Writing Memory-Safe Code for Embedded Systems

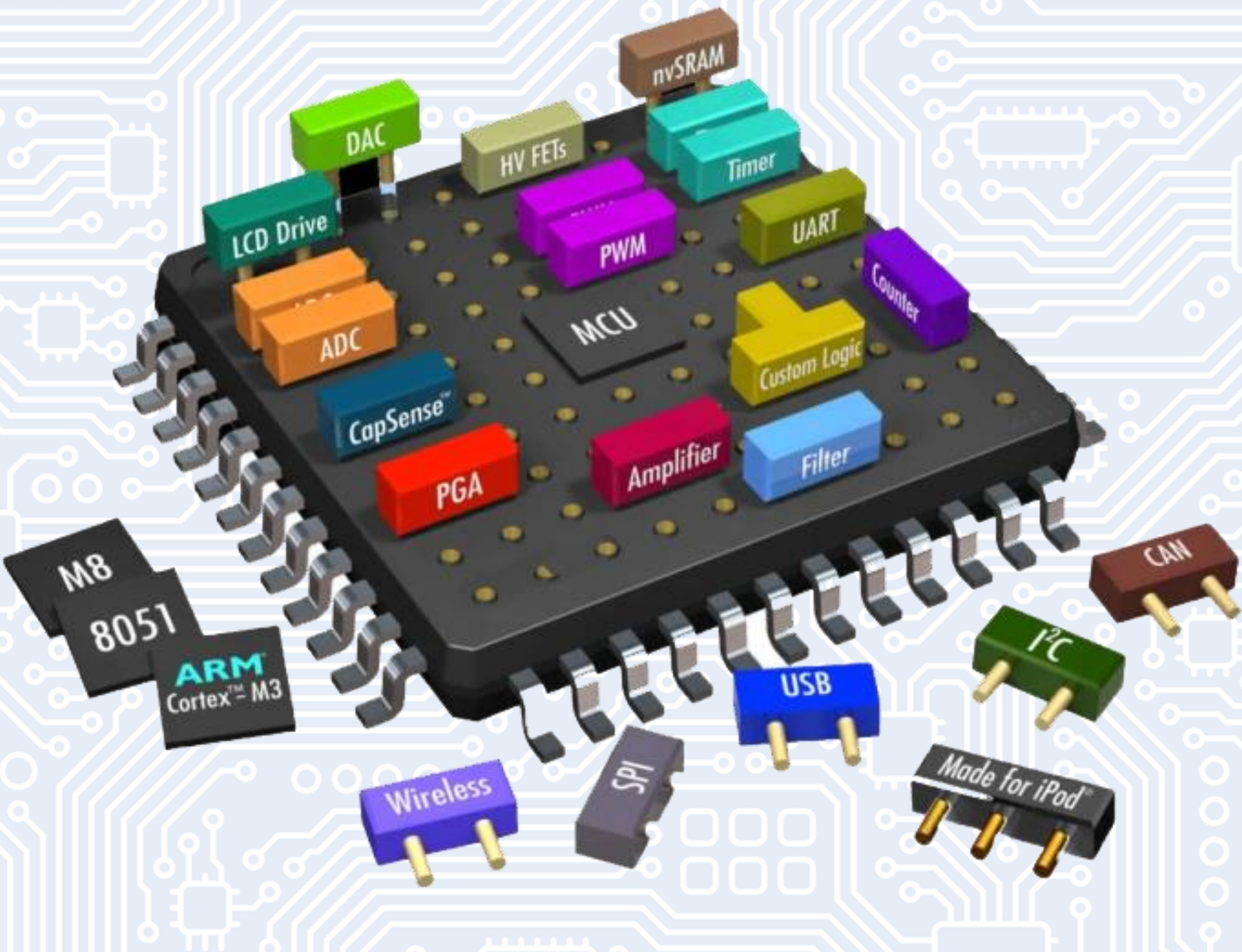


Table of Contents

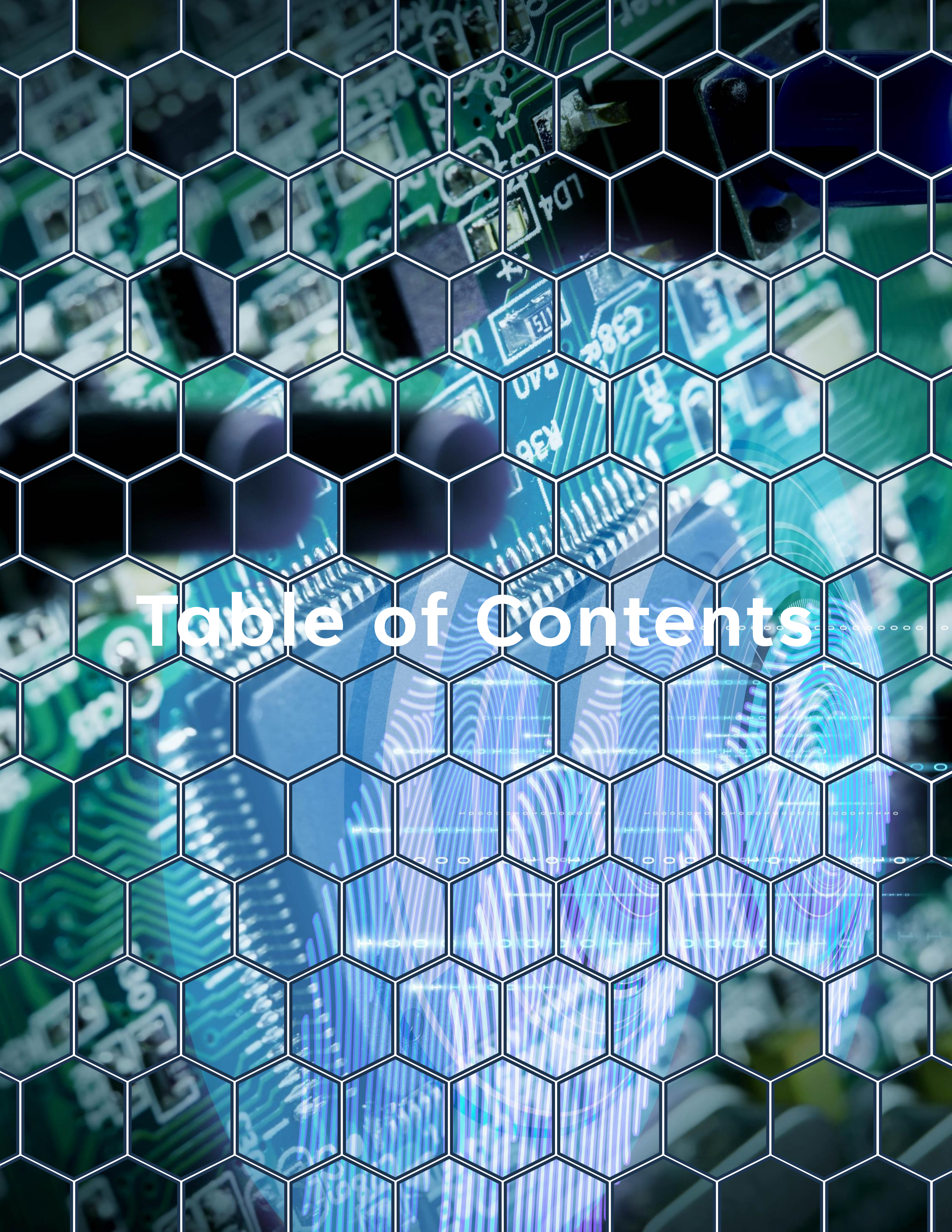


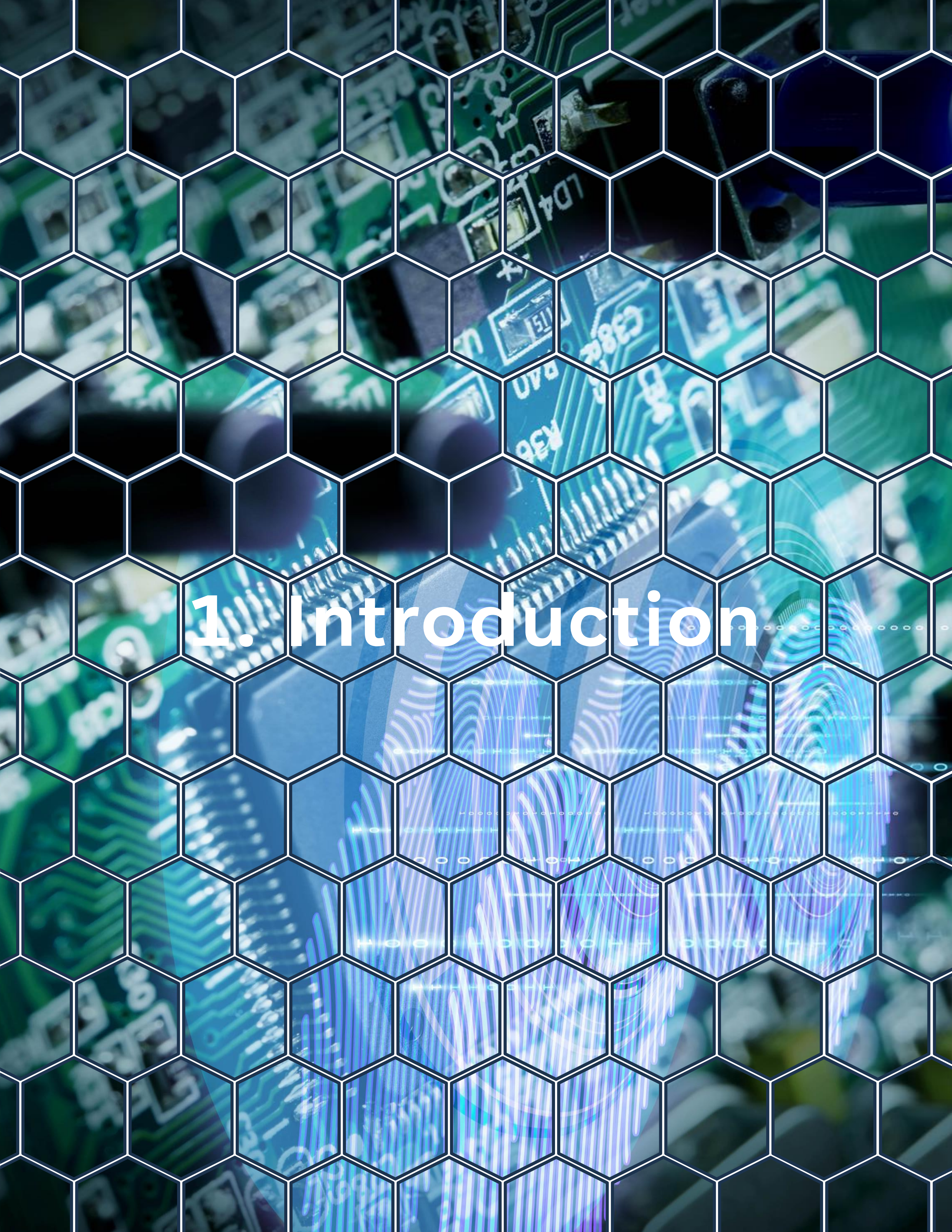
Table of Contents

1. Introduction
2. Why Memory Safety is Critical in Embedded Systems
3. Typical Memory Risks in Embedded Development
    1. Stack Overflows
    2. Heap Fragmentation and Unsafe Dynamic Memory
    3. Buffer Overflows and Bounds Violations
    4. Dangling Pointers and Use-After-Free
    5. Improper Use of volatile
    6. Uninitialized Variables and Undefined Behavior
    7. Memory Safety in Interrupt Service Routines (ISRs)

Table of Contents

4. Memory-Safe Coding Practices
    1. Avoid Dynamic Memory on Small MCUs
    2. Static Allocation and Buffer Size Guards
