# Practical Evaluation of Code Injection in Encrypted Firmware Updates

Oscar M. Guillen*, Dawin Schmidt† and Georg Sigl*
*Technische Universität München, Germany
oscar.guillen@tum.de, sigl@tum.de
†Ludwig-Maximilians-Universität München, Germany
schmidtdaw@cip.ifi.lmu.de

***Abstract*—Several firmware update mechanisms in microcontrollers still make use of confidentiality-only block cipher modes, ultimately lulling the users into a false sense of security. In this work we show how easy it is to apply well known malleability attacks to successfully inject arbitrary code into an encrypted firmware image. We demonstrate this vulnerability by attacking the Advanced Encryption Standard in Cipher Block Chaining mode on an ARM-based microcontroller. The attack makes use of patterns in the structure of the firmware image to obtain known-plaintexts which may be used to modify an encrypted image. Subsequently, malicious code may be injected to extract the memory contents of the device. This work shall help motivate the use of authenticated encryption modes even in resource constrained devices.**

## I. Introduction

Firmware update mechanisms have become an essential tool for developers of embedded devices. The need to keep the code of a device up-to-date, even after the device has been deployed, has led to the creation of remote firmware update solutions. These programs, called bootloaders in the microcontrollers jargon, enable writing to, and in some cases reading from, the internal memory of a device through standard communication interfaces. In order to protect firmware images from being eavesdropped on while being transmitted, embedded software developers rely on the use of encryption algorithms. While powerful devices such as application processors are able to run public key algorithms without noticeable performance and memory usage penalties, resource constrained devices such as general-purpose microcontrollers must rely mostly on solutions that are based on symmetric encryption algorithms. Many solutions make use of confidentiality-only block cipher modes. These modes protect the confidentiality of the content through encryption but are not designed to provide a way to verify the integrity or authenticity of the encrypted data. Examples of bootloaders which make use of these modes can be found even in the code examples provided by some manufacturers [1], [2], [8]. However, as we show in this work, failing to provide integrity verification, these modes are susceptible to a class of attacks known as malleability attacks [4]. Even when one may only care about preserving the secrecy of the firmware image, making use of a malleable block cipher mode may be exploited to ultimately lead to the disclosure of this information.

**Contributions** We present a way to bypass the protection provided by encryption in a firmware update scenario through a malleability attack. We demonstrate how malicious code may be inserted into an encrypted firmware. In addition, we demonstrate how an attacker would be able to get a hold of the plaintext firmware without having to know the secret key, showing that not even the secrecy of the firmware is guaranteed when using confidentiality-only modes. In contrast to previous attacks on malleable ciphers [11]–[13], this attack does not affect a specific protocol. It relies on patterns found in firmware images determined by the underlying processor architecture. We made use of the ARM architecture for our example as it is a widely known and used architecture for embedded devices. The results of this practical attack should help sensitize designers about the problems of using confidentiality-only cipher modes and urge them to make use of authenticated encryption (AE). And at the same time reinforce the need of proposals such as the CAESAR competition [3], which aims to identify novel AE algorithms which are suitable for widespread adoption.

**Outline** The paper is organized as follows: Section II presents the background information for the attack. Section III describes the attack scenario in the context of encrypted firmware updates. Section IV explains the details of the practical implementation of the attack. Finally, we end with the conclusions in Section V.

## II. Background

We assume the reader is familiar with the concepts of symmetric encryption and block cipher modes. In this section we give a short review of the Cipher Block Chaining (CBC) mode and the security weaknesses that it exposes.

