Annals of Telecommunications (2023) 78:209–220
https://doi.org/10.1007/s12243-022-00917-8

Finding gadgets in incremental code updates for return-oriented programming attacks on resource-constrained devices

Kai Lehniger[1] · Abdelaziz Saad[1] · Peter Langendörfer[1,2]

Received: 9 January 2022 / Accepted: 16 June 2022 / Published online: 15 July 2022

Abstract
Code-reuse attacks pose a threat to embedded devices since they are able to defeat common security defences 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 presents a novel approach how an attacker can gain knowledge of some of these instructions by sniffing unencrypted incremental updates. These updates exist to reduce the radio reception power for resource-constrained devices. It will be demonstrated how a return-oriented programming (ROP) attack can be accomplished on a MSP430 MCU 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 and how an attacker can maximize his information gain when dealing with more than one update. It also shows that the DG generated updates leak more information than the R3diff generated updates. This stresses the fact that even delta updates need to be protected with encryption. To defend against this attack, different countermeasures that consider different power consumption scenarios are proposed, but yet to be evaluated.

Keywords Return-oriented programming · IoT · Security · Incremental code update

1 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 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. In many of these application scenarios multihop networks are used. This means an OTA update is sent hop-by-hop i.e. many devices need to receive and send it even those that are currently not updated. The OTA update can consume substantial radio reception power from the device while receiving and sending it. For this reason, it is not wise to send an entire new firmware

✉ Kai Lehniger
lehniger@ihp-microelectronics.com

Abdelaziz Saad
saad@ihp-microelectronics.com

Peter Langendörfer
langendoerfer@ihp-microelectronics.com

1 Wireless Systems, IHP - Leibniz-Institut für innovative Mikroelektronik, Im Technologiepark 25, Frankfurt (Oder) 15236, Brandenburg, Germany

2 Brandenburgische Technische Universität Cottbus-Senftenberg, Platz der Deutschen Einheit 1, Cottbus 03046, Brandenburg, Germany

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]. This is opening a chance for malicious attackers to compromise the devices. Enforcing only authentication and integrity helps to prevent attackers from injecting malicious code, but the fact the confidentiality is optional provides the attackers a chance to create a RoP attack by analysing the code update.

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 does not 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. It however emphasizes that differencing updates do not hide enough information to be used to guarantee confidentiality. In order to prevent the presented attack, different scenarios for power-efficient countermeasures are proposed.

This is an extension of a paper that was already published at CSNET 2021 conference [5]. The main contributions of this work compared to the original paper are a more detailed view on the differencing algorithms and their reconstruction process for the attack, an updated section of related work, as well as a new method to correlate multiple delta updates to find even more gadgets.

The rest of this paper is categorized as follows: In Section 2, the concepts of differencing algorithms and the return-oriented programming (ROP) attack are discussed. Section 3 discusses related work. Section 4 demonstrates the conceptual steps of the attack against the R3diff and DG generated updates. In Section 5, the exploitability of the updates from the two algorithms is compared. Different countermeasures that consider different power consumption scenarios are proposed in Section 6. Finally, Section 7 concludes the paper.

2 Background

2.1 Differencing algorithm

The differencing algorithm takes the old and new firmware images as input and correlates them to produce a differential delta update [6]. A delta update is a specific type of file, which encodes the differences between the two files as a sequence of commands. With the old file as a basis the commands are used to construct the new file. Figure 1 shows how this file is used during the update process.

To create these commands, 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, which represents non-matching segments, to the new firmware.

