Incremental Code Updates Exploitation as a Basis for Return Oriented Programming Attacks on Resource-Constrained Devices

AbdElaziz Saad AbdElaziz AbdElaal*¶, Kai Lehniger*, Peter Langendörfer*¶
*IHP - Leibniz-Institut für innovative Mikroelektronik
Frankfurt (Oder), Germany
Email: {saad, lehniger, langendoerfer}@ihp-microelectronics.com
¶Brandenburgische Technische Universität Cottbus-Senftenberg
Cottbus, Germany
Email: abdelaziz.saad, peter.langendoerfer@b-tu.de

2021 5th Cyber Security in Networking Conference (CSNet) | 978-1-6654-0722-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/CSNET52717.2021.9614275

***Abstract*—Code-reuse attacks pose a threat to embedded devices since they are able to defeat common security defenses such as non-executable stacks. To succeed in his code-reuse attack, the attacker has to gain knowledge of some or all of the instructions of the target firmware/software. In case of a bare-metal firmware that is protected from being dumped out of a device, it is hard to know the running instructions of the target firmware. This consequently makes code-reuse attacks more difficult to achieve. This paper shows how an attacker can gain knowledge of some of these instructions by sniffing the unencrypted incremental updates. These updates exist to reduce the radio reception power for resource-constrained devices. Based on the literature, these updates are checked against authentication and integrity, but they are sometimes sent unencrypted. Therefore, it will be demonstrated how a Return-Oriented Programming (ROP) attack can be accomplished using only the passively sniffed incremental updates. The generated updates of the R3diff and Delta Generator (DG) differencing algorithms will be under assessment. The evaluation reveals that both of them can be exploited by the attacker. It also shows that the DG generated updates leak more information than the R3diff generated updates. To defend against this attack, different countermeasures that consider different power consumption scenarios are proposed, but yet to be evaluated.**

***Index Terms*—Return-Oriented Programming, IoT, Security, Incremental Code Update**

I. Introduction

After the deployment of a network of IoT devices, a bug or a security vulnerability can be found. Also, a feature could be needed to be added or removed from such networks. Therefore, it is very important to consider a secure, reliable and convenient update technique. The devices in these networks are deployed scattered over the place such as Internet of Things (IoT) devices in smart homes, smart cities, or in a Wireless Sensor Network (WSN). Furthermore, a WSN can be deployed in harsh/scarce environment which makes collecting the devices back to update them using cables a big challenge. Therefore, a convenient way to deliver the update is disseminating it Over the Air (OTA) using one of the Over The Air Programming (OTAP) frameworks such as R3 [1] or DG [2]. The devices in such IoT networks can be classified as high-end and low-end devices, with some challenges in regard to the low-end devices for OTA updates. Low-end devices are resource-constrained devices which usually come in a form of low-power and low-cost Micro-Controller Unit (MCU) or System-on-Chip (SoC). Also, these devices are usually battery-powered where the power consumption efficiency is very crucial. The OTA update can consume substantial radio reception power from the device while receiving it. For this reason, it is not wise to send an entire new firmware as an update to the device. Since the early 2000s, there are many proposed OTAPs frameworks which make use of the so-called differencing algorithms to generate a differential update that only contains the changes between the two firmware versions. Consequently, the consumed power during receiving a differential update will be much less than the consumed power during receiving an entire new firmware. Since IoT became a non-negligible part of our life, its security became a crucial concern quickly. Therefore, the authenticity and integrity of an update are intensively discussed in the literature and even standardized in the Software Update of IoT Devices (SUIT) IETF standard. Nevertheless, the confidentiality of the update is left optional [3], [4].

In this work, the risk of sending the differential updates unencrypted will be emphasized. The evaluation demonstrates that the generated differential updates of R3diff and DG algorithms (for which their authors stated that they generate the smallest differential update sizes) can leak enough ROP gadgets to compromise the updated device. This attack doesn't imply a direct weakness in the differencing algorithms, because their main concern is to generate a differential update as small as possible not to secure it during transmission. The weakness comes from the dissemination protocols which are used in a particular OTAP that enforce only authentication and integrity while leaving the confidentiality optional. In order to prevent the presented attack, different scenarios for power-efficient countermeasures are proposed.

The rest of this paper is categorized as follows: In section II, the concepts of differencing algorithms and the Return-Oriented Programming Attack (ROP) are discussed. Section III demonstrates the conceptual steps of the attack against the R3diff and DG generated updates. In section IV, the exploitability of the updates from the two algorithms is compared. Different countermeasures that consider different power consumption scenarios are proposed in section V. Section VI lists the previous work that managed to accomplish ROP attack with small or zero knowledge of the instructions of the target firmware/software. Finally, section VII concludes the paper.