### A. Cipher Block Chaining (CBC) Mode

The Cipher Block Chaining mode is a widely used confidentiality-only mode for symmetric block ciphers. It is one of the recommended modes for the Advanced Encryption Standard (AES) [10] and was first standardized more than 30 years ago for its use with the Data Encryption Standard (DES) [9]. To prevent the creation of identical ciphertexts in blocks which contain the same data, in this mode the patterns in the plaintext are randomized by chaining the ciphertext output of a block to the input of the next one, and introducing a random

978-3-9815370-6-2/DATE16/

initialization vector (IV) for the first block. This random value is linearly combined with the plaintext before the encryption of the first block with the use of an exclusive-or operation. Subsequent blocks combine the ciphertext generated by the encryption of the previous block with the input plaintext. Hence, the resulting ciphertext is not only influenced by the plaintext, but by the previously computed block values. As long as the IV is random, the output ciphertext may not be distinguished from random.

*1) Encryption:* We define encryption with a block cipher as $E$ such that $E : K \times \{0,1\}^n \rightarrow \{0,1\}^n$. Let us denote a plaintext message $P$ with size $m$, which is formed by several blocks $P_i$ each of size $n$-bits (where $n = 128$ in the case of AES). We denote $C$ as the resulting ciphertext, $K$ as the secret key used by the encryption algorithm (AES supports key lengths of 128, 192 and 256-bits) and $IV$ as a random initialization vector $IV = \{0,1\}^n$. Encryption with CBC is defined in Algorithm 1.

**Algorithm 1** Encryption using Cipher Block Chaining with a random IV.

**function** $CBCenc_K^{IV}(P)$
  $P_1 \dots P_m \leftarrow P$ where $|P_i| = n$
  $C_0 \leftarrow IV$
  **for** $i \leftarrow 1$ to $m$ **do**
    $C_i \leftarrow E_K(P_i \oplus C_{i-1})$
  **end for**
  $C \leftarrow C_1 \dots C_m$
  **return** $C$
**end function**

*2) Decryption:* To perform the decryption of the ciphertext the inverse operation is performed. The ciphertext is first fed into the input of the decryption routine of the block cipher, denoted as $D$ and which is the inverse function of $E$. The decrypted block is then combined by means of an exclusive-or with the previous ciphertext block. As well as for the encryption process, the IV is used as the first block in this sequence. The algorithm for decryption with CBC is shown in Algorithm 2.

**Algorithm 2** Decryption using Cipher Block Chaining with a random IV.

**function** $CBCdec_K^{IV}(C)$
  $C_1 \dots C_m \leftarrow C$ where $|C_i| = n$
  $C_0 \leftarrow IV$
  **for** $i \leftarrow 1$ to $m$ **do**
    $P_i \leftarrow D_K(C_i) \oplus C_{i-1}$
  **end for**
  $P \leftarrow P_1 \dots P_m$
  **return** $P$
**end function**

### B. Weaknesses of CBC Mode

CBC is considered secure in the sense that the output ciphertext is indistinguishable from random when a random IV is used. However, there are problems that arise when the mode is used for practical applications. The most critical one being that the ciphertexts are malleable. This means that an attacker is able to modify a ciphertext in such a way that, after decryption, the new plaintext will result in a meaningful value. As the integrity of the plaintext cannot be guaranteed, an attacker is capable of performing these modifications without being detected. The attacks that exploit this property are called malleability attacks. The following paragraphs describe the possible modifications which are important for our purposes. These are a subset of the attacks that are also applicable to hard disk encryption [5]. The practical implementation of these attacks is presented in Section III.

*1) Data Modification:* During decryption, the content of a plaintext block $P_i$ is linearly dependent to the value of the previous ciphertext block $C_{i-1}$ as it is connected through an exclusive-or operation. Modifications to a bit at position $j$ in the ciphertext $C_{i-1}$ will be propagated to the bit on the same position $j$ in the plaintext of the next block $P_i$. This allows an attacker to perform modifications which target specific plaintext blocks. Modifying a ciphertext block will also result in an erroneous value in the plaintext for the respective block, however the error is not propagated to the rest of the ciphertext blocks. The first work to tackle this problem may be traced back to [4], where the concept of non-malleable cryptography was introduced. A common attack related to this weakness is known as a *Bit Flipping* attack. An attacker may modify a single bit $j$ in a ciphertext block $C_{i-1}$. After decryption the bit flip in the ciphertext is reflected as a bit flip in the same bit position $j$ of plaintext block $P_i$. This can be seen in Figure 1a. Thus, an attacker is able to perform controlled modifications to the ciphertext if the attacker knows parts of the plaintext block $P_i$. The most common Bit Flipping attack is that of a Padding Oracle [11]–[14]. In general, the attack model is known as a Known-Plaintext Forgery (KPF) which describes the ability of an attacker to create a valid forgery if parts of the underlying plaintext are known [6].

