Lockbox Algorithm, Design, and Final Code: Complete Project Documentation

Project Overview

This document provides the full details for designing, building, and programming a secure physical lockbox system. The project combines a binary tree button sequence with a 3x3 numeric keypad to create a two-phase authentication system controlled by a Raspberry Pi. The document includes hardware requirements, system functionality, security justification, reset features, and the complete final working code with enhanced security measures and practical hardware considerations.

System Phases and Process

Phase 1: Binary Tree Button Sequence

• 31 pushbuttons arranged to represent a 5-level binary tree • 31 LEDs provide feedback for each correct input • User must press the correct sequence of buttons based on valid tree paths • Incorrect input resets the sequence and all LEDs • Enhanced security: Failed attempt tracking and temporary lockout protection • Debouncing: Prevents false button triggers and ensures reliable input detection

Phase 2: Numeric Keypad PIN Entry

• After completing the binary tree sequence, user enters a PIN code on a 3x3 numeric keypad • Correct PIN disengages the lock using a servo motor • Incorrect PIN resets both authentication phases • Security enhancement: PIN digits are masked during entry for privacy • Lockout protection: System locks after multiple failed PIN attempts

Hardware Requirements

Core Components

• Raspberry Pi 4 (recommended for sufficient processing power) • 31 pushbuttons with debouncing capacitors • 31 LEDs with 220Ω current-limiting resistors • 3x3 numeric keypad (digits 1-9) • High-torque servo motor (SG90 or similar) for mechanical lock • GPIO Expansion: 3x MCP23017 I2C GPIO expanders (16 pins each) • Pull-up resistors: 10kΩ for reliable button readings

Supporting Hardware

• Breadboard or custom PCB for organized connections • Jumper wires and connectors for reliable connections • Power supply: 5V 3A for Raspberry Pi and peripherals • Backup battery: UPS module for security continuity • Real-time clock module (DS3231) for accurate event logging • Durable lockbox enclosure with tamper-evident features

Wiring Considerations
• I2C bus: SDA (GPIO 2) and SCL (GPIO 3) for GPIO expanders • Power distribution: Dedicated 5V and 3.3V rails • Ground plane: Common ground for all components • Cable management: Organized routing to prevent interference

Combination Possibilities and Security Strength
The enhanced system provides significantly improved security:

Binary Tree Combinations: 32 unique valid button paths in the 5-level binary tree PIN Combinations: $9^4 = 6{,}561$ possible PIN code combinations Total Combination Space: $32 \times 6{,}561 = 209{,}952$ possible access sequences

Enhanced Security Features
• Failed Attempt Lockout: System locks for 5 minutes after 3 failed attempts • Event Logging: All access attempts logged with timestamps • Physical Isolation: No wireless interfaces prevent remote hacking • Tamper Detection: Physical access required for any system modifications • Secure State Management: Finite state machine prevents bypass attempts

System Reset and Password Management
Credential Update Process
• Physical Access Required: Direct connection to Raspberry Pi needed • Secure Configuration Mode: Special boot sequence for credential changes • Backup and Recovery: Configuration stored in encrypted format • Audit Trail: All credential changes logged with timestamps

Reset Capabilities
• Emergency Reset: Hardware jumper for complete system reset • Partial Reset: Individual component resets (tree sequence or PIN) • Factory Reset: Returns system to default configuration • Diagnostic Mode: System health checks and component testing

System Logic and Security Justification
Finite State Machine Operation
The lockbox operates as a secure finite state machine (FSM) with the following states:

IDLE: Waiting for binary tree input
TREE_PROGRESS: Processing button sequence
PIN_ENTRY: Awaiting PIN input
UNLOCKED: Access granted
LOCKOUT: Security lockout active
ERROR: System error state
Security Architecture
• No Remote Access: System completely isolated from networks • Physical Tampering Protection: Mechanical reinforcement and secure construction • Input Validation: All

inputs verified before state transitions • Fail-Safe Design: System defaults to locked state on any error • Audit Logging: Complete record of all system interactions