II. Background

A. Differencing Algorithm

The differencing algorithm takes the old and new firmware images as input and correlates them to produce a differential delta update [5]. To achieve that, the differencing algorithm identifies the matching and non-matching parts of the old and the new firmware. Then, it encodes the matching segments with COPY commands and the non-matching segments with ADD or INSERT commands in a so-called delta script. The COPY commands are used to tell the device to reuse already existing code snippets by coping them from the currently running firmware to same or new positions in the new firmware that is being constructed. Every ADD command consists of a header and payload. The header contains the address and the length of the payload. They are used to add the payload, that represents non-matching segments, to the new firmware.

A differencing algorithm can be either in-place or out-of-place [6] which sometimes are called single and dual-bank respectively [7]. This depends on the way its generated updates are executed on the resource-constraint devices. During the in-place update, the new firmware image is constructed right in the same memory bank where the previous old firmware exists. Consequently, some parts of the old firmware are overwritten during the update process on the device, making the old firmware unavailable during the update process. Moreover, in case of any update failure, the device can't be reverted back to the old firmware. On the other hand, the out-of-place update processing constructs the new firmware in a different memory bank without overwriting the old firmware. Therefore, in case of any update failure, the device can be reverted back to the old firmware. Table I shows several differencing algorithms which are sorted based on the year they have been proposed in. It also shows the execution type, and the runtime complexity of each algorithm. In this paper, the generated updates of the two algorithms R3diff that is a part of R3 OTAP and DG-Optimized, that are highlighted in Table I, will be under assessment. While being the most novel approach, DASA-Improved is not considered for evaluation because it is not yet implemented or evaluated [8].

TABLE I: SEVERAL DIFFERENCING ALGORITHMS, THEIR TYPES, AND THEIR RUNTIME COMPLEXITY

Algorithm	First appeared	Execution	Time Complexity
Rsync [9]	1999	Out-of-Place	$O(n^2)$
FBC [10]	2004	Out-of-Place	$O(n)$
RMTD [11]	2009	Out-of-Place	$O(n^3)$
DASA [12]	2012	Out-of-Place	$O(nlog(n))$
R3diff [1]	**2013**	***Out-of-Place***	$\boldsymbol{O(n^3)}$
DG [2]	2016	In-Place	$O(n^2)$
DG-Optimized [13]	**2019**	***In-Place***	$\boldsymbol{O(n^2)}$
DASA-Improved [8]	2020	Out-of-Place	$O(nlog(n))$

B. ROP Attack

The Return-Oriented Programming (ROP) term was introduced first by Hovav Shacham on his paper "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" in 2007 [14]. Later on, ROP attacks have proven to be applicable to wide range of architectures. ROP is classified as a code-reuse attack that is triggered using a memory corruption attack vulnerability.

The attacker searches through the binary for sequences of instructions that end with a return (RET) instruction; every found sequence is called a ROP gadget. Gadgets can be chained together to divert the control flow of the running application and constructing a turing-complete exploit. The idea is that, in almost every architecture, the RET instruction pops and jumps to the so-called return-address which is saved in the stack when a function is called. Using this fact, the attacker overwrites a return-address with the address of the first chosen ROP gadget. Thus, the processor will jump to execute that ROP gadget. When the processor reaches the end of that ROP gadget, it will find a RET instruction which again will pop and jump to the address in the stack that is pointed by the stack pointer. This address will also be controlled by the attacker to be the address of the second gadget, and so on.

III. Incremental Code Updates as a Basis for Return-Oriented Programming

A possible way to know some of the instructions of a target firmware to collect ROP gadgets is to passively sniff the firmware OTA updates while being disseminated to the devices in the network. The ADD commands are very valuable to the attacker as they contain the raw bytes that will be used to patch the current running firmware. The main challenge is that the number of the collected ROP gadgets from the updates is usually less than the number of gadgets that can be collected while having access to the complete firmware image. For example, based on our analysis, the delta script with a size 26KB that was generated by the R3diff differencing algorithm between images with sizes 177KB and 180KB respectively leaked only 17 ROP gadgets compared to 300 ROP gadgets in the scenario of having full access to the new firmware image. However, in this paper, there will be a detailed explanation of how to use only 2 of the 17 ROP gadgets to compromise a MCU device.