As defined in Section II-A, decryption of a ciphertext block in CBC mode can be expressed as:

$$P_i = D_K(C_i) \oplus C_{i-1}$$

By knowing the value of the plaintext block $P_i$ and having control over $C_{i-1}$ we are able to control the value after decryption:

$$D_K(C_i) = P_i \oplus C_{i-1}$$

To transform the plaintext $P_i$ into the value $P_i^*$, we can determine a value $v$,

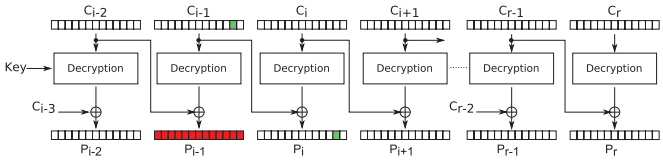
$$v = P_i^* \oplus P_i$$

and add this value to the ciphertext block,

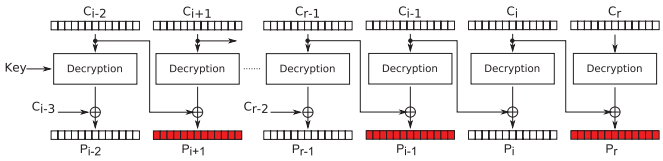
$$C_{i-1}^* = C_{i-1} \oplus v$$

such that:

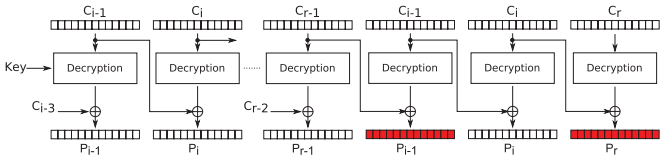
$$\begin{aligned} P_i^* &= P_i \oplus v \\ &= D_K(C_i) \oplus C_{i-1} \oplus v \end{aligned}$$




(a) *Data modification.* A bit is flipped in the second less significant byte of $C_{i-1}$, the modification is propagated to the same bit in the second byte of $P_i$, this is shown in green. The block in red constitutes the plaintext which would result into a non-predictable value after a bit is flipped in the ciphertext.




(b) *Cut-and-Paste.* Blocks $C_{i-1}$ and $C_i$ are cut from their original location and pasted between blocks $C_{r-1}$ and $C_r$. The blocks in red constitute the plaintext which would result into non-predictable values after the shift.




(c) *Copy-and-Paste.* Blocks $C_{i-1}$ and $C_i$ are copied and pasted in between blocks $C_{r-1}$ and $C_r$. The blocks in red constitute the plaintext which would result into non-predictable values after the copy.

Figure 1. CBC mode weaknesses.

We call block $i-1$ the *control block* and block $i$ the *target block*, since controlling the value in $C_{i-1}$ leads to the manipulation of the plaintext message $P_i$.

*2) Cut-and-Paste:* The sequence of the blocks may be modified to change the order of the plaintext data. By moving the ciphertext of the control and target blocks $C_{i-1}$ and $C_i$ to a different location in the ciphertext, the plaintext content of the target block $P_i$ can be moved to a specific location. This modification causes the corruption of block $P_{i+1}$ at the source. When blocks $C_{i-1}$ and $C_i$ are placed blocks $C_{r-1}$ and $C_r$, the block $P_r$ and the plaintext of the inserted control block, $P_{i-1}$, will be corrupted as well. The rest of the plaintext is left unaltered. This is shown in Figure 1b.