    3. Safe Pointer Handling
    4. Using volatile Correctly for Hardware Interaction
    5. Compiler Warnings and Static Analysis
5. Conclusion

1. Introduction



1. Introduction

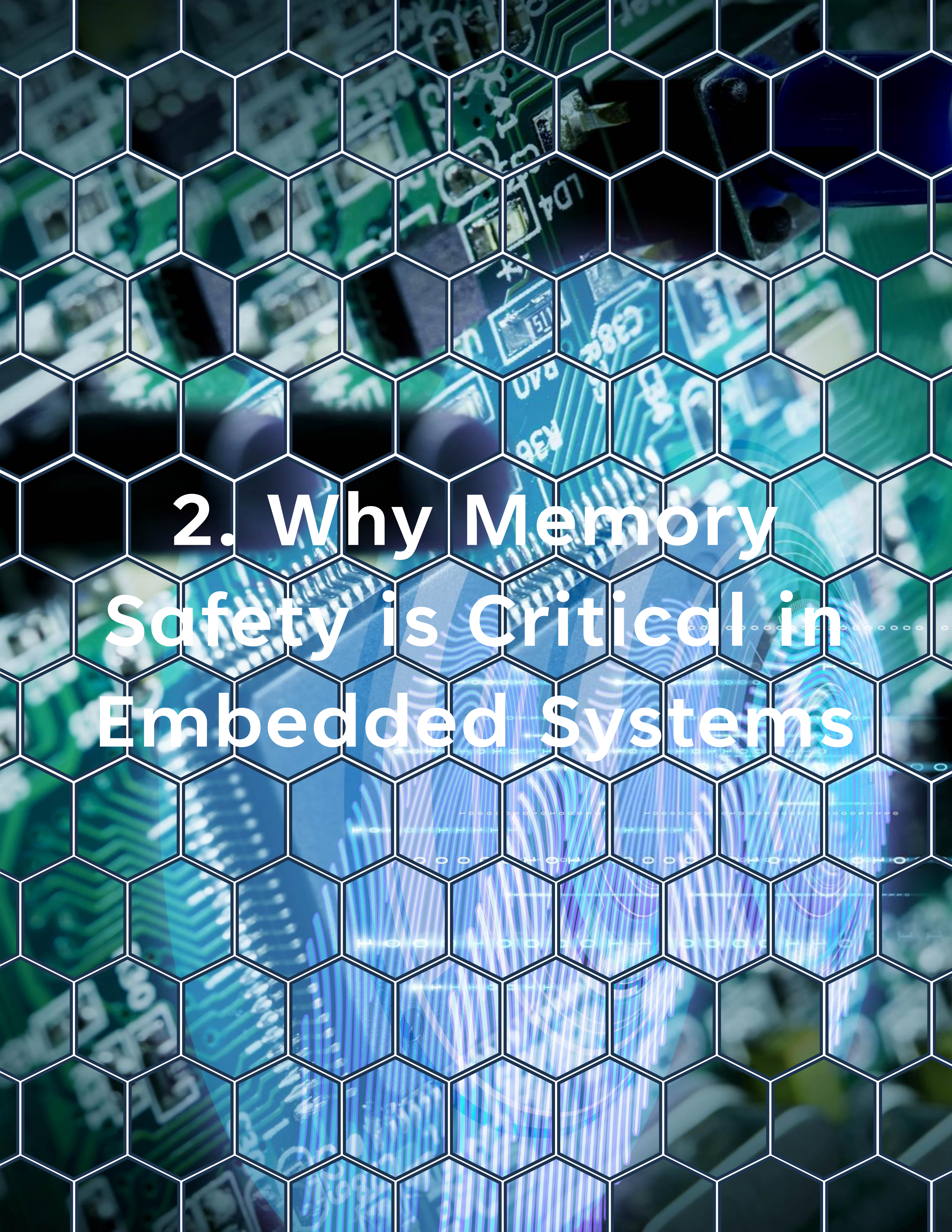
Memory safety is a critical concern in embedded systems development, where constrained resources and real-time requirements amplify the consequences of memory corruption. Unlike general-purpose computing, embedded systems often lack Memory Protection Units (MPUs) and operate with limited RAM and Flash, making memory errors catastrophic. A single buffer overflow or dangling pointer can lead to firmware crashes, undefined behavior, or even hardware damage—particularly in safety-critical applications like medical devices, automotive systems, and industrial controllers.