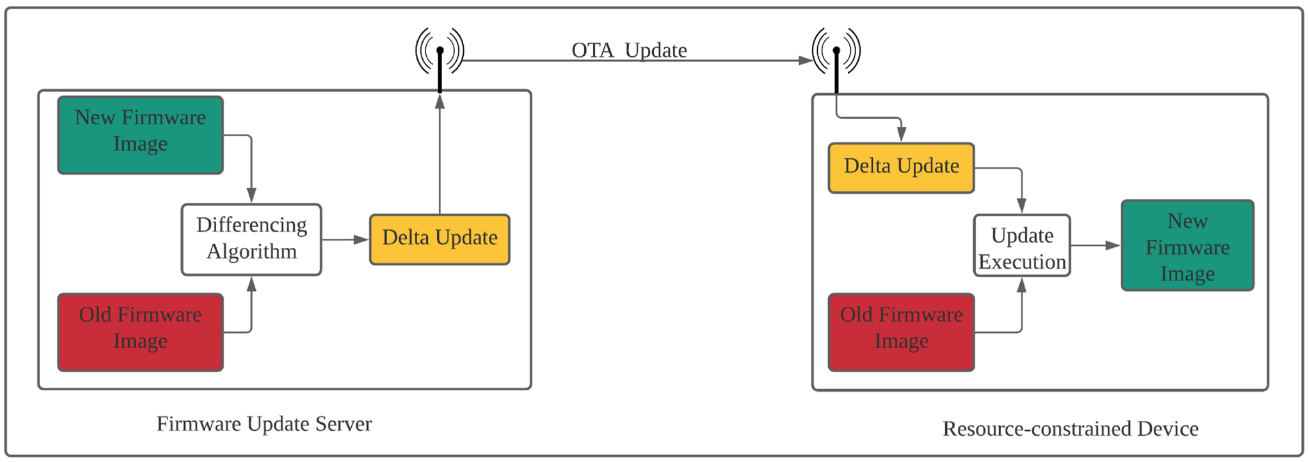

Fig. 1 Over the Air differential update overview

A differencing algorithm can be either in-place or out-of-place [7] which sometimes are called single and dual-bank respectively [8]. This depends on the way its generated updates are executed on the resource-constraint devices. During an in-place update, the commands are used to directly modify the old firmware to create the new firmware image. 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 cannot be reverted back to the old firmware. During an out-of-place update, the old firmware is only used as a source for COPY commands and the new firmware is constructed 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 1 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 1, will be under assessment. We selected these two algorithms to have an example for in-place as well as an example for an out-of-place update mechanism, i.e. we cover all types of update means. While being the most novel approach, DASA-Improved is not considered for evaluation because it is not yet implemented or evaluated [9].

2.1.1 R3diff

R3 was presented by Wei Dong et al. in 2013 [1]. R3 consists of the following algorithms: R3sim, R3con, and R3diff. R3sim tries to maximize the similarity of the two images before the actual differencing algorithm is applied, and R3con is responsible for the construction of the image on the receiver side. R3diff itself was proven to create a minimal delta size for the chosen commands and its encoding. It is only using the following two commands

Table 1 Several differencing algorithms, their types, and their runtime complexity

Algorithm	First appeared	Execution	Time complexity
Rsync [10]	1999	Out-of-Place	$O(n^2)$
FBC [11]	2004	Out-of-Place	$O(n)$
RMTD [12]	2009	Out-of-Place	$O(n^3)$
DASA [13]	2012	Out-of-Place	$O(nlog(n))$
R3diff [1]	**2013**	***Out-of-Place***	$\mathbf{O(n^3)}$
DG [2]	2016	In-Place	$O(n^2)$
DG-Optimized [14]	**2019**	***In-Place***	$\mathbf{O(n^2)}$
DASA-Improved [9]	2020	Out-of-Place	$O(nlog(n))$

- **COPY** $\langle N\rangle\langle \boldsymbol{addr}\rangle$ to copy N bytes form the specified address *addr*, and
- **ADD** $\langle N\rangle\langle \boldsymbol{b_1 ... b_N}\rangle$ to add N bytes to the image.

There is no need for an explicit destination address since all the commands are executed in order to construct the image from the beginning to the end.

2.1.2 DG-Optimized

By using an in-place update strategy DG (represented here by a later iteration DG-Optimized) showed a novel approach to create smaller delta updates, compared to the classical out-of-place strategy. Compared to R3diff and the original DG implementation, DG-Optimized uses more commands with a more complicated encoding.

- **COPY** $\langle \boldsymbol{n}\rangle\langle \boldsymbol{N_1}\rangle\langle \boldsymbol{addr_1}\rangle ... \langle \boldsymbol{N_n}\rangle\langle \boldsymbol{addr_n}\rangle$ works similar to the COPY of R3diff but additionally encodes the number of consecutive COPY commands *n* inside the opcode of the command to save some additional opcode bytes in the delta update.
- **ADD** $\langle N\rangle\langle \boldsymbol{b_1 ... b_N}\rangle$ works exactly as in case of R3diff and adds *N* bytes to the image.
- **INSERT** $\langle \boldsymbol{word}\rangle$ is a variant of the ADD command which only adds a single word. This is particularly useful to change parameters or pointers, in case function positions have changed.
- **SKIP** $\langle N\rangle$ is used to skip unchanged parts of the image.

Similar to R3diff, there are no destination addresses, since the image is processed from the beginning to the end. It only differs in the fact that it operates directly on the current image instead of creating a new one. Therefore the two additional commands are used to move over unchanged areas of the image, or only apply small changes. In general, DG is designed for small changes in the image and therefore works best for those. Since INSERT is just a special ADD command, whenever in this paper ADD commands are referred, INSERT commands are also implicitly included in the context of DG.