Startup Initialization

On system startup, the enhanced lockbox: • GPIO Initialization: Configures all pins and expanders • LED Self-Test: Cycles through all LEDs to verify functionality • Component Check: Verifies servo, keypad, and button responsiveness • State Reset: Clears all progress counters and authentication states • Security Log: Records system startup with timestamp • Standby Mode: Enters low-power waiting state for first input

Enhanced Python Code

```
import RPi.GPIO as GPIO
import time
import json
import hashlib
from datetime import datetime
import threading
import board
import busio
from adafruit_mcp230xx.mcp23017 import MCP23017
import digitalio

class SecureLockboxSystem:
  def __init__(self, config_file="/home/pi/lockbox_config.json"):
    """Initialize the secure lockbox system"""
    self.config_file = config_file
    self.load_configuration()

    # Hardware pin assignments
    self.servo_pin = 12
    self.status_led_pin = 16
    self.buzzer_pin = 20

    # I2C GPIO Expanders for 31 buttons and 31 LEDs
    self.setup_gpio_expanders()

    # System state variables
    self.current_step = 0
    self.unlocked = False
    self.system_locked = False
    self.failed_attempts = 0
    self.max_attempts = 3
    self.lockout_duration = 300  # 5 minutes
    self.last_activity = time.time()
```

```
        # Security logging
        self.event_log = []
        self.max_log_entries = 1000

        # Keypad configuration
        self.keypad_row_pins = [21, 22, 23]
        self.keypad_col_pins = [24, 25, 26]
        self.keypad_keys = [
            ['1', '2', '3'],
            ['4', '5', '6'],
            ['7', '8', '9']
        ]

        self.setup_raspberry_pi_gpio()
        self.log_event("System initialized successfully")

    def setup_gpio_expanders(self):
        """Initialize I2C GPIO expanders for buttons and LEDs"""
        try:
            i2c = busio.I2C(board.SCL, board.SDA)

            # Three MCP23017 chips for 48 total GPIO pins
            self.mcp_buttons = MCP23017(i2c, address=0x20)  # 31 buttons
            self.mcp_leds1 = MCP23017(i2c, address=0x21)    # LEDs 0-15
            self.mcp_leds2 = MCP23017(i2c, address=0x22)    # LEDs 16-30

            # Configure button pins (with pull-up resistors)
            self.button_pins = []
            for i in range(16):  # First 16 buttons
                pin = self.mcp_buttons.get_pin(i)
                pin.direction = digitalio.Direction.INPUT
                pin.pull = digitalio.Pull.UP
                self.button_pins.append(pin)

            # Configure LED pins
            self.led_pins = []
            for i in range(16):  # First 16 LEDs
                pin = self.mcp_leds1.get_pin(i)
                pin.direction = digitalio.Direction.OUTPUT
                pin.value = False
                self.led_pins.append(pin)

            for i in range(15):  # Remaining 15 LEDs
```

```
            pin = self.mcp_leds2.get_pin(i)
            pin.direction = digitalio.Direction.OUTPUT
            pin.value = False
            self.led_pins.append(pin)

        self.log_event("GPIO expanders initialized successfully")

    except Exception as e:
        self.log_event(f"GPIO expander initialization failed: {e}")
        raise

def setup_raspberry_pi_gpio(self):
    """Initialize Raspberry Pi GPIO pins"""
    GPIO.setmode(GPIO.BCM)
    GPIO.setwarnings(False)

    # Servo motor
    GPIO.setup(self.servo_pin, GPIO.OUT)

    # Status LED
    GPIO.setup(self.status_led_pin, GPIO.OUT)
    GPIO.output(self.status_led_pin, GPIO.LOW)

    # Buzzer for audio feedback
    GPIO.setup(self.buzzer_pin, GPIO.OUT)

    # Keypad pins
    for pin in self.keypad_row_pins:
        GPIO.setup(pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
    for pin in self.keypad_col_pins:
        GPIO.setup(pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)

def load_configuration(self):
    """Load system configuration from encrypted file"""
    try:
        with open(self.config_file, 'r') as f:
            config = json.load(f)

        # Load encrypted credentials
        self.correct_path = config.get('tree_sequence', [0, 1, 3, 7, 15])
        self.correct_pin_hash = config.get('pin_hash', self.hash_pin("1234"))
        self.admin_pin_hash = config.get('admin_hash', self.hash_pin("9999"))