1. Introduction

The **ATtiny1616**, a popular low-power microcontroller from Microchip (formerly Atmel), exemplifies these challenges. With only 16KB Flash and 2KB RAM, efficient and safe memory management is non-negotiable.

This article explores practical techniques for writing memory-safe firmware in C, covering stack, heap, and hardware-related pitfalls while providing actionable solutions tailored to resource-constrained MCUs.

2. Why Memory Safety is Critical in Embedded Systems

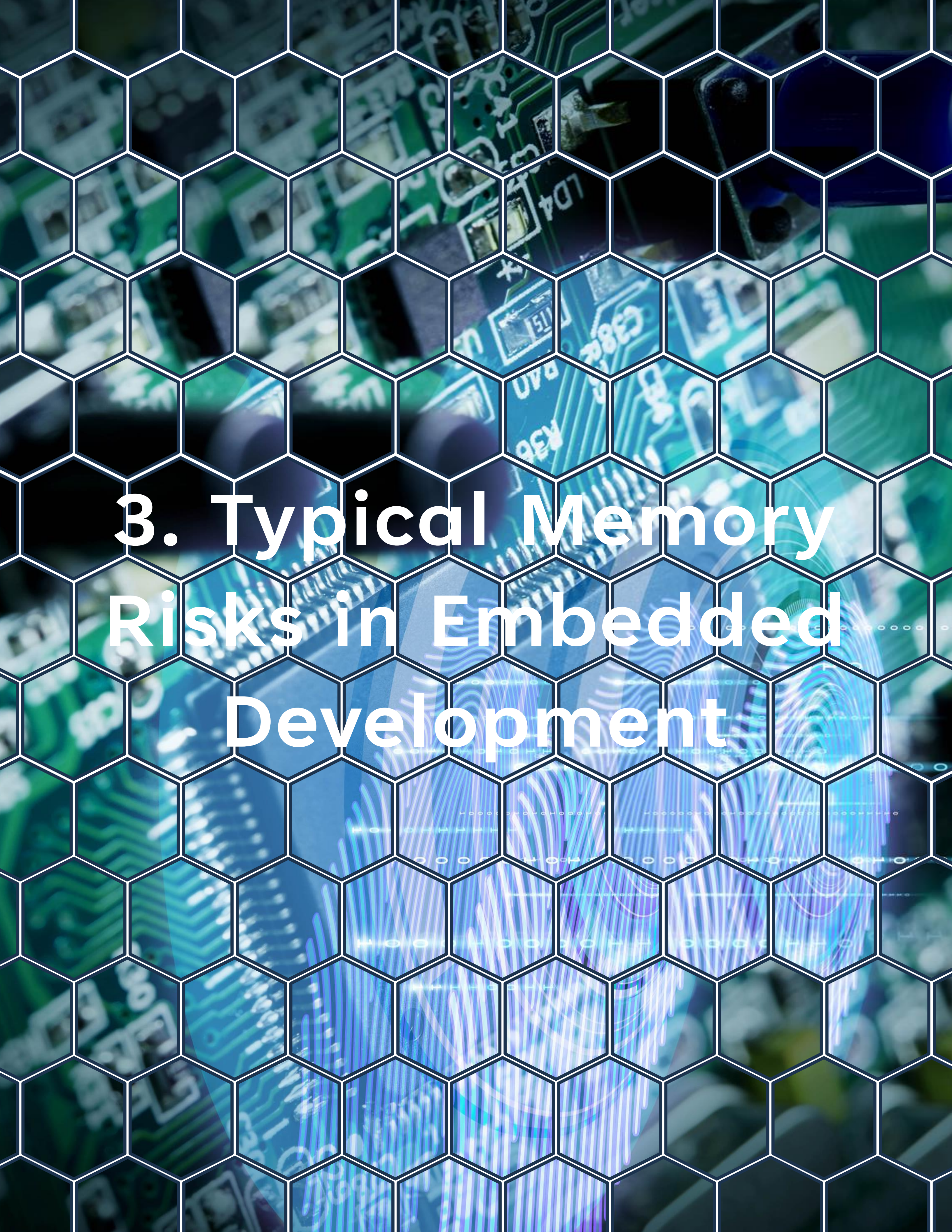


2. Why Memory Safety is Critical in Embedded Systems

Memory safety in embedded systems is not a luxury—it is a necessity. Systems running on MCUs like the ATtiny1616 often control hardware that interacts with the physical world: sensors, actuators, motors, or communication interfaces. A memory corruption bug here doesn't just crash software; it could damage hardware, cause safety hazards, or result in costly downtime.

2. Why Memory Safety is Critical in Embedded Systems

Embedded systems present a unique set of memory safety challenges that distinguish them from traditional computing environments. The absence of a Memory Protection Unit (MPU) in most microcontrollers means that errant memory accesses can corrupt critical system data structures, overwrite interrupt vectors, or modify hardware configuration registers without detection. ***This lack of hardware-level protection places the entire burden of memory safety on the software developer***.



3. Typical Memory Risks in Embedded Development

3. Typical Memory Risks in Embedded Development

3.1 Stack Overflows

The stack holds function call data, local variables, and return addresses. On small MCUs with limited SRAM (e.g., 2KB on some ATtiny1616 variants), deep call chains or large local buffers can cause stack overflows, corrupting adjacent memory.
Stack overflows occur when a function's stack frame exceeds available memory, corrupting adjacent data.