2.2 ROP attack

Return-oriented programming (ROP) attacks are classified as code-reuse attacks. The main motivation for these attacks was the Non-Executable stack security hardening for the Linux Kernel in June 1997 by Solar Designer [15, 16]. This patch stopped classical code injection attacks. Shortly after its introduction a bypass was announced by Solar Designer himself in the BugTraq mailing list [17], which became known as return-into-libc attacks.

The 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 [18] as a generalization to classical return-into-libc attacks. 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.

Different mitigation techniques have been developed against code-reuse attacks in general and ROP attacks in particular. Stack Canaries or StackGuards [19] are supported by different compilers for a wide variety of platforms. These are means to protect the stack memory. However, the concept itself can be applied to any part of the memory. The main idea of stack canaries is that the compiler will add a random value just before the return-address. Thus, in case the attacker tries to override the return-address by overflowing a buffer, the canary word will be overwritten too. Before returning from a function, the canary word is checked. The canary is a useful mitigation against ROP attacks as long as it cannot be predicted or brute-forced. However, it does not work against attacks that use function pointers instead of the return-address.

Address Space Layout Randomization (ASLR) [20] is another mitigation technique that is implemented in most modern desktop operating systems, but lacks support in lightweight OS', or bare metal applications. During the start of a process, the memory locations of its segments are randomized, making it hard for the attacker to jump to specific gadgets.

3 Related work

Encryption for firmware updates itself has been recognized as important to prevent attackers from reverse engineering and identifying possible exploits [21]. However, to the best of our knowledge, this was not researched yet for incremental updates. Also, the presented attack approach can be categorized into two steps: leaking gadgets and running the attack itself. There are several different methods described in literature for leaking gadgets. When the application itself is known and only the positions of the gadgets are unknown due to ASLR, leaking a pointer into a shared library can be enough to reveal the position of all its gadgets [22, 23]. The most similar approach to our work was done by Goodspeed and Francillon in 2009 [24]. They also launched a ROP attack on a MSP430 with a similar assumption of not knowing the application code. They only assume to know the bootloader. Their approach of leaking gadgets is by brute-forcing gadget addresses and using system crashes as indication if the guess was correct. They aim only for a very short ROP attack to bypass the protection of the bootloader.

With a similar approach, but without any knowledge about the application [25] describes a method to brute-force gadget positions of server applications by randomly guessing gadget positions and see if the process crashes. If the execution continues, they guessed correctly. The goal is to get enough gadgets to dump the entire binary to the attacker and continue with a normal ROP attack that rely on the knowledge of the binary. Recently, in 2022, this work was extended by Zhang et al. [26] to gather a more complete set of gadgets as dumping the binary can be prevented, i.e. by execution only memory (XOM) [27] which makes the code segments unreadable.

The attack described in [28] uses multiple steps to attack a vulnerable software that is protected with Intel's Software Guard Extensions (SGX) and is running in an enclave, completely isolated from the rest of the system. First page faults are used to find gadgets that pop contents from the stack, second an `EEXIT` leaf function call is used to identify the registers to which the content was popped into and third a gadget with a memcpy is searched. These gadgets are enough to leak encryption keys and the hidden binary, which allows to take complete control of the attacked process.

All these attacks use leakage sources on the attacked device as they assume the vulnerable software has already been distributed or has been distributed in a way that does not allow an attacker to gain information, i.e. encryption. They also have in common that they are actively probing/brute-forcing for gadgets which makes this part of the attack detectable due to the high amount of crashes or messages.

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. [29] 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 attack similar to the one that has been discussed in the paper.

4 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.

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 defences are either not applicable in the resource-constraint world, as they 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 [30] that led to severe vulnerabilities and the 2nd in the 2020 report [31]. 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 does not have special hardware such as Trusted Execution Environment (TEE). Also, we assume that the updates are sent unencrypted.

Exploitation steps As shown in Fig. 2, an attacker executes the following steps to extract 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 does not 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 sees which encoding of which algorithm matches 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 decodes the update based on the detected differencing algorithm above and identifies the destination addresses of the COPY and ADD commands based on the detected differencing algorithm in step 2. In Fig. 2, the grey color represents the identified COPY commands. Green represents the payload of an ADD command, together with its destination address in red.
4. Generally, a delta update needs the firmware image to be executed to completely construct the new firmware version. Since this image is unknown to the attacker by definition, the full reconstruction is not possible. The process of partially reconstructing the image works in the same way for R3diff and DG and only differs in the way the commands need to be evaluated.

   Since the source for COPY commands is unknown, they contain no valuable information that could be restored. Only ADD commands (marked green) contain a payload that represent parts of the actual image. However, to find the destination address for an ADD command, which is necessary to insert the found instructions at their correct addresses, all commands must be tracked. This is important for the gadget localization later. This is due to the fact that the destination addresses are only implicitly given, for the selected algorithms. When all the destination addresses for the ADD commands have been calculated, the image can partially be reconstructed by locating the payload at the given destination address for each ADD command. Alternatively, the partial image reconstruction can be seen as a normal delta execution, but without the necessary input file.