*3) Copy-and-Paste:* A similar attack to Cut-and-Paste, in which plaintext blocks are replicated by copying cipher blocks and pasting them at different locations. Using this technique, the content in the sequential plaintext blocks $P_i$ to $P_q$ may be replicated as many times as wanted by copying the respective sequential ciphertext blocks $C_{i-1}$ to $C_q$, where $q \geq i$, and pasting them to another location. This can be seen for the case of a single plaintext block $P_i$ in Figure 1c, where $C_{i-1}$ and $C_i$ are pasted between $C_{r-1}$ and $C_r$. This operation causes plaintexts $P_{i-1}$ and $P_r$ to be corrupted.

## III. Practical Code Injection

In this section we explain the details of a CBC-Malleability attack against encrypted firmware updates. We place focus on the ARM architecture, but these techniques may be applied to different processor architectures in a similar way.

### A. Attack Scenario

For our attack scenario we assume that an attacker has access to an encrypted firmware image and the device to which this image should be loaded. Due to the use of encryption, the attacker does not have access to the plaintext content of the payload. The attacker only knows the architecture of the CPU and the structure of the compiler generated code. We assume the firmware image has been encrypted making use of a confidentiality-only mode, more specifically CBC mode.

It is the attacker's goal to find a way to obtain the plaintext content of the payload, that is, the firmware image. In this section we will show how an attacker may be able to inject her own malicious code into the encrypted firmware image. Once the manipulated image has been loaded onto the device, the attacker may be able to extract the content of the device. This includes the firmware image which was just uploaded and may also comprise other sensitive data such as previously stored encryption keys.

To be able to perform the attack previously described, an attacker requires two conditions to be fulfilled:

(a) A block $i$ with known plaintext.
(b) A block $i$-1 which can be changed without being detected and without causing an impact on the attacked program.

In the following, possible attack vectors for obtaining known-plaintexts are enumerated, and later compared in Table I.

**Exception Vector Table (EVT).** For ARM-based processors, the EVT includes vectors which are reserved for future use. The values of these vectors are regularly filled with zeros when they are not used. By knowing the processor family of the specific microcontroller, the addresses of the used vectors are known as well. Furthermore, the EVT is typically known to be found at a fixed position in the firmware image. We focus on attacking the EVT in the rest of this work since the attack is only architecture dependent and does not require knowledge about the compiler used or old firmware images ($c.f.$ Table I).

**Predictable firmware sections.** Sections within the firmware image may contain static data or code values which may be predictable by an attacker who has carefully reviewed the documentation of the firmware layout. These values may be, $e.g.$, static constants which are stored in the form of tables.

**Compiler conventions.** Compiler conventions such as the calling conventions for the prologue and epilogue in functions or the Embedded-Application Binary Interface (EABI) may be known or easily guessed by the attacker.

**Previous versions of the firmware image.** If a previous version of the firmware image is available in plaintext, an attacker may assume that at least parts of the new firmware image will match parts of the one that is not encrypted.

**Padding values.** If the padding scheme is known or the attacker is able to guess it, she may be able to use this information to perform a malleability attack. A simple example would be the zero-padding scheme. In this scheme all the bytes that are required to be padded are simply filled with zeros.