A. Attack Model

The attacker will be in the middle between the base station/firmware server and the devices in the network to sniff the traffic of the firmware update. The target device is assumed to have a buffer-overflow vulnerability in the stack that could be used to overwrite a return-address or a function pointer since all the defenses are either not applicable in the resource-constraint world, incur large runtime overhead or can be overcome. According to the Common Weakness Enumeration (CWE), memory buffer overflow comes at the 1st place of the list of the 2019 critical weaknesses [15] that led to severe vulnerabilities and the 2nd in the 2020 report [16]. Thus, it is very likely to assume the existence of that vulnerability and confirming it using techniques such as Fuzzing. The running firmware is assumed to be protected against firmware dumping attacks that enable the attacker to readout the firmware from the device. Also, the target device is assumed to have a Memory Protection Unit (MPU) that enforces the stack to be non-executable which prevents the attacker from executing injected code from the stack, but it doesn't have special hardware such as Trusted Execution Environment (TEE). Also, we assume that the updates are sent unencrypted.

B. Exploitation Steps

As it is shown in Figure 1, an attacker goes through the following steps to collect the ROP gadgets from the sniffed OTA updates.

1) The attacker scans the radio range to know at which frequency the wireless communication occurs using an SDR device such as HackRF and BladeRF devices and signal processing software such as GNURadio. After determining the frequency, the attacker starts sniffing the traffic, converting it to raw bytes and storing them in a hex file format. These raw bytes, if they were received without packet losses, should represent a complete delta file which consists mainly of ADD, COPY commands, and checksum of the generated updates.
2) While assuming that the attacker doesn't know the used differencing algorithm that has been used to generate these updates, he detects it by a plausibility check against different delta encodings of the widely used algorithms and see which encoding of which algorithm match the sniffed delta update binary. It is very unlikely that two different delta formats can produce the same delta file byte string with different meanings.
3) The attacker identifies the locations of the COPY and ADD commands based on the detected differencing algorithm in step 2. In this Figure, the gray color represents the identified COPY commands and the red and green colors represent the identified header and payload of the ADD commands respectively.
4) The attacker constructs a partial image that is similar to the code image that will be running on the device after update with some unknown gaps in it. The COPY commands (marked with gray color) represents the **unknown** parts of the firmware image as they don't contain a payload. The ADD commands (marked with green) represent the **known** and valuable information to the attacker. The headers of both COPY and ADD commands help in identifying the locations of the known and unknown parts in the constructed partial image and aligning the payloads of the ADD commands at their right locations. This construction phase is very important before starting the reverse engineering of the partial constructed image by disassembling it to look for ROP gadgets.
5) In the last step, the reconstructed parts of the images are searched for gadgets. There exists a number of tools such as mona [17], Ropper and ROPgadget [18], [19] that can automate this process. However, till the time of writing this paper, there were no off-the shell tools for the MSP430 MCU architecture, that was used during the evaluation.

IV. Evaluation

The evaluation of the attack focuses on the analysis of the R3diff and DG generated updates for firmwares that are running on a MSP430X MCU and the amount of information leakage that could be used to collect ROP gadgets to construct a ROP attack. In the evaluation, two evaluation examples have been considered. In the first example, it is assumed a firmware that has been updated to a new version and later on has been reverted back. This firmware is a proof-of-concept cross platform LED blinking application that has updated the blinking frequency and some bugs have been fixed. In the second example, an environment measurement firmware has been tracked while it is being updated 3 times.

A. First Evaluation Example

The results of the first example are depicted in the Tables II and III. In Table II, firmware with size **180KB** has been upgraded to a new firmware with size **177KB** that contains **380** ROP gadgets. In Table III, it is assumed that the developers reverted back to the old firmware. Thus the old firmware image size is now **177KB** and the new firmware image size is **180KB**.

As it is clear from the **two Tables** II and III, the DG algorithm produces a delta script which is bigger in size than the delta script generated by the R3diff algorithm. Also, the average payload lengths of the ADD instructions in the case of the DG algorithm is much bigger than the one in case of the R3diff algorithm. This consequently results in a higher possibility of finding ROP gadgets in DG updates compared to R3diff updates. Thus DG is more vulnerable to ROP attack than R3diff. It's worth to mention that, although the number of different bytes between the firmwares is 44KB, the R3diff generated updates leaked 17 gadgets in the first update as it is shown in the Table II and leaked 18 gadgets in the the second update as it is shown in the Table III. However, the attack is still depending on the type of collected gadgets not only the numbers of the collected gadgets. The two gadgets in the code