   The result is an image which only contains the instructions of the ADD commands. All the regions that are constructed with COPY commands (marked grey) are left blank.
5. In the last step, the reconstructed parts of the images are searched for gadgets. There exists a number of tools such as mona [32], Ropper and ROPgadget [33, 34] that can automate this process. However, until the time of writing this paper, there were no off-the shell tools for

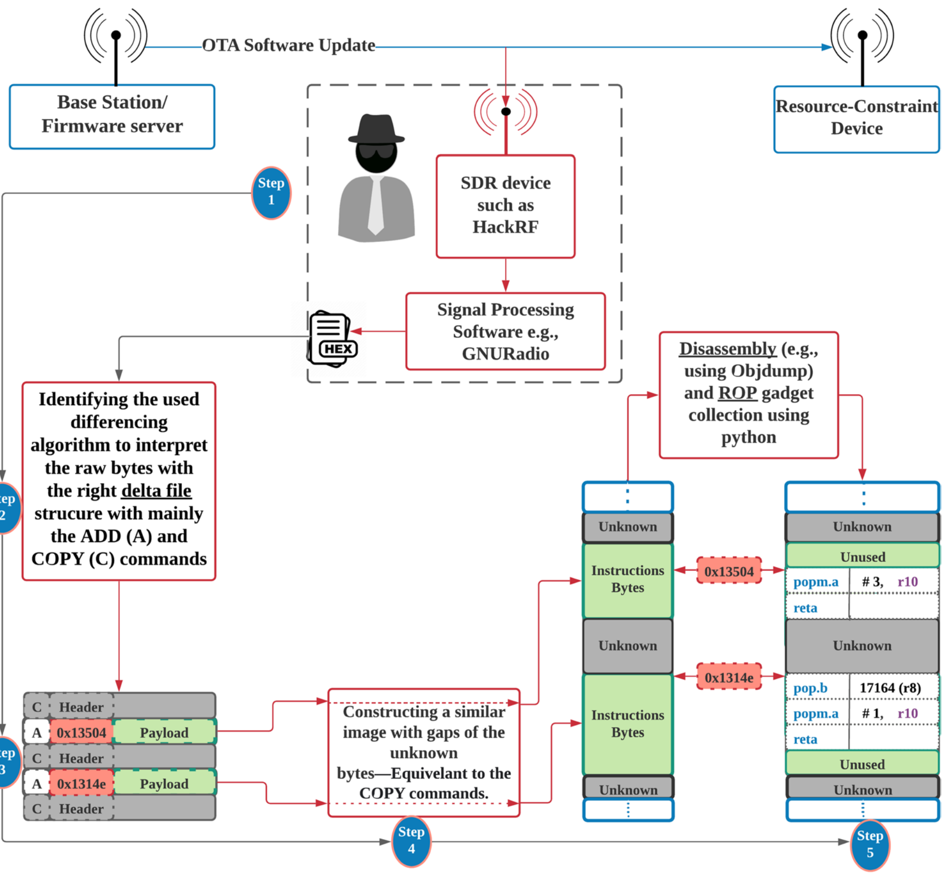
OTA Software Update
Base Station/ Firmware server
Resource-Constraint Device
SDR device such as HackRF
Signal Processing Software e.g., GNURadio
HEX
Step 1
Step 2
Step 3
Step 4
Step 5
Identifying the used differencing algorithm to interpret the raw bytes with the right delta file strucure with mainly the ADD (A) and COPY (C) commands
Disassembly (e.g., using Objdump) and ROP gadget collection using python
C Header
A 0x13504 Payload
C Header
A 0x1314e Payload
C Header
Constructing a similar image with gaps of the unknown bytes—Equivelant to the COPY commands.
Unknown
Instructions Bytes
Unknown
Instructions Bytes
Unknown
0x13504
0x1314e
Unknown
Unused
popm.a #3, r10
reta
Unknown
pop.b 17164 (r8)
popm.a #1, r10
reta
Unused
Unknown

Fig. 2 Steps to extract gadgets out of an incremental update: (1) eavesdropping messages, (2) identifying the differencing algorithm, (3) decode delta update information, (4) partly reconstructing the image, (5) finding ROP gadgets

the MSP430 MCU architecture, that was used during the evaluation.

5 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.