Table I
ATTACK VECTORS AND REQUIRED KNOWLEDGE.

| Method | Required knowledge | | | |
|---|---|---|---|---|
| | Architecture | Compiler | Prev. firmware | Padding |
| Exception Vector Table | yes | no | no | no |
| Predictable sections | yes | yes | no | no |
| Compiler conventions | no | yes | no | no |
| Known old firmware | no | no | yes | no |
| Padding | no | no | no | yes |

### B. Known Plaintext in the EVT

In ARM-based microcontrollers, the EVT may be used as the known plaintext to perform a malleability attack. For this work, we focus on the EVT structure of microcontrollers (MCUs) of the ARM Cortex-M family. Each vector in the EVT may contain a 32-bit address which is used as a pointer to the Interrupt Service Routine that needs to be called when an exception occurs. Typically, this table is placed at address `0x00000000`, although it may be shifted to the address space from `0x00000080` to `0x3FFFFF80`. The first ten types of exceptions are fixed values defined by ARM. Microcontroller manufacturers are able to include more exception vectors as required for a specific microcontroller. This means that the total length of the EVT is implementation dependent.

Table II shows the typical structure of an EVT. As illustrated, there are some vectors which are reserved for future use. The position of these vectors as well as their content is known to an attacker as these vectors are initialized by the compiler to a fixed value. In the EVT shown in Table II, positions 7, 8, 9, 10 and 13 are reserved for future applications and are known to be filled with zeros.

An attacker who wishes to modify these values to inject her own instructions would have at her disposition 20 bytes, given that the vectors are 4 Bytes wide and there are five of them which are known. However, not all of the 20 bytes may be used to perform an attack. The bytes which constitute the plaintext are split into blocks of regular size. The number of bytes per block depends on the data length of the block cipher used. For AES this would correspond to 16 Bytes per block. As explained in Section II-B1, the manipulation of a ciphertext (*control*) block leads to the modification of the following plaintext (*target*) block. Finding the right control and target blocks is the first task for an attacker. The hexadecimal view shown in Figure 2 depicts the values of the EVT in relation to the blocks they belong to. Assuming that the firmware image is aligned to the blocks of the block cipher, the four bytes corresponding to the exception vector number 7, located at offset `0x1C`, would be found in the second block. In order to modify the content of these values, the content of the bytes at positions `0x0C`, `0x0D`, `0x0E` and `0x0F` which correspond to the previous block would need to be modified, and the values of the ciphertext for the current block must remain untouched. This means that the first block should be used as the control block and the second block as target. By keeping the second block intact, the third block could not be modified and the 12 known bytes in it could therefore not be used. A similar result would occur when trying to use the exception vector 13, located at offset `0x34`. The third block would be used as control block and the forth one as target block. This would modify the ciphertext of the third block and therefore the plaintext of this block would result in a non-predictable value. For this reason, the best strategy is to make use of vectors 8, 9 and 10 for a malleability attack, shown in bold green in Table II. That means, we make use of the ciphertext in the second block as control block to modify the content of the plaintext of the third one. This yields a total of 12 bytes to inject a payload.

Table II
EXCEPTION VECTOR TABLE FOR AN ARM CORTEX-M4 MCU. THE EXCEPTION VECTORS SHOWN IN GREEN CONTAIN THE KNOWN PLAINTEXT VALUES MORE SUITABLE FOR CODE INJECTION.

| Exception | IRQ | Exception Type | Vector Address |
|---|---|---|---|
| - | - | Stack Pointer at Reset | 0x00 |
| 1 | - | Reset | 0x04 |
| 2 | -14 | Non-Maskable Interrupt | 0x08 |
| 3 | -13 | Hard fault | 0x0C |
| 4 | -12 | Memory management fault | 0x10 |
| 5 | -11 | Bus fault | 0x14 |
| 6 | -10 | Usage fault | 0x18 |
| 7 | - | Reserved | 0x1C |
| 8 | - | **Reserved** | 0x20 |
| 9 | - | **Reserved** | 0x24 |
| 10 | - | **Reserved** | 0x28 |
| 11 | -5 | Supervisor Call (SVC) | 0x2C |
| 12 | - | Reserved for Debug | 0x30 |
| 13 | - | Reserved | 0x34 |
| 14 | -2 | PendSV | 0x38 |
| 15 | -1 | SysTick | 0x3C |
| 16 | 0 | Interrupt Request (IRQ) 0 | 0x40 |
| 16 + N | 0 + N | IRQ N | 0x40 + N * 0x04 |