3. Typical Memory Risks in Embedded Development

⚠️ Unsafe Function: Risk of Stack Overflow

```c
#include <avr/io.h>

// Static buffer in global memory — avoids stack overflow risk
static uint8_t safe_buffer[512];

void safe_function(void) {
    // Operate on statically allocated buffer
    for (uint16_t i = 0; i < 512; i++) {
        safe_buffer[i] = i % 256;
    }
}
```

Why Unsafe?

On the ATtiny1616, using a 512-byte local array rapidly consumes stack space. Deep function calls or interrupts could cause a stack overflow, corrupting memory or crashing the system.

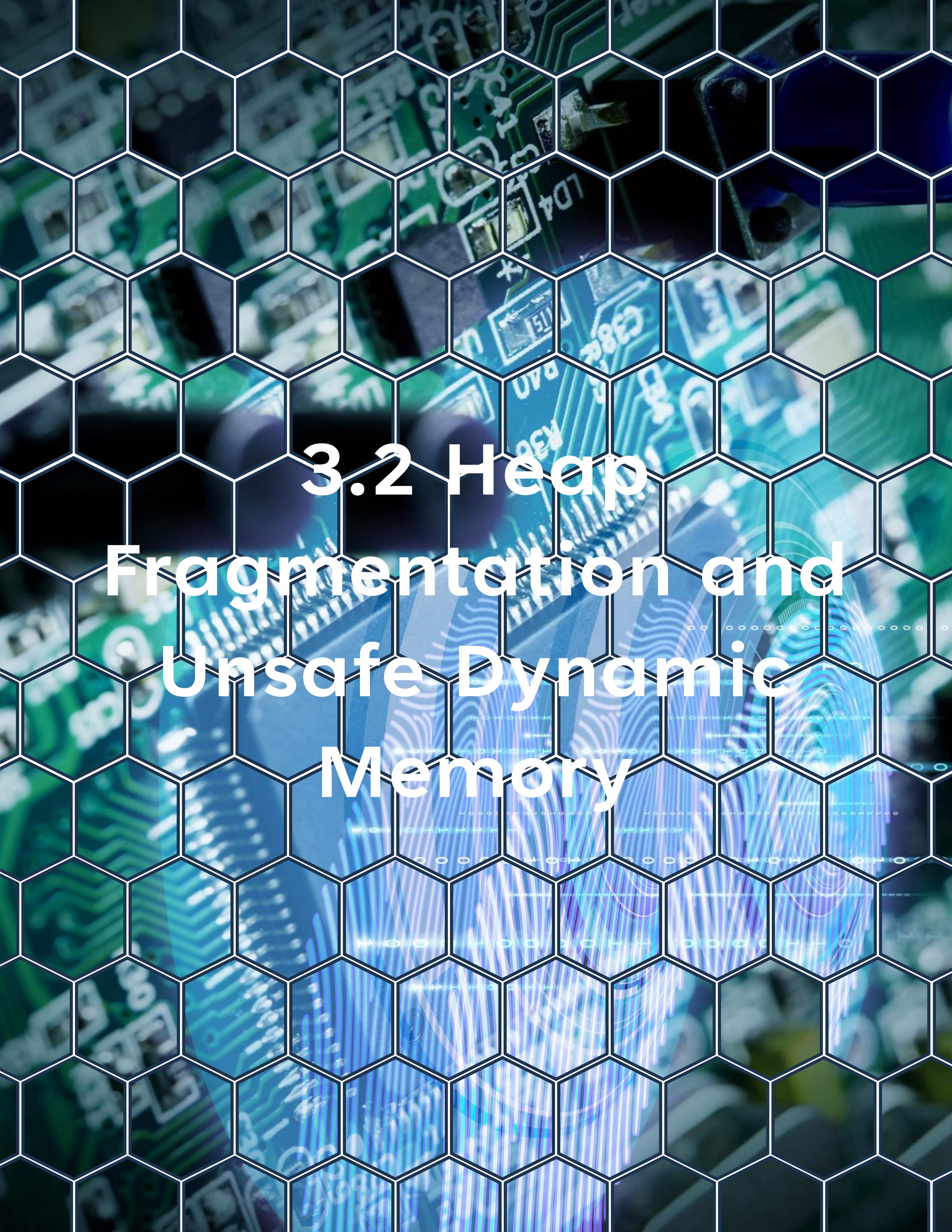
3. Typical Memory Risks in Embedded Development

✅ Safe Function: Static Allocation

```c
#include <avr/io.h>


void safe_function(void) {
    // safe_buffer is allocated in static/global memory, not on
    // the stack, even though it's declared inside the function.
    static uint8_t safe_buffer[512];
    // Operate on statically allocated buffer
    for (uint16_t i = 0; i < 512; i++) {
        safe_buffer[i] = i % 256;
    }
}
```

Why Safer?

The buffer resides in global memory, preserving stack space for function calls and ISRs. This method is predictable, essential for memory-limited MCUs.



3.2 Heap Fragmentation and Unsafe Dynamic Memory

3.2 Heap Fragmentation and Unsafe Dynamic Memory

Although C provides malloc() and free(), dynamic memory management is risky in embedded systems with limited RAM. Fragmentation accumulates over time, leading to allocation failures or unpredictable behavior.

⚠ Unsafe Function: Risk of Heap Fragmentation

```c
#include <stdlib.h>
#include <avr/io.h>

// Function allocating memory dynamically on each call –
// unsafe in small MCUs
void unsafe_dynamic_memory(void) {
    // Allocate 128 bytes on the heap
    uint8_t *buffer = (uint8_t *)malloc(128);

    if (buffer != NULL) {
        // Use the buffer
        for (uint8_t i = 0; i < 128; i++) {
            buffer[i] = i;
        }
        // Free memory, but fragmentation accumulates over time
        free(buffer);
    }
```

3.2 Heap Fragmentation and Unsafe Dynamic Memory

Why Unsafe?

- On MCUs like the **ATtiny1616**, the heap shares memory with the stack in limited SRAM.
- Repeated `malloc()` and `free()` calls cause **fragmentation**, where small unusable gaps form in memory.
- Over time, even if enough total free memory exists, `malloc()` may fail due to fragmentation.
- Heap overflows can corrupt stack space, leading to system crashes or unpredictable behavior.

3.2 Heap Fragmentation and Unsafe Dynamic Memory

✅ Safe Function: Static Memory Allocation

```c
#include <avr/io.h>


void safe_memory_use(void) {
    // The buffer is allocated in static/global memory, not on
    // the stack — even though it resides within the function's
    // code block.
    static uint8_t safe_buffer[128];

    // Use pre-allocated buffer safely
    for (uint8_t i = 0; i < 128; i++) {
        safe_buffer[i] = i;
    }
}
```

Why Safer?