5.1 First evaluation example

The results of the first example are depicted in Tables 2 and 3. In Table 2, firmware with size **180KB** has been upgraded to a new firmware with size **177KB** that contains **380** ROP gadgets. In Table 3, 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 2 and 3, 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 length 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 is 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 Table 2 and leaked 18 gadgets in the second update as it is shown in Table 3. 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 Listings 1 and 2 were found. Using these two gadgets, it was still possible to attack the device with the “write anything, anywhere” power.

5.1.1 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

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

Listing 1 A ROP gadget that assigns an arbitrary 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

Table 2 The analysis of updating a firmware with size 180k to a firmware with 177k, the tables 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 3 The analysis of updating a firmware with size 177k to a firmware with 180k, the tables also concludes that the number of collected gadgets in case of DG is much Higher then 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%**

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.

5.1.2 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.

5.2 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 4.

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

In the evaluation that is presented in Table 4, the tracking of the multiple versions of the firmware binary is 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 4, 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 do not work alone, the other collected gadgets can also be helpful. The “useful” word here indicates that those gadgets are worth 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.

With R3diff and DG we analysed the vulnerability of the two types of differential code update means, i.e. in-place and out-of-place. The latter shows by far less vulnerability, which may lead to the assumption that selecting the proper mechanism is sufficient to prevent attackers from being successful. But the fact that each update is providing sufficient gadgets to construct a ROP attack it becomes clear that additional protection means are essentially needed.

5.3 Correlating updates

When the number of gadgets is relatively low, as for the results with the R3diff algorithm and the number of gadgets of an update is not sufficient, the attacker can always wait for the next update. However, if an attacker fails to gather enough gadgets in a single update, it is possible to increase the number of overall gadgets by correlating the results.

While for a single delta only the ADD commands contain valuable information, with multiple delta files the COPY commands also gain importance as this allows us to track if a gadget we already discovered is moved to a new position. We therefore extended our analysis for the second evaluation example by also tracking the COPY commands. The results are presented in Fig. 3. It shows the number of gadgets plotted for the same firmware versions as in Table 4, with the number of gadgets in the original binary in blue and for the deltas in red. Additionally it also contains the number of gadgets when the information of the deltas is correlated, represented by the brown plot. With the correlation we were

Table 4 Tracking an environment measurement firmware evolution and calculating the number of ROP gadgets from the complete firmware images, R3diff and DG updates

MSP430	Firmware				R3dif f			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**

able to increase the number of found gadgets from the previously 110 to now 179.

It is also important to track COPY commands that do not move gadgets directly, as they can still overwrite gadgets of the previous version. This can also mean that even with correlation the number of found gadgets can decrease with more updates. To illustrate this we included an additional version 1.5 in Fig. 3 with a lot of new code, resulting in very large deltas between this new version and version 1, as well as a lot of gadgets that can be found. With the update to version 2, most of these found gadgets are then being overwritten by COPY commands. Of course a lot has to do with the fact that version 1.5 overall has more gadgets than version 2. However, with version 1.5 included in the correlation it can also be noted that for versions 2 and 3 the overall number of gadgets is slightly higher than without.

In conclusion we think correlating multiple deltas can be greatly beneficial for the overall number of found gadgets. The more deltas can be acquired the more information about the firmware can be gathered; however, there is no guarantee that the number of gadgets will always increase.

6 Countermeasures

In this paper we showed experimentally that incremental code updates sent unencrypted provide sufficient gadgets to construct a ROP Attack. Even more although out-of-place update mechanisms are less vulnerable when it comes to the number of gadgets they still leak a sufficient set of gadgets so that using a different update scheme is not solving the issue. In order to reliably prevent an attacker from getting gadgets, ensuring confidentiality of the incremental updates is key. The tricky thing with this is the power consumption of this process. As the devices are mostly battery driven energy-efficiency is of utmost importance and needs to be considered when designing and/or applying an encryption scenario. Here we are discussing multiple scenarios considering the limited power available in the devices. They range from already existing encryption schemes (with their already shared keys) to using built-in encryption of Over The Air Programming (OTAP) schemes. The following scenarios can be used if full encryption is not feasible.

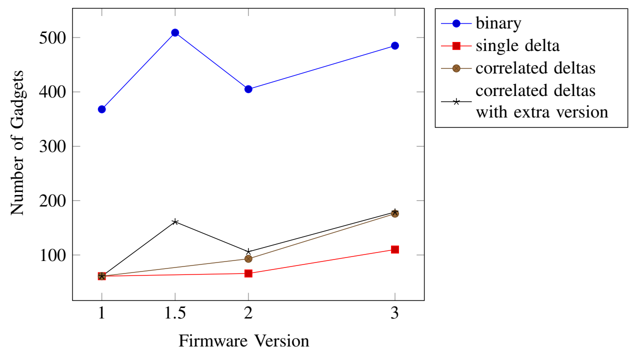

Fig. 3 Number of gadgets for R3diff delta correlation

6.1 Partial encryption of incremental updates

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 make 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. This approach is better suited for R3diff than for DG due to the COPY-ADD ratio. For example, in Table 2 the DG delta (44KB) consists of 37KB payload while the R3diff delta (26KB) only consists of 10KB payload.