### C. Practical Copy-and-Paste Attack

The space limit of 12 bytes may not be sufficient to inject enough instructions to write a malicious payload. However as explained in Section II-B, ciphertext blocks may be copied and pasted to different locations, what is known as a *Copy-and-Paste* attack. Thanks to this property, the second and third ciphertext blocks of Figure 2 may be repetitively used until a desired area is covered. Since the plaintext of the second block will be effectively randomized by the modification to the ciphertext, the code which is injected should take this into account and jump over the garbled blocks and into the right locations ($c.f.$ Figure 3 in Section IV-C). This reduces the total number of bytes per block which may be used for code, due to the jump instructions which need to be added. However, the addition of jump instructions allows us to arbitrarily increase the space for inserting malicious payload.

### D. Executing Malicious Code

After an attacker has found a way to inject the malicious payload into the encrypted firmware, the next goal is to be able to execute it. To achieve this, the Program Counter (PC) of the CPU must be modified to continue execution starting at the address of the first instruction of the malicious code. For the manipulation of the Program Counter the Exception Vector

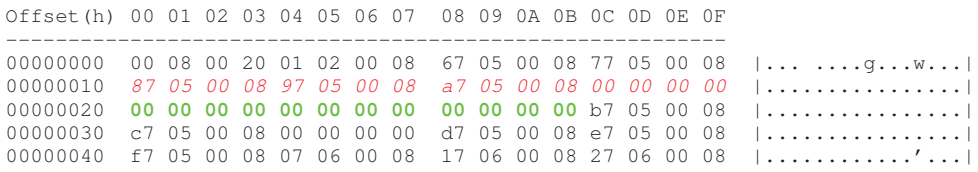



Figure 2. Hexadecimal representation of the Exception Vector Table. Offset 0x20 to 0x2B, shown in bold green, are places which may be used to inject code. The values marked in red italics would result into non-predictable values after decryption as a consequence of the code injection.

Table may be misused once more. By modifying the content of the EVT, the addresses to the interrupt handling routines may be modified to point to the malicious payload instead of the original routines.

## IV. Implementation

This section describes the practical details on how to implement the code injection attack against encrypted firmware updates. The malicious code presented here serves as an example on how an attacker would be able to inject her own code into the encrypted firmware image and use it to extract data from within the microcontroller.

### A. Bootloader with CBC Decryption

The functionality of a bootloader[1] was extended to support the AES block cipher in CBC mode by adding the appropriate decryption routines. It is worth mentioning that since the attack depends only on the mode of operation and the block size, other block cipher algorithms with a block length of 128 bits could have been used instead of AES to yield the same results.

### B. Design of a Malicious Payload

In this section we describe the design of a malicious payload which may be used to get access to sensitive information stored within the microcontroller ($e.g.$ firmware image and secret key material). The malicious payload implements a memory dump through a general purpose input and output (GPIO) pin. The malicious payload is based on a concept developed by Travis Goodspeed [7].

The main idea behind the memory dump program is to make use of two pins of the microcontroller to send the data out as if we were using Serial Peripheral Interface (SPI) bus communication. For this, one pin is used to extract the data while the other pin is used to generate a clock signal. The memory content is read as 4-Byte words starting at address `0x00000000`.

A logic analyzer is used to read out the data. The software for the logic analyzer must be configured to use a protocol decoder for SPI. The pin used to send out the data is set up as the Master Output Slave Input (MOSI) and the pin through which the clock signal is generated is setup as the Serial Clock (SCLK).

