, a longer analysis time is acceptable in exchange for no overhead applied directly at the destination. Finally, unlike other C symbolic execution tools like Otter [45] or CREST [10], `clang` can operate directly on C source code that we cross-compile for an embedded ARM processor without modification.

Retcon uses state naming as a proxy for that state's semantics, but this is not always the case. As a result, updates may crash or behavior erroneously if our analysis transfers state incorrectly. As discussed in Section 3.1, Retcon's role is as part of a validation, verification, and deployment pipeline, not as a total replacement. Many errors, such as choosing predicates that rarely succeed in practice, if at all, are readily noticeable when tested. Through empirical observation of test updates, issues that might take down a fleet of sensors can be easily avoided, and full deployment can even be accelerated by prioritizing update points that occur frequently.

While fast, Retcon's updates still exhibit some variability in update timing, as evaluating predicates and performing state transfer can require varying amounts of computation depending on the current system state, and a delay is inherently added to interrupt routines. As a result, these updates may not be directly applicable to systems with hard real-time constraints. A hard real-time extension to Retcon could see application developers pick a small subset of well-known predicates that always execute and introduce (safe and feasible) constant-time delays to interrupt deliveries to mask update interruptions.

Finally, although we require some manual developer intervention for new state values or special hardware state transfer Retcon offers a significant improvement in reducing developer workload by automatically transferring common between application versions, relieving what would otherwise be an exhaustive manual effort to line up source and destination addresses across update versions. As a result of embedded application constraints, we can fully automate the process of determining when a safe update point occurs (at compile- and run-time) that is consistent between application versions. In simple applications like the APS (6), Retcon correctly finds the same natural update point that an embedded developer would manually select, while in applications like the pacemaker, where correct behavior is determined by multiple independent state machines operating all at once, Retcon can easily identify update points that would require deep application expertise and careful evaluation to confirm as a safe update point.

8 Conclusion

In this paper, we presented Retcon, which performs live updates on event-driven applications executing on constrained embedded devices. Retcon updates applications safely when they satisfy a novel *asynchronous quiescence* condition that we define for the event-driven setting, uses a static analysis toolchain to identify those points, and correctly transfers state between application versions. A simple embedded runtime ensures updates are safe and timely: our evaluation demonstrates that Retcon can update a variety of applications in a manner of microseconds.

9 Acknowledgements

We thank the anonymous reviewers and shepherd for their helpful feedback, Cristiano Giuffrida and Lucy Cherkasova whose insightful comments helped shape this work, and Will Huang and the rest of Lab11 for their helpful comments. This material is based upon work supported by the U.S. Department of Energy's Office of Energy Efficiency and Renewable Energy (EERE) under the award number DE-EE0008220, an Okawa Foundation Research Grant, and the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

[1] [n. d.]. Clang Static Analyzer. https://clang-analyzer.llvm.org/.
[2] [n. d.]. Zephyr RTOS. https://www.zephyrproject.org/.
[3] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. 2005. OPUS: Online Patches and Updates for Security.. In *USENIX Security Symposium*.
[4] Jeff Arnold and M Frans Kaashoek. 2009. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*.
[5] Rockwell Automation. 2019. ControlFLASH User Manual. . Accessed: 2020-05-21.

[6] Emmanuel Baccelli, Cenk Gündoğan, Oliver Hahm, Peter Kietzmann, Martine S Lenders, Hauke Petersen, Kaspar Schleiser, Thomas C Schmidt, and Matthias Wählisch. 2018. RIOT: An open source operating system for low-end embedded devices in the IoT. *IEEE Internet of Things Journal* 5, 6 (2018), 4428–4440.

[7] R. Barry. 2009. *FreeRTOS reference manual: API functions and configuration options.*

[8] Francesco Basile, Pasquale Chiacchio, and Diego Gerbasio. 2012. On the implementation of industrial automation systems based on PLC. *IEEE Transactions on Automation Science and Engineering* 10, 4 (2012), 990–1003.

[9] Liam Bindle and Demilade Adeoye. 2021. LiamBindle/MQTT-C: A portable MQTT C client for embedded systems and PCs alike. https://github.com/LiamBindle/MQTT-C.