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 Fig. 4. The delta script encoding and decoding need also to be modified to adopt this separation 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 Fig. 5. 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.

6.2 OTAP built-in encryption

In this scenario, based on the assumption that the attacker does not have access to the currently deployed firmware on the resource-constraint device, we do not 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

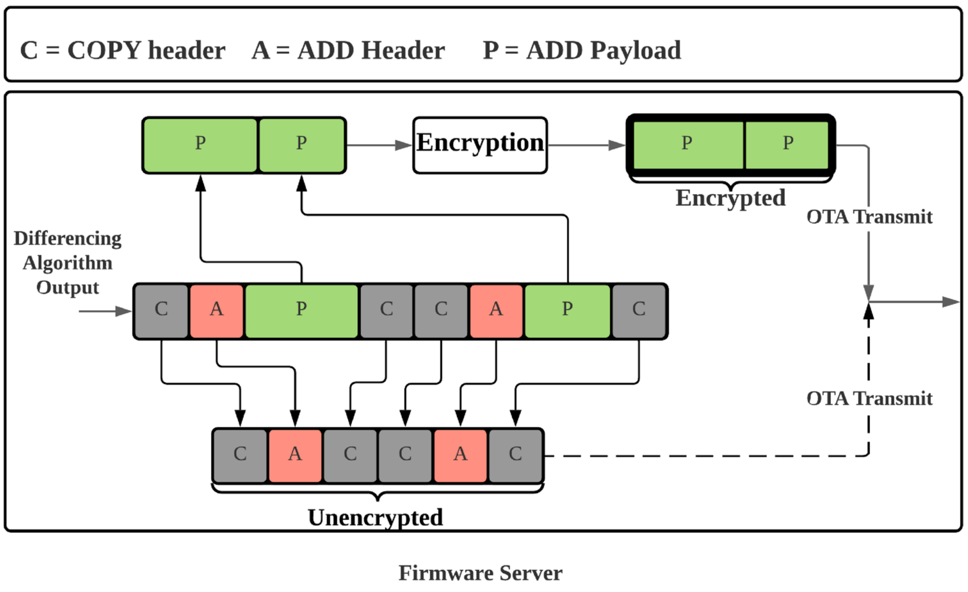

Fig. 4 Separating the ADD command payloads from the rest of the delta update for encryption

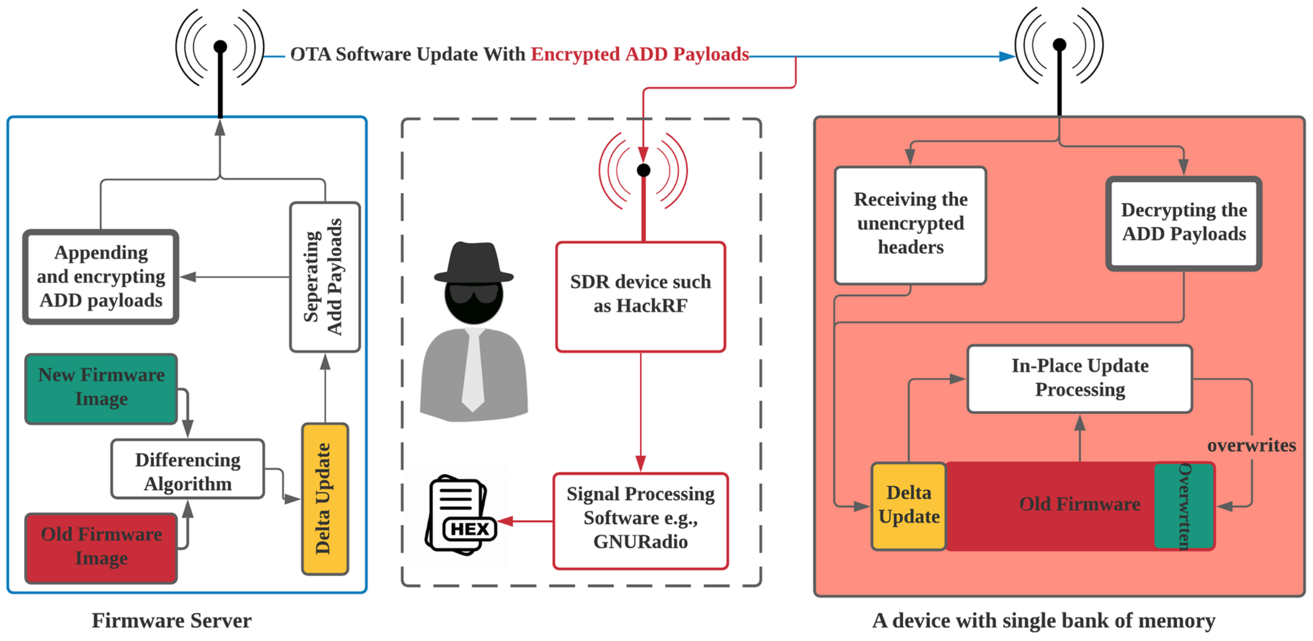
OTA Software Update With Encrypted ADD Payloads
Appending and encrypting ADD payloads
Seperating Add Payloads
New Firmware Image
Differencing Algorithm
Old Firmware Image
Delta Update
Firmware Server
SDR device such as HackRF
HEX
Signal Processing Software e.g., GNURadio
Receiving the unencrypted headers
Decrypting the ADD Payloads
In-Place Update Processing
overwrites
Delta Update
Old Firmware
Overwritten
A device with single bank of memory