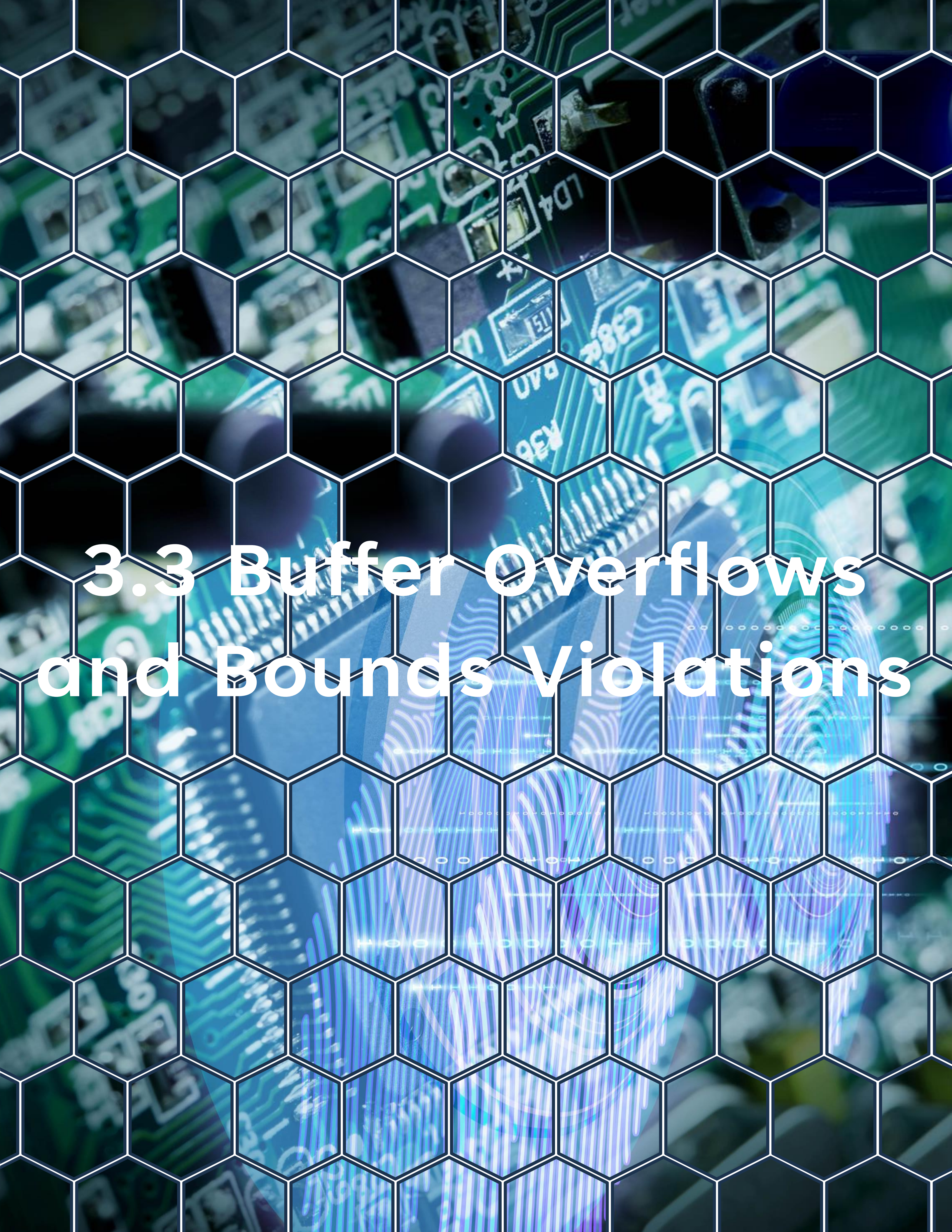
- The buffer resides in fixed global memory — no heap is used.
- Memory layout is predictable, eliminating fragmentation risks.
- afer for systems with tight RAM constraints

3.2 Heap Fragmentation and Unsafe Dynamic Memory

Why Safer?

- The buffer resides in fixed global memory — no heap is used.
- Memory layout is predictable, eliminating fragmentation risks.
- Safer for systems with tight RAM constraints and no memory management hardware.

3.3 Buffer Overflows and Bounds Violations



3.3 Buffer Overflows and Bounds Violations

Failing to enforce array bounds often results in overwriting adjacent memory regions, corrupting data, and destabilizing the system. This is a frequent root cause of severe embedded faults.

⚠️ Unsafe Function: Buffer Overflow Risk

```c
#include <avr/io.h>

#define BUFFER_SIZE 8
uint8_t data_buffer[BUFFER_SIZE];

// Function with bounds violation — causes buffer overflow
void unsafe_write_buffer(void) {
    for (uint8_t i = 0; i <= BUFFER_SIZE; i++) {
        // The loop runs one extra time, writing beyond the
        // buffer's boundary
        data_buffer[i] = i;
    }
}
```

3.3 Buffer Overflows and Bounds Violations

Why Unsafe?

- BUFFER_SIZE is 8, valid indices are 0 to 7.
- The condition i <= BUFFER_SIZE causes the loop to run for i = 8, writing to data_buffer[8], which is outside the allocated memory.
- On MCUs like the ATtiny1616, this corrupts adjacent memory, causing erratic behavior or system crashes.

3.3 Buffer Overflows and Bounds Violations

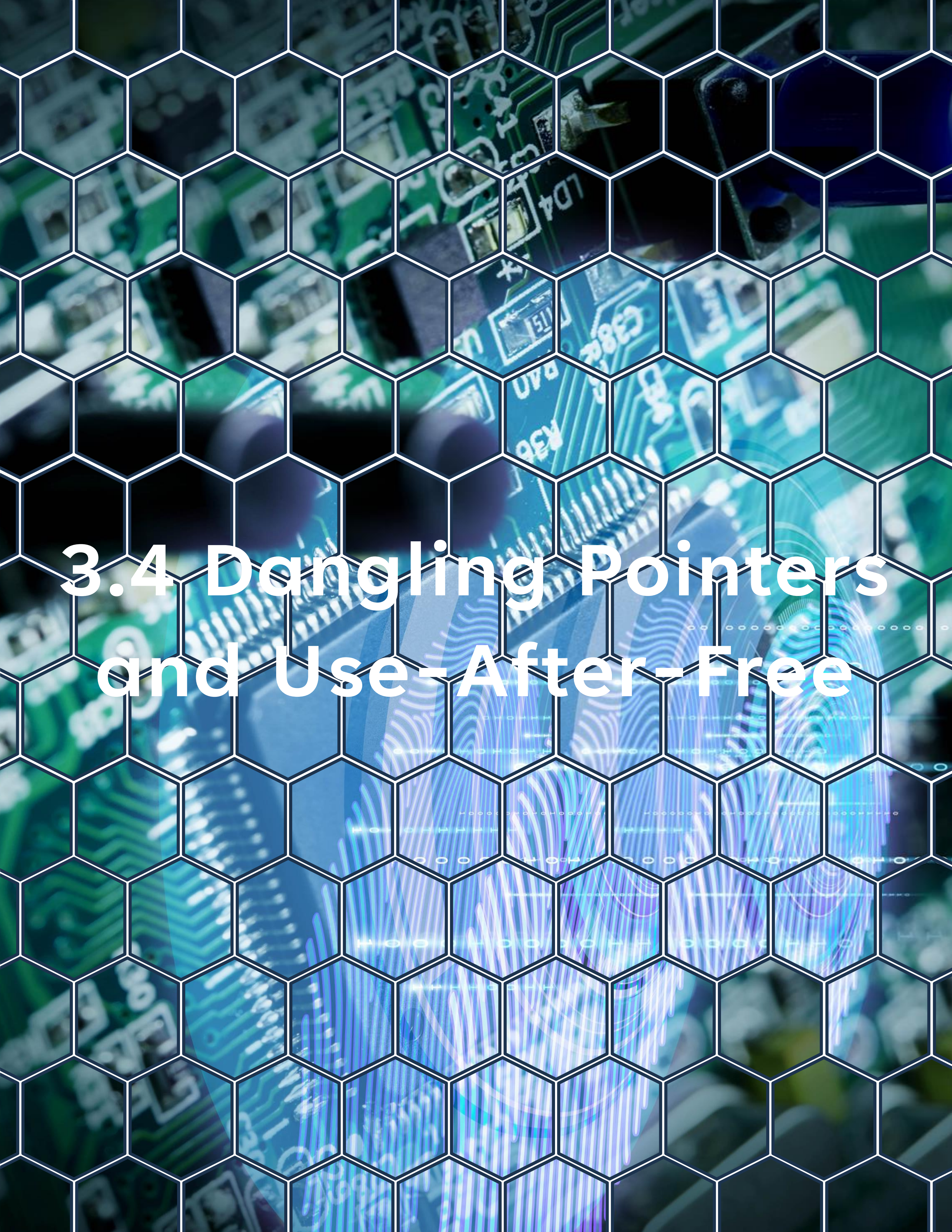
✅ Safe Function: Proper Bounds Checking

```c
#include <avr/io.h>

#define BUFFER_SIZE 8
uint8_t data_buffer[BUFFER_SIZE];

// Correct use — prevents buffer overflow
void safe_write_buffer(void) {
    for (uint8_t i = 0; i < BUFFER_SIZE; i++) {
        // Loop stays within valid indices 0 to 7
        data_buffer[i] = i;
    }
}
```