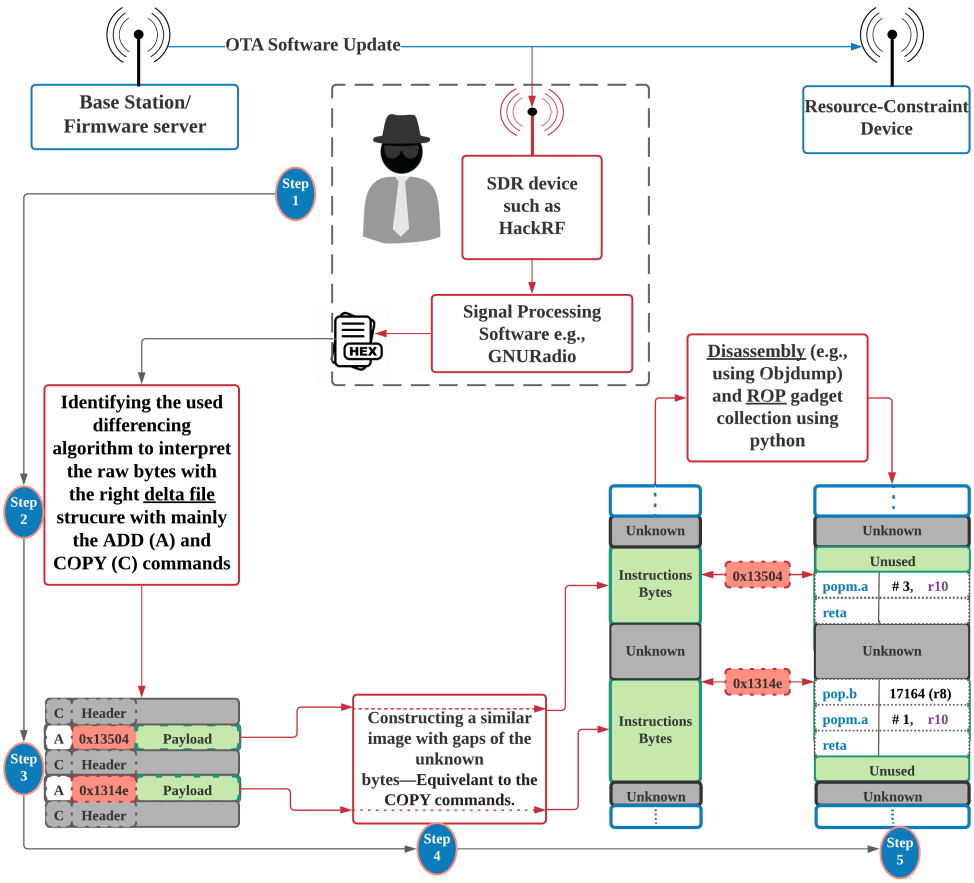

Fig. 1: Steps to extract gadgets out of an incremental update.

Listings 1 and 2 were found. Using these two gadgets, it was still possible to attack the device with the ”write anything, anywhere” power.

```
0x13504:    popm.a   #3, r10
0x13506:    reta
```

Listing 1: A ROP gadget that assignes an arbitrarly value from the stack to the r8, r9, and r10 registers.

```
0x1314e:    pop.b    17164(r8)
0x13152:    popm.a   #1, r10
0x13154:    reta
```

Listing 2: A ROP gadget that pops one byte from the top of the stack into the memory location that is computed by 17164(r8) = 17164+r8.

1) The First ROP Gadget Outcomes: The first line in the ROP gadget in the code Listing 1 pops three consequent values from the top of the stack to the registers r8, r9 and r10 respectively and increment the stack pointer by 3 words. What is important is the value that will be assigned to r8 since this register will be used by the first instruction in the second ROP gadget in the code Listing 2.

2) The Second ROP Gadget Outcomes: The first line in the ROP gadget in the code Listing 2 is very dangerous as it pops whatever exist on top of the stack (controlled by the attacker) and stores it into the given argument location which is resolved as follows 17164(r8) = 17164 + r8. Since the value of the r8 register can be controlled using the first ROP gadget in the code Listing 1, the memory location 17164 + r8 can be controlled by the attacker. Thus, the attacker arbitrarily writes to any memory locations.

B. Second Evaluation Example

In the second example, we traced the evolution of a firmware that is used as an environment measurement application. The firmware was updated multiple times to add more features and to correct bugs that had been discovered. The results are shown in Table IV.