[1]Infineon XMC4000 Family Bootloader:http://www.infineon.com/dgdl/Infineon-TOO_Bootloader_v1.1-ATI-v01_01-EN.pdf?fileId=db3a30433e4143bd013e46a7478440e7&ack=t

### C. Implementation of the CBC-Malleability Attack

Through an attack that combines different malleability vulnerabilities of the CBC mode, the payload presented in the previous section can be injected into an encrypted firmware image.

The steps to be performed are shown in Figure 3, and explained below:

1) Copy the control and target blocks from the EVT. As mentioned in Section III-B, the second and third blocks include vectors of the EVT with known-plaintext which can be used to store up to twelve bytes.
2) Inject the malicious payload:
   (a) Paste both blocks to the end of the ciphertext as many times as needed to store the malicious payload. The blocks are appended to avoid causing modifications that would impact the attacked program.
   (b) Modify the control block to inject the malicious payload by flipping the right bits. Care must be taken to correctly link the instructions of the payload. Jump instructions must be added to jump over garbled blocks. And since an instruction cannot be split over two blocks, NOP operations must be inserted to fill in any gaps between instructions in the target block.
3) Modify the IRQ exception vector to point to the entry point of the payload:
   (a) Replace blocks 4 and 5 with the control and target blocks. Applying the Copy-and-Paste attack once more, the fourth and fifth blocks will be replaced with a copy of the second and third blocks.
   (b) Overwrite the IRQ exception vector, found in the EVT at offset `0x40`, to point to the malicious payload starting at address:
   $\texttt{OriginalAddress} + (\texttt{CiphertextLength} + 16)$

   `OriginalAddress` is the starting address to which the firmware will be copied in memory, while `CiphertextLength` denotes the length of the firmware before applying the Copy-and-Paste attack. The offset of 16 bytes is needed because the target block will always be found in the second of a pair of ciphertexts. Note that vectors from offset `0x30` up to `0x3F`, corresponding to reserved vectors and the PendSV and SysTick exception vectors, will be overwritten and their original values will remain unknown.
4) Cause an external interrupt and wait for the execution of the payload when the IRQ is triggered.

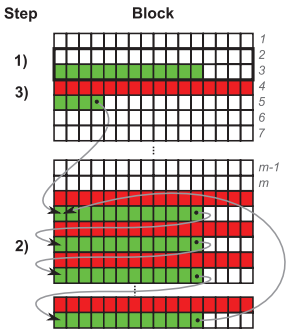



Figure 3. Malleability attack against an encrypted firmware image. Blocks 1 and 2 represent the control and target block respectively. Bytes colored green are known and may be arbitrarily modified. Bytes colored red will be corrupted after decryption. The arrows show the control flow needed to execute the malicious payload.

### D. Proof of Concept

The practicality of the attacks mentioned in the previous section was attested by making use of a proof-of-concept setup. For this, a firmware update was performed, in which the firmware image was encrypted using AES with key length of 256-bits in CBC mode. Only the reserved values in the Exception Vector Table were known beforehand. The tests were performed making use of the Infineon XMC4500 Relax Kit which includes a 32-Bit ARM Cortex M4 core running at 120 MHz. The choice of the microcontroller was arbitrary. Any other ARM-based microcontroller with the same characteristics as the ones described in Section III-B could have been used instead. Code for the microcontroller was programmed making use of IAR Embedded Workbench[2]. The Saleae Logic-Analyzer[3] was used to record the memory dump.

For the tests, the microcontroller was directly connected to a PC through USB. The pins used as outputs for the malicious code, P1.0 and P1.1, were connected to the Saleae logic analyzer. The application code for the microcontroller consisted of a simple program that turned on an LED when one of the buttons on the Relax Kit was pressed and turned it off when it was pressed again. For this, an IRQ was used, which executed an interrupt service routine once the button was pressed. The image was modified as described in the previous chapters and the payload was injected. After this modification, the next step was to perform the firmware update process. The modified bootloader was programmed to decrypt the firmware image on the fly before writing it to the Flash memory. To complete the proof-of-concept, the last step was to execute the malicious code to start the memory dump routine. To accomplish this, the button was pressed to trigger the IRQ which finally led to the execution of the malicious code. The logic analyzer was used afterwards to obtain a copy of the data stored within the device.