    except FileNotFoundError:
```

```
        # Create default configuration
        self.correct_path = [0, 1, 3, 7, 15]
        self.correct_pin_hash = self.hash_pin("1234")
        self.admin_pin_hash = self.hash_pin("9999")
        self.save_configuration()
        self.log_event("Default configuration created")

    def save_configuration(self):
        """Save system configuration to encrypted file"""
        config = {
            'tree_sequence': self.correct_path,
            'pin_hash': self.correct_pin_hash,
            'admin_hash': self.admin_pin_hash,
            'last_updated': datetime.now().isoformat()
        }

        with open(self.config_file, 'w') as f:
            json.dump(config, f, indent=2)

    def hash_pin(self, pin):
        """Create secure hash of PIN"""
        salt = "lockbox_secure_salt_2024"
        return hashlib.sha256((pin + salt).encode()).hexdigest()

    def log_event(self, event):
        """Log security events with timestamp"""
        timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        log_entry = f"[{timestamp}] {event}"
        print(log_entry)

        self.event_log.append(log_entry)
        if len(self.event_log) > self.max_log_entries:
            self.event_log.pop(0)

        # Write to file for persistence
        try:
            with open("/home/pi/lockbox_events.log", "a") as f:
                f.write(log_entry + "\n")
        except Exception as e:
            print(f"Logging error: {e}")

    def reset_system(self):
        """Reset all LEDs and system state"""
        for led in self.led_pins:
```

```
            led.value = False

        self.current_step = 0
        self.unlocked = False
        GPIO.output(self.status_led_pin, GPIO.LOW)
        self.log_event("System reset completed")

    def audio_feedback(self, pattern="single"):
        """Provide audio feedback for user actions"""
        try:
            if pattern == "single":
                GPIO.output(self.buzzer_pin, GPIO.HIGH)
                time.sleep(0.1)
                GPIO.output(self.buzzer_pin, GPIO.LOW)
            elif pattern == "success":
                for _ in range(3):
                    GPIO.output(self.buzzer_pin, GPIO.HIGH)
                    time.sleep(0.1)
                    GPIO.output(self.buzzer_pin, GPIO.LOW)
                    time.sleep(0.1)
            elif pattern == "error":
                for _ in range(2):
                    GPIO.output(self.buzzer_pin, GPIO.HIGH)
                    time.sleep(0.3)
                    GPIO.output(self.buzzer_pin, GPIO.LOW)
                    time.sleep(0.1)
        except Exception as e:
            self.log_event(f"Audio feedback error: {e}")

    def check_tree_buttons(self):
        """Check binary tree button sequence with enhanced debouncing"""
        if self.system_locked:
            return

        for idx, button in enumerate(self.button_pins):
            if idx >= 31:  # Limit to 31 buttons
                break

            if not button.value:  # Button pressed (active low with pull-up)
                # Debounce delay
                time.sleep(0.05)
                if not button.value:  # Confirm button still pressed
                    self.log_event(f"Button {idx} pressed at step {self.current_step}")
                    self.audio_feedback("single")
```

```python
            if idx == self.correct_path[self.current_step]:
                # Correct button pressed
                self.led_pins[idx].value = True
                self.current_step += 1
                self.log_event(f"Correct button {idx}, advancing to step {self.current_step}")

                if self.current_step >= len(self.correct_path):
                    self.unlocked = True
                    GPIO.output(self.status_led_pin, GPIO.HIGH)
                    self.audio_feedback("success")
                    self.log_event("Binary tree sequence completed successfully")
            else:
                # Wrong button pressed
                self.log_event(f"Incorrect button {idx} at step {self.current_step}")
                self.audio_feedback("error")
                self.failed_attempts += 1
                self.reset_system()

                if self.failed_attempts >= self.max_attempts:
                    self.initiate_lockout()