[10] Jacob Burnim and Koushik Sen. 2008. Heuristics for scalable dynamic test generation. In *2008 23rd IEEE/ACM Intl. Conf. on Automated Software Engineering*.

[11] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. 2007. Polus: A powerful live updating system. In *29th Intl. Conf. on Software Eng.*

[12] Yang Chen, Omprakash Gnawali, Maria Kazandjieva, Philip Levis, and John Regehr. 2009. Surviving sensor network software faults. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*.

[13] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *29th annual IEEE international conference on local computer networks*. IEEE.

[14] FDA. 2017. Firmware Update to Address Cybersecurity Vulnerabilities Identified in Abbott's (formerly St. Jude Medical's) Implantable Cardiac Pacemakers: FDA Safety Communications. . Accessed: 2020-05-25.

[15] Sebastian Fischmeister and Klemens Winkler. 2005. Non-blocking deterministic replacement of functionality, timing, and data-flow for hard real-time systems at runtime. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*.

[16] Recorded Future. 2022. Continued Targeting of Indian Power Grid Assets by Chinese State-Sponsored Activity Group. https://go.recordedfuture.com/hubfs/reports/ta-2022-0406.pdf.

[17] Cristiano Giuffrida, Anton Kuijsten, and Andrew S Tanenbaum. 2013. Safe and automatic live update for operating systems. *ACM Sigplan Notices* 48, 4 (2013), 279–292.

[18] Giovani Gracioli and Antônio Augusto Fröhlich. 2010. ELUS: A dynamic software reconfiguration infrastructure for embedded systems. In *2010 17th International Conference on Telecommunications*.

[19] Tianxiao Gu, Chun Cao, Chang Xu, Xiaoxing Ma, Linghao Zhang, and Jian Lu. 2012. Javelus: A low disruptive approach to dynamic software updates. In *2012 19th Asia-Pacific Software Engineering Conference*.

[20] Deepak Gupta, Pankaj Jalote, and Gautam Barua. 1996. A formal framework for on-line software version change. *IEEE Trans. on Software engineering* 22, 2 (1996), 120–131.

[21] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. 2005. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*.

[22] Christopher M Hayden, Edward K Smith, Michail Denchev, Michael Hicks, and Jeffrey S Foster. 2012. Kitsune: Efficient, general-purpose dynamic software updating for C. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*.

[23] Petr Hosek and Cristian Cadar. 2013. Safe software updates via multi-version execution. In *2013 35th International Conference on Software Engineering (ICSE)*.

[24] Jonathan W Hui and David Culler. 2004. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*.

[25] Zhihao Jiang, Miroslav Pajic, Salar Moarref, Rajeev Alur, and Rahul Mangharam. 2012. Modeling and verification of a dual chamber implantable pacemaker. In *Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*.

[26] JUUL Labs OSS. [n. d.]. MCUboot. https://juullabs-oss.github.io/mcuboot/.

[27] Sungjoo Kang, Ingeol Chun, and Wontae Kim. 2014. Dynamic software updating for cyber-physical systems. In *The 18th IEEE International Symposium on Consumer Electronics (ISCE 2014)*.

[28] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. 2016. Instant {OS} Updates via Userspace Checkpoint-and-Restart. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*.

[29] Justin Z Lee, Mark J Henrich, Paul Bibby, Siva K Mulpuru, Paul A Friedman, Yong-Mei Cha, and Komandoor Srivathsan. 2019. Pacemaker firmware update and interrogation malfunction. *HeartRhythm case reports* 5, 4 (2019), 213.

[30] Philip Levis and David Culler. 2002. Maté: A tiny virtual machine for sensor networks. *ACM Sigplan Notices* 37, 10 (2002), 85–95.

[31] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. 2005. TinyOS: An operating system for sensor networks. In *Ambient intelligence*. 115–148.

[32] Philip Levis, Neil Patel, David Culler, and Scott Shenker. 2004. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. of the 1st USENIX/ACM Symp. on Networked Systems Design and Implementation*.

[33] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*.

[34] Dana Lewis. 2019. OpenAPS 0ref Reference Design. https://github.com/openaps/oref0.

[35] Lucy Ellen Lwakatare, Teemu Karvonen, Tanja Sauvola, Pasi Kuvaja, Helena Holmström Olsson, Jan Bosch, and Markku Oivo. 2016. Towards DevOps in the embedded systems domain: Why is it so hard?. In *2016 49th hawaii international conference on system sciences (hicss)*.