In this example, the attacker could be looking at the findings of each update separately or he could be **correlating** every finding with the previous ones to collect more ROP gadgets. In this case, the attacker correlates his current findings from the newly constructed image with the previous findings from the perviously constructed image. If the attacker just started his sniffing process and collected small number of gadgets that could be used to construct a successful ROP attack, he

TABLE II: THE ANALYSIS of UPDATING a FIRMWARE WITH SIZE 180K to a FIRMWARE WITH 177K, THE TABLE CONCLUDES THAT the NUMBER of COLLECTED GADGETS in CASE of DG IS MUCH HIGHER THAN the ONES in CASE of R3DIFF

Firmware version	Firmware Size	**No. of ROP gadgets**
Old	177KB	-
New	180KB	**380**
NO. of different bytes	44KB	
Algorithm	**DG**	**R3diff**
NO. of ADDs	1432	1935
NO. of COPYs	392	1984
% of ADDs	78%	49%
Update Size	44KB	26KB
Total ADD payloads lengths	37KB	10KB
Average of ADDs payloads length	27	6
NO. of ROP gadgets in the update	**297**	**17**
% of update ROP gadgets	%(297/380) = **87%**	%(17/380) = **4%**

TABLE III: THE ANALYSIS of UPDATING a FIRMWARE WITH SIZE 177K TO a FIRMWARE WITH 180K, THE TABLE ALSO CONCLUDES THAT THE NUMBER of COLLECTED GADGETS in CASE of DG IS MUCH HIGHER THAN THE ONES in CASE of R3DIFF

Firmware version	Firmware Size	**No. of ROP gadgets**
Old	180KB	-
New	177KB	**390**
NO. of different bytes	44KB	
Algorithm	**DG**	**R3diff**
NO. of ADDs	1418	1975
NO. of COPYs	406	2016
% of ADDs	77%	49%
Delta Script Size	44KB	27KB
Total ADD payloads lengths	36KB	11KB
Average of ADDs payloads length	27	6
NO. of ROP gadgets in the update	**321**	**18**
% of update ROP gadgets	%(321/390) = **82%**	%(18/390) = **4%**

waits until he sniffs another update and correlates it with the previous update.

In the evaluation that is presented in the Table IV, the tracking of the the multiple versions of the firmware binary are checked **separately** (no correlation with the previous updates because every update already leaked enough useful ROP gadgets) so that the update between each subsequent versions was calculated and tested against ROP attack. The test showed that all the updates whether generated by R3diff or DG are vulnerable to a ROP attack so correlating the updates was not necessary.

From Table IV, it is still clear that DG is more vulnerable than R3diff as the number of collected ROP gadgets in case of DG is higher by an order of magnitude than the one in the case of R3diff. The **useful gadgets** column is indicating gadgets that either have a pop instruction or any instruction that could be writing to the memory. The useful gadgets don't work alone, the other collected gadgets can also be helpful. The "useful" word here indicates that those gadget worths to be investigated by the attacker before the other ones.

Two main reasons explain why DG generated updates leak more ROP gadgets than R3diff. The first one is that the DG algorithm assumes small changes between different firmware images. The second one is due to the difference in the update execution mechanism between R3diff and DG. Since the new software image in the DG is being constructed in the same memory bank where the previous software image exists, there are many cases where a previous COPY command could overwrite some potential bytes that could be used in other COPY commands. Consequently, those parts of the firmware that could have been copied using a COPY instruction, due to the in-place construction nature of DG, they will be reconstructed using an ADD instruction. Thus, giving the attacker better opportunity to collect more ROP gadgets.

To conclude, both R3diff and DG generated updates can leak enough ROP gadgets to attack the device that is receiving the update. But, DG is much more vulnerable than R3diff.

V. Possible Countermeasures

It was proven in this work that the incremental code updates that are sent unencrypted can be used as a serious attack vector to maliciously control a resource-constrained device. With that being said, it is clear that the direct countermeasure against the demonstrated attack is to encrypt the Full OTA incremental updates. The full encryption of the incremental update can dissipate substantial power. Therefore, multiple encryption scenarios will be proposed which consider devices with less power resources, and whether an already existing encryption scheme (with its already shared keys) will be used **or** an Over The Air Programming (OTAP)'s built-in encryption will be used. The following scenarios can be used if full encryption is not feasible.

A. Unaffordable Full Update Encryption

If the encryption is expensive or the update frequency is high, we propose a more efficient technique than encrypting the full incremental update. The general idea is encrypting some parts of it that makes it difficult for the attacker to collect ROP gadgets.