Fig. 5 Changes in the delta script transmission and reception processes in the firmware server side and the resource-constrained device side

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.

6.2.1 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 the 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 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.

This approach comes with the extra benefit that each update implicitly also updates the “built in key”. This limits the time interval an attacker has to reveal that key to get access to the unencrypted data. But once the attacker manages to reveal a key he needs only to track the updates in order to always have the currently valid key.

6.2.2 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.

6.2.3 Embedding an encryption key in the base image

Since the presented 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.

Therefore, 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 use 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. Here a single key for all devices may be used which is normally considered to weaken the security of the network. The reason is that if an attacker reveals ROP gadgets for one device these gadgets work with other devices as well, independent of whether updates of the other devices are encrypted with the same or a different key. It is also possible that an update can change the pre-embedded key to another fresh key.

7 Conclusion

In this paper, we analysed the risk that arises from unencrypted incremental code updates with respect to building ROP attacks. In order to provide a comprehensive study we analysed both types of incremental code updates, i.e. in-place and out-of-place, investigating two representative approach DG and R3diff respectively. Our analysis clearly revealed that despite DG provides by far more gadgets to the attacker, also the gadgets to be extracted from R3diff updates are sufficient to generate a ROP attack. We also showed that an attacker can gather more gadgets, e.g. to build more sophisticated attacks or to reduce his effort in designing the attack, by just recording more updates.

As both types of update mechanisms are leaking sufficient gadget to construct ROP attacks, adapting the type of update approach is not preventing any ROP attack and additional means are needed. We addressed this point by discussing different power-efficient means to ensure confidentiality of the incremental update messages. All of them apply encryption based on pre-shared key, in one of them the old image is used as pre-shared key which avoids any issues with key distribution but requires to keep the original code image confidential.

Funding Open Access funding enabled and organized by Projekt DEAL. This work was supported by the Federal Ministry of Education and Research (BMBF) under research grant number 01IS18065E.

Declarations

Consent to participate Not applicable

Ethics approval Not applicable.

Consent for publication Not applicable.

Competing interests Not applicable.