Why Safer?

- Uses strict `< BUFFER_SIZE` condition.
- Ensures all writes remain within allocated memory.
- Prevents data corruption and improves system reliability.

3.4 Dangling Pointers and Use-After-Free



3.4 Dangling Pointers and Use-After-Free

Dereferencing freed or invalid pointers accesses unpredictable memory content, causing erratic system behavior. Such issues are common in poorly managed dynamic memory or improper pointer handling.

⚠️ Unsafe Function: Dangling Pointer Example

```c
#include <stdlib.h>
#include <avr/io.h>

void unsafe_use_after_free(void) {
    // Allocate 16 bytes on the heap
    uint8_t *ptr = (uint8_t *)malloc(16);

    if (ptr != NULL) {
        ptr[0] = 0x55;  // Use allocated memory
        free(ptr);      // Memory is freed

        ptr[0] = 0xAA;  // Use-after-free — undefined behavior
    }
}
```

3.4 Dangling Pointers and Use-After-Free

Why Unsafe?

- After calling `free(ptr)`, the memory is deallocated.
- Accessing `ptr[0]` after freeing results in undefined behavior.
- May corrupt memory, crash the program, or cause silent faults.
- Particularly dangerous on embedded MCUs like ATtiny1616 with tight memory constraints.

3.4 Dangling Pointers and Use-After-Free

✅ Safe Function: Nullifying Dangling Pointers

```c
#include <stdlib.h>
#include <avr/io.h>

void safe_pointer_handling(void) {
    uint8_t *ptr = (uint8_t *)malloc(16);  // Allocate 16 bytes

    if (ptr != NULL) {
        ptr[0] = 0x55;  // Use memory safely
        free(ptr);      // Free memory

        ptr = NULL;     // Nullify pointer after freeing

        if (ptr != NULL) {
            ptr[0] = 0xAA;  // This block won't execute —
                            // prevents use-after-free
        }
    }
}
```

Why Safer?

- Pointer is set to NULL immediately after freeing.
- Prevents accidental access to invalid

3.4 Dangling Pointers and Use-After-Free

Why Safer?

- Pointer is set to NULL immediately after freeing.
- Prevents accidental access to invalid memory.
- Optional check if (ptr != NULL) ensures safe behavior.
- In embedded systems, it's recommended to avoid dynamic memory altogether, but if used, proper pointer handling is critical.

3.5 Improper Use of volatile

3.5 Improper Use of volatile

Embedded code interacts with hardware registers that can change outside program control (e.g., by peripherals or ISRs). Failure to declare such variables volatile can lead to stale values and faulty logic.

⚠️ Unsafe Function: Missing volatile Keyword

```c
#include <avr/io.h>
#include <avr/interrupt.h>

uint8_t adc_ready_flag = 0;  // Missing 'volatile' — unsafe for shared data

ISR(ADC0_RESRDY_vect) {
    adc_ready_flag = 1;  // Set flag in interrupt
}

void unsafe_check_adc_flag(void) {
    ADC0.CTRLA |= ADC_ENABLE_bm;  // Start ADC

    while (1) {
        if (adc_ready_flag) {  // Compiler may optimize this check incorrectly
            adc_ready_flag = 0;
            uint16_t result = ADC0.RES;
            // Process result
        }
    }
```

3.5 Improper Use of volatile

Why Unsafe?

- adc_ready_flag is modified in the ISR and read in the main loop.
- Without volatile, the compiler may:
    - Cache adc_ready_flag in a register.
    - Skip reading its updated value from memory.
- The main loop may never detect the flag change, causing logic failure.

3.5 Improper Use of volatile

✅ Safe Function: Function with volatile

```c
#include <avr/io.h>
#include <avr/interrupt.h>

volatile uint8_t adc_ready_flag = 0;  // 'volatile' ensures correct memory access

ISR(ADC0_RESRDY_vect) {
    adc_ready_flag = 1;  // Set flag safely in interrupt
}

void safe_check_adc_flag(void) {
    ADC0.CTRLA |= ADC_ENABLE_bm;  // Start ADC

    while (1) {
        if (adc_ready_flag) {  // Compiler reads the actual memory each time
            adc_ready_flag = 0;
            uint16_t result = ADC0.RES;
            // Process result safely
        }
    }
}
```

Why Safer?

- volatile tells the compiler:
  - The variable may change outside normal program flow (e.g., ISR).
  - Always read its latest value from memory.

3.5 Improper Use of volatile

Why Safer?

- volatile tells the compiler:
  - The variable may change outside normal program flow (e.g., ISR).
  - Always read its latest value from memory.
- Ensures reliable flag checking between main code and ISRs.

3.6 Uninitialized Variables and Undefined Behavior

3.6 Uninitialized Variables and Undefined Behavior

Uninitialized stack or global variables contain unpredictable values, resulting in unreliable program behavior or difficult-to-trace bugs.

⚠️ Unsafe Function: Uninitialized Variable Usage

```c
#include <avr/io.h>

void unsafe_uninitialized_use(void) {
    uint8_t counter;  // Uninitialized local variable –
                      // contains random value

    if (counter == 0) {  // Undefined behavior – value is unpredictable
        // Perform some action
        PORTA.OUT |= PIN0_bm;
    }
}
```

Why Unsafe?