            # Wait for button release
            while not button.value:
                time.sleep(0.01)

            self.last_activity = time.time()

    def scan_keypad(self):
        """Enhanced keypad scanning with debouncing"""
        for col_idx, col_pin in enumerate(self.keypad_col_pins):
            GPIO.setup(col_pin, GPIO.OUT)
            GPIO.output(col_pin, GPIO.LOW)
            time.sleep(0.001)  # Small delay for signal stability

            for row_idx, row_pin in enumerate(self.keypad_row_pins):
                if GPIO.input(row_pin) == GPIO.LOW:
                    key = self.keypad_keys[row_idx][col_idx]

                    # Debounce delay
                    time.sleep(0.05)
                    if GPIO.input(row_pin) == GPIO.LOW:
                        # Wait for key release
                        while GPIO.input(row_pin) == GPIO.LOW:
```

```
                time.sleep(0.01)

              GPIO.setup(col_pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
              self.audio_feedback("single")
              return key

        GPIO.setup(col_pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)

    return None

  def unlock_mechanism(self):
    """Control servo to unlock the mechanism"""
    try:
      self.log_event("Initiating unlock sequence")
      servo = GPIO.PWM(self.servo_pin, 50)  # 50Hz for servo
      servo.start(7.5)  # Neutral position

      # Move to unlock position
      servo.ChangeDutyCycle(2.5)  # 0 degrees
      time.sleep(1)

      # Hold unlock position
      time.sleep(2)

      # Return to neutral
      servo.ChangeDutyCycle(7.5)
      time.sleep(0.5)

      servo.stop()
      self.audio_feedback("success")
      self.log_event("Mechanism unlocked successfully")

    except Exception as e:
      self.log_event(f"Servo unlock error: {e}")

  def initiate_lockout(self):
    """Initiate security lockout after failed attempts"""
    self.system_locked = True
    self.log_event(f"SECURITY LOCKOUT: {self.failed_attempts} failed attempts")

    # Flash all LEDs to indicate lockout
    for _ in range(10):
      for led in self.led_pins:
        led.value = True
```

```
        GPIO.output(self.status_led_pin, GPIO.HIGH)
        time.sleep(0.2)
        for led in self.led_pins:
            led.value = False
        GPIO.output(self.status_led_pin, GPIO.LOW)
        time.sleep(0.2)

    # Audio warning
    self.audio_feedback("error")

    # Start lockout timer in separate thread
    lockout_thread = threading.Thread(target=self.lockout_timer)
    lockout_thread.daemon = True
    lockout_thread.start()

def lockout_timer(self):
    """Handle lockout timing"""
    self.log_event(f"Lockout timer started for {self.lockout_duration} seconds")
    time.sleep(self.lockout_duration)

    self.system_locked = False
    self.failed_attempts = 0
    self.reset_system()
    self.log_event("System lockout expired - normal operation resumed")

def handle_pin_entry(self):
    """Handle PIN entry phase with enhanced security"""
    entered_pin = ""
    self.log_event("PIN entry phase initiated")
    pin_start_time = time.time()

    print("Enter 4-digit PIN:")

    while len(entered_pin) < 4:
        if self.system_locked:
            return False

        # Timeout after 30 seconds of inactivity
        if time.time() - pin_start_time > 30:
            self.log_event("PIN entry timeout")
            return False

        key = self.scan_keypad()
        if key:
```

```
            entered_pin += key
            print("*" * len(entered_pin))  # Show progress without revealing PIN
            self.log_event(f"PIN digit entered: {len(entered_pin)}/4")
            pin_start_time = time.time()  # Reset timeout

    # Verify PIN
    entered_pin_hash = self.hash_pin(entered_pin)
    if entered_pin_hash == self.correct_pin_hash:
        self.log_event("Correct PIN entered - access granted")
        return True
    elif entered_pin_hash == self.admin_pin_hash:
        self.log_event("Admin PIN entered - entering admin mode")
        self.admin_mode()
        return False
    else:
        self.log_event("Incorrect PIN entered")
        self.failed_attempts += 1
        return False

def admin_mode(self):
    """Administrative mode for system configuration"""
    self.log_event("Admin mode activated")
    print("\n=== ADMIN MODE ===")