[2]IAR Embedded Workbench: https://www.iar.com/iar-embedded-workbench/

[3]Saleae Logic Analyzer: https://www.saleae.com/logic/

## V. Conclusions

Drawbacks of using confidentiality-only encryption algorithms have been known for many years. However, several applications still make use of confidentiality-only modes under the false assumption, that only the secrecy of the content needs to be protected. By failing to provide strong integrity, the weaknesses of confidentiality-only block cipher modes, such as CBC, may be misused to ultimately compromise the secrecy of the encrypted content. In this work we showed how malleability attacks can be implemented when having knowledge of the plaintext structure. Attacks such as the one described in this paper may be thwarted through the use of authenticated encryption modes, which protect not only the privacy, but also the integrity and authenticity of the encrypted content.

## Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments and suggestions. This work was partly funded by the German Federal Ministry of Education and Research (BMBF), project SIBASE, grant number 01IS13020A.

## References

[1] Atmel. AVR231: AES Bootloader. http://www.atmel.com/images/doc2589.pdf, 2012. [Online; accessed 22-June-2015].

[2] Atmel. AT02333: Safe and Secure Bootloader Implementation for SAM3/4. http://www.atmel.com/Images/Atmel-42141-SAM-AT02333-Safe-and-Secure-Bootloader-Implementation-for-SAM3-4_Application-Note.pdf, 2013. [Online; accessed 22-June-2015].

[3] Daniel Bernstein. CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness. http://competitions.cr.yp.to/caesar.html, 2014. [Online; accessed 23-June-2015].

[4] Danny Dolev, Cynthia Dwork, and Moni Naor. Non-malleable Cryptography. In *Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 542–552, New York, NY, USA, 1991. ACM.

[5] Clemens Fruhwirth. *New Methods in Hard Disk Encryption*. Institute for Computer Languages, Theory and Logic Group, Vienna University of Technology, 2005.

[6] Virgil D. Gligor, Pompiliu Donescu, and Jonathan Katz. On Message Integrity in Symmetric Encryption. In *1st NIST Workshop on AES Modes of Operation*, 2000.

[7] Travis Goodspeed. Some Shellcode Tips for MSP430 and Related MCUs. In *Children's Bible Coloring Book of PoC || GTFO*, volume 2, pages 10–12. 30th CCC Congress in Hamburg, 2013.

[8] Derek Lau. Freescale Semiconductor Application Note: Secure Bootloader Implementation. http://cache.freescale.com/files/microcontrollers/doc/app_note/AN4605.pdf, 2012. [Online; accessed 22-June-2015].

[9] National Institute of Standards and Technology. FIPS PUB 81: DES Modes of Operation, 1980.

[10] National Institute of Standards and Technology. SP 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques, 2001.

[11] Kenneth G. Paterson and Arnold K. L. Yau. Padding Oracle Attacks on the ISO CBC Mode Encryption Standard. In *Topics in Cryptology–CT-RSA 2004*, pages 305–323. Springer, 2004.

[12] Kenneth G. Paterson and Arnold K. L. Yau. Cryptography in Theory and Practice: The Case of Encryption in IPsec. In *Advances in Cryptology, EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 12–29. Springer Berlin Heidelberg, 2006.

[13] Serge Vaudenay. Security Flaws Induced by CBC Padding Applications to SSL, IPSEC, WTLS... In *Advances in Cryptology EUROCRYPT 2002*, pages 534–545. Springer, 2002.

[14] Arnold K. L. Yau, Kenneth G. Paterson, and Chris J. Mitchell. Padding Oracle Attacks on CBC-mode Encryption with Secret and Random IVs. In *Fast Software Encryption*, pages 299–319. Springer, 2005.