Local variables in C are **not automatically initialized.**

3.6 Uninitialized Variables and Undefined Behavior

Why Unsafe?

- Local variables in C are **not automatically initialized.**
- counter contains whatever random data happens to be on the stack.
- Using its value without initialization results in:
  - Unpredictable behavior.
  - Difficult-to-trace bugs, especially on MCUs like ATtiny1616 with shared stack and SRAM.

3.6 Uninitialized Variables and Undefined Behavior

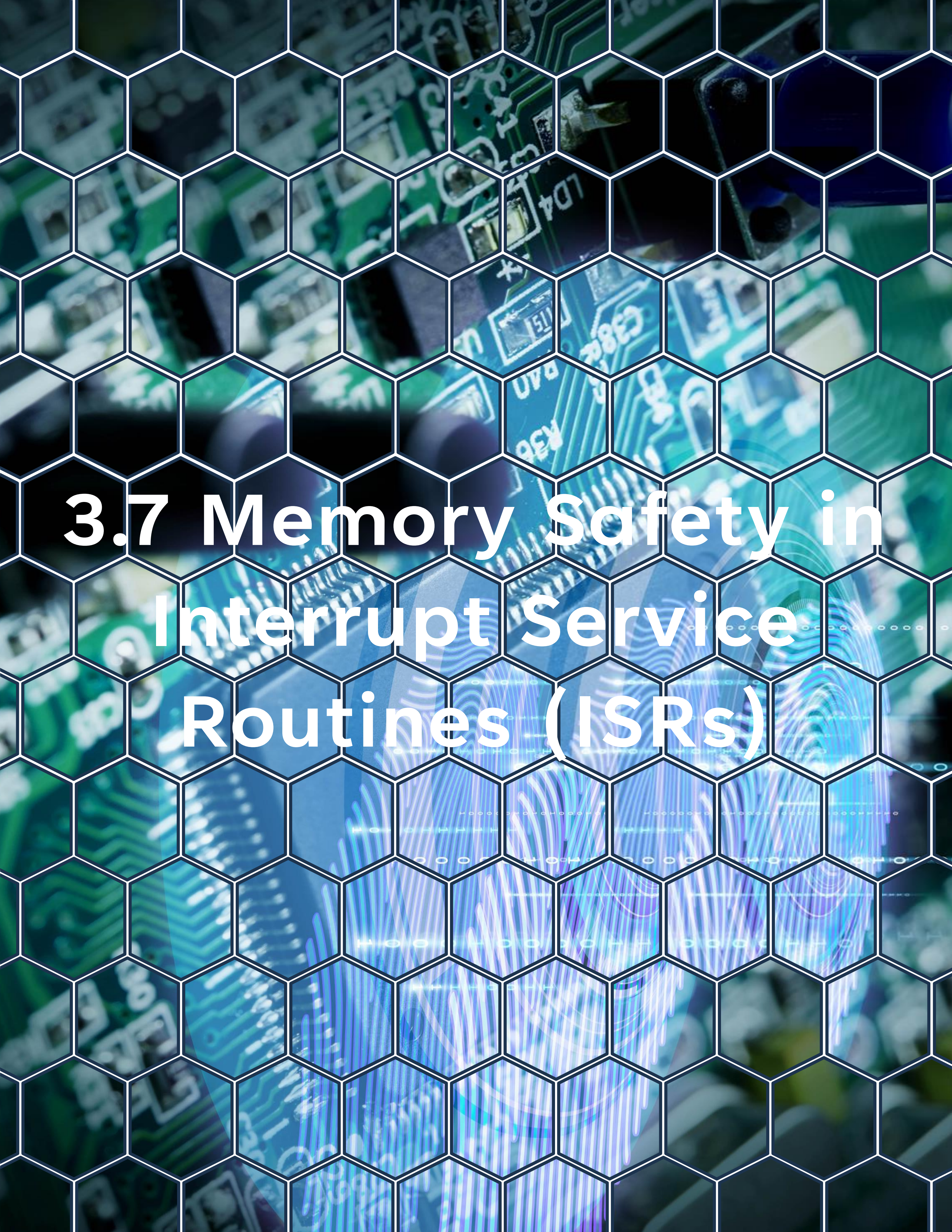
✅ Safe Function: Proper Initialization

```c
#include <avr/io.h>

void safe_initialized_use(void) {
    uint8_t counter = 0;  // Initialize variable explicitly

    if (counter == 0) {  // Behavior is now predictable and safe
        PORTA.OUT |= PIN0_bm;
    }
}
```

Why Safer?

- Variable counter is explicitly initialized to a known, defined value.
- Prevents the system from operating on garbage data.
- Leads to consistent and reliable behavior, essential for embedded reliability.

3.7 Memory Safety in Interrupt Service Routines (ISRs)



3.7 Memory Safety in Interrupt Service Routines (ISRs)

ISRs share memory with the main program. Failing to protect shared data with synchronization mechanisms or improper ISR design can cause race conditions and corruption.

⚠️ Unsafe Function: Shared Data Corruption Risk

```c
#include <avr/io.h>
#include <avr/interrupt.h>

uint8_t shared_counter = 0;  // Shared between main and ISR — no protection

ISR(TCB0_INT_vect) {
    shared_counter++;  // Non-atomic operation — vulnerable to corruption
}

void unsafe_main_loop(void) {
    TCB0.CTRLA |= TCB_ENABLE_bm;  // Start Timer

    while (1) {
        // Read shared variable without disabling interrupts
        if (shared_counter >= 100) {
```

3.7 Memory Safety in Interrupt Service Routines (ISRs)

⚠️ Unsafe Function: Shared Data Corruption Risk

```
#include <avr/io.h>
#include <avr/interrupt.h>

uint8_t shared_counter = 0;  // Shared between main and ISR — no protection

ISR(TCB0_INT_vect) {
    shared_counter++;  // Non-atomic operation — vulnerable to corruption
}

void unsafe_main_loop(void) {
    TCB0.CTRLA |= TCB_ENABLE_bm;  // Start Timer

    while (1) {
        // Read shared variable without disabling interrupts
        if (shared_counter >= 100) {
            shared_counter = 0;  // Non-atomic modification — unsafe
            PORTA.OUT |= PIN0_bm;
        }
    }
}
```

Why Unsafe?

- `shared_counter` is modified both in the ISR and main loop.

3.7 Memory Safety in Interrupt Service Routines (ISRs)

Why Unsafe?

- shared_counter is modified both in the ISR and main loop.
- Increment and assignment are non-atomic on an 8-bit MCU.
- If an interrupt occurs during read-modify-write in the main loop:
  - Data corruption or race conditions may happen.
  - Leads to unpredictable system behavior.

3.7 Memory Safety in Interrupt Service Routines (ISRs)

✅ Safe Function: Atomic Access Protection

```c
#include <avr/io.h>
#include <avr/interrupt.h>

volatile uint8_t shared_counter = 0;  // Marked volatile for safe ISR access

ISR(TCB0_INT_vect) {
    shared_counter++;  // Update counter in ISR
}

void safe_main_loop(void) {
    TCB0.CTRLA |= TCB_ENABLE_bm;  // Start Timer

    while (1) {
        uint8_t local_copy;

        cli();                 // Disable interrupts to ensure atomic read
        local_copy = shared_counter;
        shared_counter = 0;    // Safe reset
        sei();                 // Re-enable interrupts

        if (local_copy >= 100) {
            PORTA.OUT |= PIN0_bm;
        }
    }
}
```

Why Safer?