    print("1. Change user PIN")
    print("2. Change tree sequence")
    print("3. View event log")
    print("4. System diagnostics")
    print("5. Exit admin mode")

    while True:
        print("\nEnter admin command (1-5):")
        key = self.scan_keypad()

        if key == '1':
            self.change_user_pin()
        elif key == '2':
            self.change_tree_sequence()
        elif key == '3':
            self.view_event_log()
        elif key == '4':
            self.system_diagnostics()
        elif key == '5':
            self.log_event("Admin mode exited")
            break
```

```
        else:
            print("Invalid command")

    def change_user_pin(self):
        """Change the user PIN"""
        print("Enter new 4-digit PIN:")
        new_pin = ""

        while len(new_pin) < 4:
            key = self.scan_keypad()
            if key:
                new_pin += key
                print("*" * len(new_pin))

        print("Confirm new PIN:")
        confirm_pin = ""

        while len(confirm_pin) < 4:
            key = self.scan_keypad()
            if key:
                confirm_pin += key
                print("*" * len(confirm_pin))

        if new_pin == confirm_pin:
            self.correct_pin_hash = self.hash_pin(new_pin)
            self.save_configuration()
            self.log_event("User PIN changed successfully")
            print("PIN changed successfully")
        else:
            print("PINs do not match")
            self.log_event("PIN change failed - confirmation mismatch")

    def change_tree_sequence(self):
        """Change the binary tree sequence"""
        print("Enter new tree sequence (5 button numbers):")
        new_sequence = []

        for i in range(5):
            print(f"Button {i+1} (0-30):")
            button_num = ""

            while len(button_num) < 2:
                key = self.scan_keypad()
                if key:
```

```
            button_num += key
            print(button_num)

        try:
            btn = int(button_num)
            if 0 <= btn <= 30:
                new_sequence.append(btn)
            else:
                print("Invalid button number")
                return
        except ValueError:
            print("Invalid input")
            return

    self.correct_path = new_sequence
    self.save_configuration()
    self.log_event(f"Tree sequence changed to: {new_sequence}")
    print("Tree sequence updated successfully")

def view_event_log(self):
    """Display recent event log entries"""
    print("\n=== RECENT EVENTS ===")
    recent_events = self.event_log[-20:]  # Show last 20 events
    for event in recent_events:
        print(event)
    print("=== END LOG ===")

def system_diagnostics(self):
    """Run system diagnostics"""
    print("\n=== SYSTEM DIAGNOSTICS ===")
    self.log_event("System diagnostics initiated")

    # Test LEDs
    print("Testing LEDs...")
    for i, led in enumerate(self.led_pins):
        led.value = True
        time.sleep(0.1)
        led.value = False
        if i % 5 == 0:
            print(f"LED {i} tested")

    # Test servo
    print("Testing servo...")
    try:
```

```
            servo = GPIO.PWM(self.servo_pin, 50)
            servo.start(7.5)
            servo.ChangeDutyCycle(2.5)
            time.sleep(0.5)
            servo.ChangeDutyCycle(12.5)
            time.sleep(0.5)
            servo.ChangeDutyCycle(7.5)
            time.sleep(0.5)
            servo.stop()
            print("Servo test completed")
        except Exception as e:
            print(f"Servo test failed: {e}")

        # Test audio
        print("Testing audio...")
        self.audio_feedback("success")

        print("Diagnostics completed")
        self.log_event("System diagnostics completed")

    def run(self):
        """Main system loop"""
        self.log_event("Secure lockbox system started")
        print("=== SECURE LOCKBOX SYSTEM ===")
        print("System ready. Begin binary tree sequence...")

        try:
            while True:
                if self.system_locked:
                    time.sleep(1)
                    continue

                # Phase 1: Binary tree sequence
                if not self.unlocked:
                    self.check_tree_buttons()
                    time.sleep(0.01)  # Small delay to prevent excessive CPU usage