As it was mentioned earlier, delta scripts (incremental updates) mainly consist of COPY command headers, ADD command headers, ADD payloads, and checksum. The more valuable parts to the attacker are the **payloads** of the **ADD commands**, as they can be reverse-engineered and ROP gadgets are collected from them. Consequently, encrypting the payloads of the ADD commands will prevent the attacker from collecting gadgets from them. However, encrypting every ADD payload separately could increase the overall size of the delta script and would also interfere directly with the algorithm itself, so it is better to **append** all the ADD payload together and **separate** them from the COPY and ADD commands headers as it is shown in Figure 2. The delta script encoding and decoding need also to be modified to adopt this separation

TABLE IV: TRACKING an ENVIRONMENT MEASUREMENT FIRMWARE EVOLUTION AND CALCULATING THE NUMBER of ROP GADGETS FROM THE COMPLETE FIRMWARE IMAGES, R3DIFF AND DG UPDATES

MSP430	Firmware				**R3diff**			**DG**		
Firmware	Size	Diff	**Gadgets**	**Useful**	Delta	**Gadg.**	**Useful**	Delta	**Gadg.**	**Useful**
Base	99KB	-	-	-	-	-	-	-	-	-
1	122KB	35KB	**368**	**203**	20KB	**61**	**17**	28KB	**280**	**140**
2	186KB	75KB	**405**	**254**	35KB	**66**	**31**	48KB	**357**	**215**
3	181KB	54KB	**485**	**332**	32KB	**110**	**54**	32KB	**313**	**202**

between the headers and the encrypted ADD payloads. This way it is possible to transmit them separately and apply encryption only to the packets containing the payload, see Figure 3. On the receiving device side, the update execution mechanism should be modified to decrypt and use the ADD payloads on the fly when it finds an ADD header. This countermeasure is strong and more power efficient than encrypting the full OTA update. However, the attacker can still sneak some information about what changed in the new version of the firmware by looking at the ADD commands headers which are sent unencrypted. This information could help him in other attack styles.

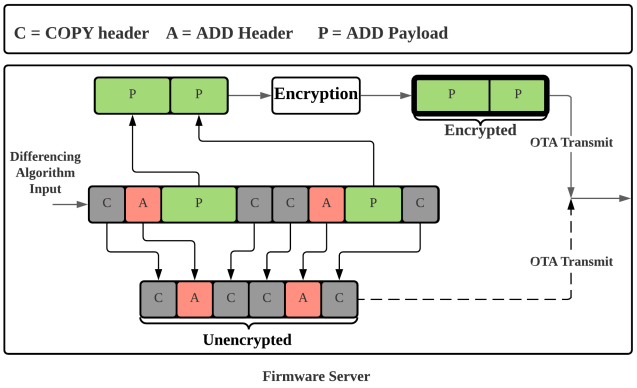

Fig. 2: Changes in the delta script transmission process.

B. OTAP Built-in Encryption

In this scenario, based on the assumption that the attacker doesn't have access to the currently deployed firmware on the resource-constraint device, we don't have to tackle the challenge of exchanging encryption keys. This is because, parts of the old firmware that is currently deployed in the device will be used as a pre-shared key. The reason we are introducing this countermeasure is to encrypt the update **once** using the deployed image as a pre-shared key and send it to all the devices in need of the update. This technique is generally easy to be implemented. It only differs slightly depending on the update strategy.

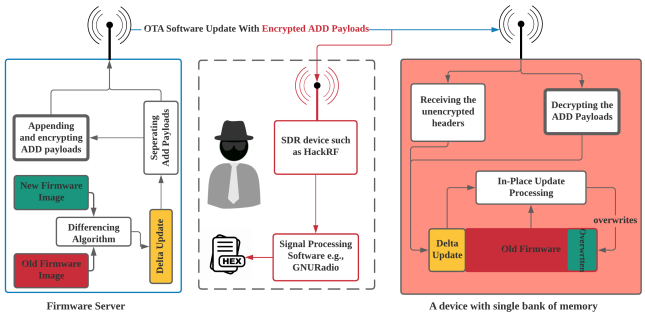

Fig. 3: Changes in the delta script transmission and reception processes in the firmware server side and the resource-constrained device side.

1) Out-Of-Place Algorithm: In the case of the out-of-place algorithm, the built-in encryption will be very straightforward to implement. The reason is that old deployed image, which we consider it as the pre-shared key, is not changed during the update execution. As it was previously mentioned, the new image is being constructed in a second memory bank while leaving the old image unmodified. An iterator will begin at the at the beginning of the old image (the key) and increment by one while XORing the byte value pointed to by this iterator with the values from the incremental updates to get encrypted. If it happened that the iterator reached the end of the old image, its value will be rewound to start again at the beginning of the old image.

2) In-Place Algorithm: In case of an in-place algorithm, the implementation will differ slightly. The reason is that during the in-place update the old image (the key) values could be changed by a previous ADD or COPY command since the update execution reconstructs the new image right in the same location of the old image. A workaround is to not encrypt with an old image directly, but to take the changes of previous commands into account. By doing so, we make sure that the key values are synchronized between the firmware server and the device that is receiving the update.