[36] Kristis Makris and Rida A Bazzi. 2009. Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction.. In *USENIX ATC*.

[37] Kristis Makris and Kyung Dong Ryu. 2007. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*.

[38] Jami Montgomery. 2004. A model for updating real-time applications. *Real-Time Systems* 27, 2 (2004), 169–189.

[39] Imanol Mugarza, Jorge Parra, and Eduardo Jacob. 2018. Analysis of existing dynamic software updating techniques for safe and secure industrial control systems. *Intl. journal of safety and security eng.* 8, 1 (2018), 121–131.

[40] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. 2006. Practical dynamic software updating for C. *ACM SIGPLAN Notices* 41, 6 (2006), 72–83.

[41] Christian Niesler, Sebastian Surminski, and Lucas Davi. 2021. HERA: Hotpatching of Embedded Real-time Applications. In *Proc. of 28th Network and Distributed System Security Symposium (NDSS 2021)*.

[42] Agnes C Noubissi, Julien Iguchi-Cartigny, and Jean-Louis Lanet. 2011. Hot updates for java based smart cards. In *IEEE 27th Intl. Conf. on Data Eng. Workshops*.

[43] Luís Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. 2019. Mvedsua: Higher availability dynamic software updates via multi-version execution. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[44] Niels Reijers and Chi-Sheng Shih. 2018. CapeVM: A Safe and Fast Virtual Machine for Resource-Constrained Internet-of-Things Devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*.

[45] Elnatan Reisner, Charles Song, Kin-Keung Ma, Jeffrey S Foster, and Adam Porter. 2010. Using symbolic evaluation to understand behavior in configurable software systems. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*.

[46] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. 2013. A survey of dynamic software updating. *Journal of Software: Evolution and Process* 25, 5 (2013), 535–568.

[47] Habib Seifzadeh, Ali Asghar Pourhaji Kazem, Mehdi Kargahi, and Ali Movaghar. 2009. A method for dynamic software updating in real-time systems. In *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*.

[48] Nordic Semiconductor. [n. d.]. nRF9160 cellular IoT System-in-Package. https://www.nordicsemi.com/-/media/Software-and-other-downloads/Product-Briefs/nRF9160-SiP-product-brief.pdf.

[49] Rodrigo Steiner, Giovani Gracioli, Rita de Cássia Cazu Soldi, and Antônio Augusto Fröhlich. 2012. An operating system runtime reprogramming infrastructure for WSN. In *2012 IEEE Symposium on Computers and Communications (ISCC)*.

[50] SUSE. [n. d.]. SUSE Linux Enterprise Live Patching. https://www.suse.com/products/live-patching/.

[51] Trusted Firmware. [n. d.]. ARM Trusted Firmware-M. https://www.trustedfirmware.org/.

[52] Kaleem Ullah, Munam Ali Shah, and Sijing Zhang. 2016. Effective ways to use Internet of Things in the field of medical and smart health care. In *2016 international conference on intelligent systems engineering (ICISE)*.

[53] Michael Wahler and Manuel Oriol. 2014. Disruption-free software updates in automation systems. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*.

[54] Michael Wahler, Manuel Oriol, and Aurelien Monot. 2015. Real-time multi-core components for cyber-physical systems. In *2015 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*.

[55] Michael Wahler, Stefan Richter, Sumit Kumar, and Manuel Oriol. 2011. Non-disruptive large-scale component updates for real-time controllers. In *2011 IEEE 27th International Conference on Data Engineering Workshops*.

[56] Michael Wahler, Stefan Richter, and Manuel Oriol. 2009. Dynamic software updates for real-time systems. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*.

[57] Jonathan Westhues. 2016. LDmciro: Ladder Logic for PIC and AVR. https://cq.cx/ladder.pl.

[58] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave Jing Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. 2020. {BLESA}: Spoofing Attacks against Reconnections in Bluetooth Low Energy. In *14th {USENIX} Workshop on Offensive Technologies*.

[59] Hansong Xu, Wei Yu, David Griffith, and Nada Golmie. 2018. A survey on industrial Internet of Things: A cyber-physical systems perspective. *IEEE Access* 6 (2018), 78238–78259.