- Disables interrupts (`cli()`) during critical

3.7 Memory Safety in Interrupt Service Routines (ISRs)

Why Safer?

- Disables interrupts (cli()) during critical section to prevent ISR interference.
- Ensures atomic read and reset of shared_counter.
- Marked volatile to prevent compiler optimizations hiding memory updates.
- Guarantees consistent, corruption-free behavior.

4. Memory-Safe Coding Practices

4. Memory-Safe Coding Practices

4.1 Avoid Dynamic Memory on Small MCUs

For devices like the ATtiny1616, it's advisable to avoid dynamic memory functions (`malloc()`, `free()`) entirely. Static or global allocations provide predictability and prevent fragmentation.

4.2 Static Allocation and Buffer Size Guards

Define arrays and buffers with fixed sizes, ensuring access always stays within bounds. Guard conditions prevent overruns:

```
#define BUFFER_SIZE 16
uint8_t rx_buffer[BUFFER_SIZE];

void store_byte(uint8_t index, uint8_t data) {
    if (index < BUFFER_SIZE) {
```

4. Memory-Safe Coding Practices

4.2 Static Allocation and Buffer Size Guards

Define arrays and buffers with fixed sizes, ensuring access always stays within bounds. Guard conditions prevent overruns:

```
#define BUFFER_SIZE 16
uint8_t rx_buffer[BUFFER_SIZE];

void store_byte(uint8_t index, uint8_t data) {
    if (index < BUFFER_SIZE) {
        rx_buffer[index] = data;
    }
}
```

4.3 Safe Pointer Handling

Always initialize pointers before use and set them to NULL after freeing memory (if dynamic allocation is used):

4. Memory-Safe Coding Practices

4.3 Safe Pointer Handling

Always initialize pointers before use and set them to NULL after freeing memory (if dynamic allocation is used):

```
uint8_t *ptr = NULL;

void example() {
    ptr = get_valid_buffer();
    if (ptr != NULL) {
        *ptr = 0x55;
    }
}
```

4.4 Using volatile Correctly for Hardware Interaction

Declare shared variables modified by ISRs or ardware as volatile to prevent compiler mizations that may cache their values:

4. Memory-Safe Coding Practices

4.4 Using volatile Correctly for Hardware Interaction

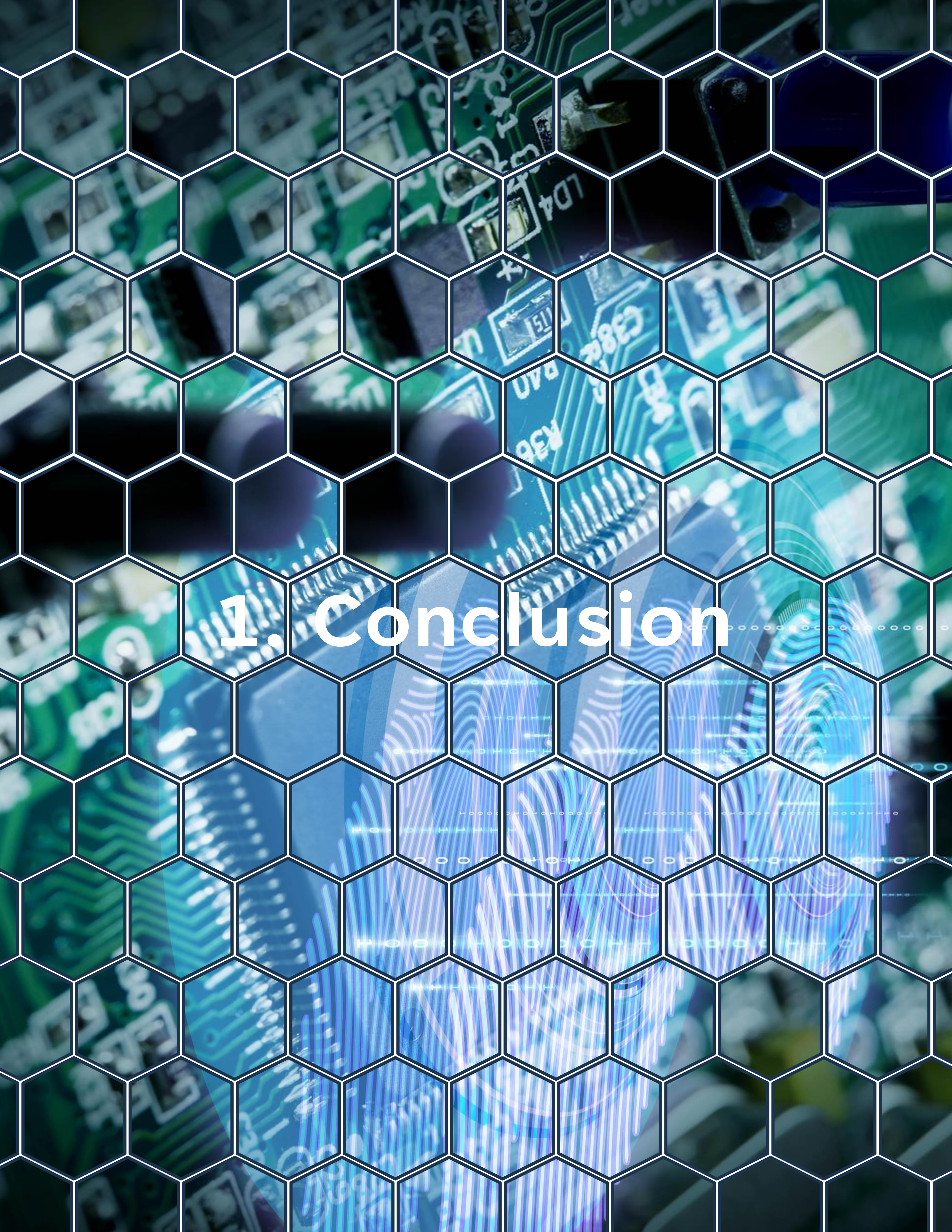
Declare shared variables modified by ISRs or hardware as volatile to prevent compiler optimizations that may cache their values:

```
volatile uint8_t timer_flag = 0;
```

4.5 Compiler Warnings and Static Analysis

Enable maximum compiler warnings and use static analysis tools to detect memory safety issues early:

```
avr-gcc -Wall -Wextra -Wpedantic -O2 -mmcu=attiny1616
```



1. Conclusion

1. Conclusion

Memory safety is the foundation of reliable embedded system development. On platforms like the ATtiny1616, where every byte counts and hardware protections are minimal, developers must adopt disciplined coding practices to prevent elusive bugs and catastrophic failures.

1. Conclusion

By avoiding dynamic memory, using proper bounds checks, handling volatile variables with care, and respecting the limitations of the microcontroller, engineers can build robust, memory-safe firmware. Such attention to detail ensures long-term stability, system reliability, and successful operation in mission-critical environments typical of embedded systems.