Since this solution is using the old image as a pre-shared key for encryption and decryption, a question arises about

the randomness of this key with each update. If every update introduces small changes to the deployed old image, this will directly imply that most of the key bytes will stay the same across multiple updates. Consequently, the key randomization will not be strong. Further, the randomization is not the only problem, the attacker can also guess some parts of the deployed image (key) based on his knowledge of the architecture of the device and the application that this device is used for.

3) Embedding an Encryption Key in the Base Image: Another possible way that works also independently of the existing data encryption is to embed a secret pre-shared key in the base image (version zero). This key will be known only to the firmware server and the devices that uses or used this base image. Every time an update needs to be distributed, the full or parts of update will be encrypted with that key, and on the device side the decryption will occur using the pre-embedded key in that old image. It is also possible that an update can change the pre-embedded key to another fresh key.

VI. Related Work

Since this paper introduced a novel way to use the incremental updates as a basis for a code-reuse attack against a closed-binary proprietary firmware, this section will discuss the related work that could achieve a similar attacks without having a copy of the running closed-binary firmware.

A. Half-Blind Attacks

In 2009, in the paper **"Half-Blind Attacks: Mask ROM Bootloaders are Dangerous"**, Travis Goodspeed et al. [20] managed to prove that they can dump the running secret firmware out of the device using its Boot Strap Loader (BSL) and by only guessing the location of one ROP gadget. This ROP gadget is used in privilege escalation to be able to execute the BSL commands that send the firmware back to the attacker. It is half-blind as they are guessing the location of the privilege escalation ROP gadget by brute-forcing its entry address. And the rest of the attack is carried out by jumping to the known entry address of the bootloader which is already publicly known to the attacker. By dumping the firmware, the intellectual property of the firmware will be broken. Further, the attacker can use the dumped firmware to build his code-reuse attack with much more information under his access, which consequently lead to more serious attacks. The limitation of this attack is that it depends on the level of security the bootloader has. If the bootloader address is unknown to the attacker or completely overwritten by a new user bootloader, then the attack will be more difficult to succeed. Also, this attack is guessing one ROP gadget by brute forcing all the entry even addresses of the code address space. The percentage of this brute-force success is 1%, a device lock is 4%, and wrong guessing but the device will be still functioning is 95% [20].

This attack is similar to our attack as it doesn't assume any access to the closed-binary firmware that is running on the device. It, instead, assumes that the device is running a default BSL. The difference is that our attack doesn't guess the locations of the ROP gadgets. It, instead, uses the incremental updates to collect the ROP gadgets. If the right gadgets weren't found, the attacker stays in the network to sniff more updates until he can find his required gadgets. Our attack can potentially achieve the same behavior of the Half-Blind attack without the time that is spent on the guessing of the ROP gadgets. Also, our attack is passively sniffing the updates. It only interacts with the device when it has a complete payload of the ROP gadgets chain. Consequently, this makes our attack undetectable, by the network operators, more than the half-blind attack which intensively interacts with the device until it finds the right gadget. A correlation can be made between our demonstrated attack and the half-blind attack to possibly remove the need of the privilege escalation gadget. This will be possible if the attacker can find one or two gadgets (that have the same privilege escalation behavior) in the code updates.

B. Full-Blind ROP Attacks

In 2014, in the paper **"Hacking Blind"**, Andrea Bittau et al [21]. showed that they can fully blindly exploit a remote closed-source and closed-binary software. It is an extended version of the half-blind attack. The attack is assuming that the remote target software restarts after crash. This attack again brute-forces the locations of specific ROP gadgets entry addresses. The crash is being used as a signal for the attacker to decide whether the guessed ROP gadget entry address is correct or not. In this attack, they managed to blindly exploit closed-binary software within 20 minutes. The limitation of this attack is that it mainly uses the crash of the software as a signal for the success of the guessed ROP gadget. If the target firmware can catch the crash signal and wait arbitrarily time before crashing, this will completely stop the attack [21]. In the previous explained half-blind attack against MCUs, this is not a limitation as the attacker needs to guess the address of only one entry point to a privilege escalation ROP gadget. This attack again is intensively interacting with the device which can be noticed and detected in contrast to our attack which works passively until it collects the required ROP payload.

C. Other Incremental Update Attacks

If the update authenticity, integrity, and freshness are not strongly checked, this opens a wide variety of firmware modification attacks. Well-known examples of these attacks have been discussed in the "Firmware Update Attacks and Security for IoT Devices" survey by Meriem Bettayeb et al. [22] in 2019. These attacks are considered to be active attacks and they are out of the scope of this thesis. Regarding the incremental update passive attacks, to the best of our knowledge, there is no passive attacks similar to the one that has been discussed in the paper.