References

1. Dong W, Mo B, Huang C, Liu Y, Chen C (2013) R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems. In: 2013 Proceedings IEEE INFOCOM, pp 315–319. https://doi.org/10.1109/INFCOM.2013.6566786
2. Kachman O, Balaz M (2016) 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), pp 1–4. https://doi.org/10.1109/DDECS.2016.7482473
3. Zandberg K, Schleiser K, Acosta F, Tschofenig H, Baccelli E (2019) Secure firmware updates for constrained iot devices using open standards: a reality check. IEEE Access 7:71907–71920
4. A Firmware Update Architecture for Internet of Things. https://tools.ietf.org/html/draft-ietf-suit-architecture-16. Accessed 14 July 2022
5. AbdElaal ASA, Lehniger K, Langendörfer P (2021) Incremental code updates exploitation as a basis for return oriented programming attacks on resource-constrained devices. In: 2021 5Th cyber security in networking conference (CSNet), pp 55–62. https://doi.org/10.1109/CSNet52717.2021.9614275
6. Arakadakis K, Charalampidis P, Makrogiannakis A, Fragkiadakis A (2020) Firmware over-the-air programming techniques for IoT networks – A survey. In ACM Computing Surveys (CSUR), pp 1–36
7. Lehniger K, Weidling S (2019) The impact of diverse execution strategies on incremental code updates for wireless sensor networks. In: Benavente-Peces C, Ahrens A, Camp O (eds) Proceedings of the 8th international conference on sensor networks, SENSORNETS 2019, Prague, Czech Republic, February 26-27, 2019, pp 30–39. https://doi.org/10.5220/0007383400300039
8. Kachman O, Balaz M (2020) 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), pp 1–6. https://doi.org/10.1109/DDECS50862.2020.9095744
9. Arakadakis K, Fragkiadakis A (2020) Incremental firmware update using an efficient differencing algorithm: poster abstract. In: Proceedings of the 18th Conference on Embedded Networked Sensor Systems. SenSys 2020. J Assoc Comput Mach pp 691–692. https://doi.org/10.1145/3384419.3430471
10. Tridgell A (1999) Efficient algorithms for sorting and synchronization. Doctoral dissertation. The Australian National University
11. Jaein J, Culler D. (2004) 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., pp 25–33. https://doi.org/10.1109/SAHCN.2004.1381899
12. Hu J, Xue CJ, He Y, Sha EH (2009) Reprogramming with minimal transferred data on wireless sensor network. In: 2009 IEEE 6Th international conference on mobile adhoc and sensor systems, pp 160–167. https://doi.org/10.1109/MOBHOC.2009.5337000
13. Mo B, Dong W, Chen C, Bu J, Wang Q (2012) An efficient differencing algorithm based on suffix array for reprogramming wireless sensor networks. In: 2012 IEEE international conference on communications (ICC), pp 773–777. https://doi.org/10.1109/ICC.2012.6364214
14. Kachman O, Baláz M, Malík P (2019) Universal framework for remote firmware updates of low-power devices. Comput Commun 139:91–102
15. Linux kernel patch from the Openwall Project (1997) https://www.openwall.com/linux/README.shtml
16. Non-Executable Stack Patch (1997) http://lkml.iu.edu/hypermail/linux/kernel/9706.0/0341.html. Accessed 14 July 2022
17. Solar's Clearification of the possible bypasses. https://www.openwall.com/linux/README.shtml. Accessed 14 July 2022
18. Shacham H (2007) The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). CCS '07. Association for Computing Machinery, 552–561. https://doi.org/10.1145/1315245.1315313
19. Cowan C, Pu C, Maier D, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q, Hinton H (1998) Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX Security Symposium, vol 98, San Antonio, TX. 63–78
20. Team P (2003) Pax address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt. Accessed 14 July 2022
21. Falas S, Konstantinou C, Michael MK (2021) A modular end-to-end framework for secure firmware updates on embedded systems. ACM J Emerg Technol Comput Syst 18(1):1–19
22. Serna FJ (2012) Cve-2012-0769, the case of the perfect info leak. In: Blackhat conference, feb
23. Snow KZ, Monrose F, Davi L, Dmitrienko A, Liebchen C, Sadeghi A-R (2013) Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: 2013 IEEE symposium on security and privacy. IEEE, pp 574–588
24. Goodspeed T, Francillon A (2009) Half-blind attacks: mask rom bootloaders are dangerous. In: Proceedings of the 3rd USENIX conference on offensive technologies. USENIX Association, pp 6–6
25. Bittau A, Belay A, Mashtizadeh A, Mazières D, Boneh D (2014) Hacking blind. In: 2014 IEEE Symposium on security and privacy, pp 227–242. https://doi.org/10.1109/SP.2014.22
26. Zhang T, Cai M, Zhang D, Huang H (2022) Sebrop: blind rop attacks without returns. Front Comp Sci 16(4):1–18
27. Lie D, Thekkath C, Mitchell M, Lincoln P, Boneh D, Mitchell J, Horowitz M (2000) Architectural support for copy and tamper resistant software. Acm Sigplan Notices 35(11):168–177
28. Lee J, Jang J, Jang Y, Kwak N, Choi Y, Choi C, Kim T, Peinado M, Kang BB (2017) Hacking in darkness: Return-oriented programming against secure enclaves. In: 26Th USENIX security symposium (USENIX security 17), pp 523–539
29. Bettayeb M, Nasir Q, Talib MA (2019) Firmware update attacks and security for iot devices: Survey. In: Proceedings of the ArabWIC 6th annual international conference research track. ArabWIC 2019. J Assoc Comput Mach https://doi.org/10.1145/3333165.3333169
30. CWE Top 25 Most Dangerous Software Errors (2019) Common Weakness Enumeration. https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html (accessed 14 July 2022)
31. CWE Top 25 Most Dangerous Software Weaknesses (2020) Common Weakness Enumeration. https://cwe.mitre.org/top25/archive/2022/2020_cwe_top25.html (accessed 14 July 2022)
32. Inventory RC (2020) Mona. https://github.com/corelan/mona
33. Schirra S (2020) Ropper. https://github.com/sashs/Ropper
34. Salwan J (2020) ROPgadget. https://github.com/JonathanSalwan/ROPgadget