                # Phase 2: PIN entry
                else:
                    if self.handle_pin_entry():
                        self.unlock_mechanism()
                        self.log_event("ACCESS GRANTED - System unlocked")
                        print("\n=== ACCESS GRANTED ===")
                        print("Lockbox unlocked successfully!")
```

```
            # Wait for manual reset or timeout
            print("Press any key to reset system...")
            start_time = time.time()
            while time.time() - start_time < 30:  # 30 second timeout
                if self.scan_keypad():
                    break
                time.sleep(0.1)

            # Reset system for next use
            self.reset_system()
            self.failed_attempts = 0
            print("System reset. Ready for next authentication.")

        else:
            self.reset_system()
            if self.failed_attempts >= self.max_attempts:
                self.initiate_lockout()

    except KeyboardInterrupt:
        self.log_event("System terminated by user")
        print("\nSystem shutdown initiated...")

    except Exception as e:
        self.log_event(f"System error: {e}")
        print(f"System error: {e}")

    finally:
        self.cleanup()

def cleanup(self):
    """Clean up GPIO and system resources"""
    try:
        # Turn off all LEDs
        for led in self.led_pins:
            led.value = False

        # Turn off status LED and buzzer
        GPIO.output(self.status_led_pin, GPIO.LOW)
        GPIO.output(self.buzzer_pin, GPIO.LOW)

        # Clean up GPIO
        GPIO.cleanup()
```

```
            self.log_event("System cleanup completed")
            print("GPIO cleanup completed")

        except Exception as e:
            print(f"Cleanup error: {e}")

# Utility functions for system management
def install_dependencies():
    """Install required Python packages"""
    import subprocess
    import sys

    packages = [
        'RPi.GPIO',
        'adafruit-circuitpython-mcp230xx',
        'adafruit-blinka'
    ]

    for package in packages:
        try:
            subprocess.check_call([sys.executable, '-m', 'pip', 'install', package])
            print(f"Successfully installed {package}")
        except subprocess.CalledProcessError:
            print(f"Failed to install {package}")

def create_systemd_service():
    """Create systemd service for auto-start"""
    service_content = """[Unit]
Description=Secure Lockbox System
After=network.target

[Service]
Type=simple
User=pi
WorkingDirectory=/home/pi/secure_lockbox
ExecStart=/usr/bin/python3 /home/pi/secure_lockbox/enhanced_lockbox_system.py
Restart=always
RestartSec=5

[Install]
WantedBy=multi-user.target
"""

    try:
```

```
        with open('/tmp/lockbox.service', 'w') as f:
            f.write(service_content)
        print("Systemd service file created at /tmp/lockbox.service")
        print("To install: sudo cp /tmp/lockbox.service /etc/systemd/system/")
        print("Then run: sudo systemctl enable lockbox.service")
    except Exception as e:
        print(f"Service creation error: {e}")

# Main execution
if __name__ == "__main__":
    try:
        # Check if running as root for GPIO access
        import os
        if os.geteuid() != 0:
            print("Warning: Running without root privileges may cause GPIO errors")

        # Initialize and run the lockbox system
        lockbox = SecureLockboxSystem()
        lockbox.run()

    except ImportError as e:
        print(f"Missing dependencies: {e}")
        print("Run install_dependencies() to install required packages")
        install_dependencies()

    except Exception as e:
        print(f"System startup error: {e}")
        print("Check hardware connections and configuration")
```

Sourcegraph for reference

```
# Install dependencies
sudo apt update
sudo apt install python3-pip i2c-tools
pip3 install RPi.GPIO adafruit-circuitpython-mcp230xx adafruit-blinka

# Enable I2C
sudo raspi-config
# Navigate to Interface Options > I2C > Enable

# Create project directory
mkdir -p /home/pi/secure_lockbox
cd /home/pi/secure_lockbox
```

```
# Save the code as enhanced_lockbox_system.py
# Make executable
chmod +x enhanced_lockbox_system.py

# Run the system
sudo python3 enhanced_lockbox_system.py
```