VII. Conclusion

In this paper, we showed that incremental code updates can be used as a serious attack vector in exploiting low-end resource constraint devices. The generated updates of two

differencing algorithm, namely R3diff and DG, were under assessment. The analysis showed that the DG generated updates leak more information to the attacker than the R3diff generated updates. However, both differencing algorithms generate updates that leak enough information to build a working ROP attack. We also showed that an attacker can stay longer in the network and correlates his previous and new findings together to be able to extract more ROP gadgets. In addition to that, we proposed multiple countermeasures, all of them based on the encryption of the updates. Some of these countermeasures expects an existing underlying data encryption and pre-shared keys. And others use a proposed an OTAP built-in encryption.

REFERENCES

[1] W. Dong, B. Mo, C. Huang, Y. Liu, and C. Chen, "R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems," in *2013 Proceedings IEEE INFOCOM*, 2013, pp. 315–319.

[2] O. Kachman and M. Balaz, "Optimized differencing algorithm for firmware updates of low-power devices," in *2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2016, pp. 1–4.

[3] K. Zandberg, K. Schleiser, F. Acosta, H. Tschofenig, and E. Baccelli, "Secure firmware updates for constrained iot devices using open standards: A reality check," *IEEE Access*, vol. 7, pp. 71 907–71 920, 2019.

[4] "A firmware update architecture for internet of things," https://tools.ietf.org/html/draft-ietf-suit-architecture-16.

[5] K. Arakadakis, P. Charalampidis, A. Makrogiannakis, and A. Fragkiadakis, "Firmware over-the-air programming techniques for iot networks – a survey," 2020.

[6] K. Lehniger and S. Weidling, "The impact of diverse execution strategies on incremental code updates for wireless sensor networks," in *Proceedings of the 8th International Conference on Sensor Networks, SENSORNETS 2019, Prague, Czech Republic, February 26-27, 2019*, C. Benavente-Peces, A. Ahrens, and O. Camp, Eds. SciTePress, 2019, pp. 30–39. [Online]. Available: https://doi.org/10.5220/0007383400300039

[7] O. Kachman and M. Balaz, "Efficient patch module for single-bank or dual-bank firmware updates for embedded devices," in *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS)*, 2020, pp. 1–6.

[8] K. Arakadakis and A. Fragkiadakis, *Incremental Firmware Update Using an Efficient Differencing Algorithm: Poster Abstract*. New York, NY, USA: Association for Computing Machinery, 2020, p. 691–692. [Online]. Available: https://doi.org/10.1145/3384419.3430471

[9] A. Tridgell, "Efficient algorithms for sorting and synchronization," 1999.

[10] Jaein Jeong and D. Culler, "Incremental network programming for wireless sensors," in *2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004.*, 2004, pp. 25–33.

[11] J. Hu, C. J. Xue, Y. He, and E. H. . Sha, "Reprogramming with minimal transferred data on wireless sensor network," in *2009 IEEE 6th International Conference on Mobile Adhoc and Sensor Systems*, 2009, pp. 160–167.

[12] B. Mo, W. Dong, C. Chen, J. Bu, and Q. Wang, "An efficient differencing algorithm based on suffix array for reprogramming wireless sensor networks," in *2012 IEEE International Conference on Communications (ICC)*, 2012, pp. 773–777.

[13] O. Kachman, M. Baláz, and P. Malík, "Universal framework for remote firmware updates of low-power devices," *Comput. Commun.*, vol. 139, pp. 91–102, 2019.

[14] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," ser. CCS '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 552–561. [Online]. Available: https://doi.org/10.1145/1315245.1315313

[15] MITRE, "2019 cwe top 25 most dangerous software errors."

[16] ——, "2020 cwe top 25 most dangerous software weaknesses."

[17] R. C. Inventory, "mona," https://github.com/corelan/mona.

[18] S. Schirra, "Ropper," https://github.com/sashs/Ropper.

[19] J. Salwan, "Ropper," https://github.com/JonathanSalwan/ROPgadget.

[20] T. Goodspeed and A. Francillon, "Half-blind attacks: mask rom bootloaders are dangerous," in *Proceedings of the 3rd USENIX conference on Offensive technologies. USENIX Association*, 2009, pp. 6–6.

[21] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *2014 IEEE Symposium on Security and Privacy*, 2014, pp. 227–242.

[22] M. Bettayeb, Q. Nasir, and M. A. Talib, "Firmware update attacks and security for iot devices: Survey," in *Proceedings of the ArabWIC 6th Annual International Conference Research Track*, ser. ArabWIC 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3333165.3333169