An in-depth security evaluation of the Nintendo DSi gaming console

pcy Sluys[1], Lennert Wouters[1 [0000−0001−5740−5066]], Benedikt Gierlichs[1 [0000−0002−5866−1990]], and Ingrid Verbauwhede[1 [0000−0002−0879−076X]]

COSIC, KU Leuven, Kasteelpark Arenberg 10, Heverlee, Belgium
firstname.lastname@esat.kuleuven.be

Abstract. The Nintendo DSi is a handheld gaming console released by Nintendo in 2008. In Nintendo's line-up the DSi served as a successor to the DS and was later succeeded by the 3DS. The security systems of both the DS and 3DS have been fully analyzed and defeated. However, for over 14 years the security systems of the Nintendo DSi remained standing and had not been fully analysed. To that end this work builds on existing research and demonstrates the use of a second-order fault injection attack to extract the ROM bootloaders stored in the custom system-on-chip used by the DSi. We analyse the effect of the induced fault and compare it to theoretical fault models. Additionally, we present a security analysis of the extracted ROM bootloaders and develop a modchip using cheap off-the-shelf components. The modchip allows to jailbreak the console, but more importantly allows to resurrect consoles previously assumed irreparable.

Keywords: Nintendo DSi · boot ROM · fault injection · secure boot · modchip · embedded security

1 Introduction

The Nintendo DSi is a handheld gaming console released by Nintendo in 2008. It was designed as a small upgrade to its predecessor, the Nintendo DS, adding extra 'multimedia' features such as two cameras and a web browser. These consoles still have active *homebrew* communities of people making their own self-published games and programs. Additionally, these communities are interested in reverse-engineering the publicly undocumented hardware. This is done to develop more precise emulators, write code that is capable of rendering more spectacular graphical effects, or simply as a reason in and of itself.

The security systems of the DS [6] and the 3DS [30, 8, 31] have been fully analyzed and defeated. This has not yet been the case for the DSi.

The goal of this research is to analyze the previously unexamined parts of the security system of the Nintendo DSi, more specifically, its *boot ROMs*. This would allow for better hardware preservation and brick recovery: current exploits [17] rely on the second-stage bootloader residing in eMMC (Embedded MultiMedia Card) memory existing and having a correct digital signature. This eMMC, a

Samsung moviNAND chip [16], has a low erase-write-cycle lifetime, and might be affected by bugs in the wear-levelling management firmware (this is the case for other eMMCs made by the same manufacturer in that era [24]). Simply replacing the eMMC chip with another would not fix the situation, as the console uses the eMMC's CID (Card IDentifier, a uniquely identifying number of every eMMC memory) to derive cryptographic keys used to encrypt the FAT32 filesystem [16]. Furthermore, this work can be used as inspiration on how to tackle other, similar targets.

This paper is structured as follows: section 2 provides background information and covers related work. In sections 3 and 4, we show how the boot ROMs can be extracted. Section 5 reflects on the fault injection campaign and proposes an explanation of what type of faults actually occur when performing the readout attack. A security analysis is then performed in section 6. The results of this analysis are used in section 7 to build a modchip capable of jailbreaking the console. Section 8 then provides a conclusion.

1.1 Contributions

This work presents the following contributions: We extract the boot ROMs of the DSi using a second-order EMFI attack. We then look into the security aspects of these ROMs, completing the security analysis of the Nintendo DSi. Finally, we develop a modchip able to jailbreak the system in its very first bootstage. This modchip attack can be used to revive bricked consoles with a broken eMMC.

1.2 Responsible disclosure

We did not disclose our research results to Nintendo ahead of submission for several reasons.

First, Nintendo appears to only accept vulnerability reports through their HackerOne bug bounty program. At the time of writing Nintendo only accepts submissions for the Switch console. Note that the 3DS (the DSi's successor) is explicitly listed as out of scope[1]. Secondly, by submitting a report through the HackerOne program we would agree to not publish our findings, even if the report is considered out of scope. Finally, Nintendo discontinued the DSi[2], and no new game titles have been released since 2016[3]. Due to the above factors, vulnerability disclosure to the vendor is currently not considered.

2 Background and related work

This section gives an overview of the hardware of the Nintendo DSi, fault injection, and previous attacks on the DSi. These elements are necessary to under-

[1] See `https://hackerone.com/nintendo/updates`, `https://archive.ph/Yh7YV`.
[2] The exact date is unclear. Nintendo never announced an official date when the DSi would go out of support, instead changing the console's status silently. The 3DS was discontinued in 2020. [3] *Crazy Train*, a downloadable DSiWare title.

stand the attacks used in this work, and the current state-of-the-art regarding DSi exploits.

2.1 The Nintendo DSi gaming console

The DSi is an interesting hybrid between the DS and 3DS: it keeps the former's CPU and GPU, while the peripherals, chipset, boot process and overall security architecture resemble those of the 3DS much more closely.

The DSi, much like the DS, has two CPU cores, an ARM7TDMI and an ARM946E-S, typically shortened to respectively ARM7 and ARM9. The ARM7 is used for I/O tasks and has exclusive access to many I/O peripherals, while the ARM9 is much faster and more powerful, and has exclusive access to the GPU. It has SoC-internal SRAM specific to each separate CPU core with configurable mapping options, and external DRAM shared between the two cores. These CPUs can also communicate using a FIFO interface. Unlike the DS, it boots from eMMC NAND, which contains the second-stage bootloaders (in raw eMMC blocks) as well as the system menu and various apps (on an encrypted FAT32 filesystem). It has extra peripherals such as cameras and an SD card slot. More information about the DS and DSi can be found in [14].

The DS only used symmetric-key Blowfish encryption without authentication, and a system menu residing in external flash without cryptographic protection mechanisms. This naturally lead to the proliferation of 'flashcarts' [6], on which homebrew (and pirated) games can be loaded and played. Compared to this, the DSi and 3DS both use a full secure boot chain using digital signatures and AES encryption, from the first bootloader [8, 5] down to individual games and applications [26].

2.2 Fault injection

Fault injection is an attack method targeting the physical implementation of a device, by actively tampering with its operation. By bringing one or more environmental parameters (such as supply voltage, clock signal, incident electromagnetic field, etc.) outside the operating range of the device for a short amount of time, the target can be made to malfunction without crashing or shutting down. After these parameters return to their normal range, the effects of this malfunctioning can still propagate logically, possibly subverting the security properties of the device [41, 1, 34]. Naturally, people have proposed countermeasures to stop such attacks [2], resulting in an arms race between attackers and defenders.

Fault injection can be used to circumvent security checks [12], tamper with cryptographic algorithms to obtain secrets [9, 35], and even directly take control over the execution flow of a processor [28, 37]. In the context of gaming consoles, this often translates to obtaining decryption keys [18, 10] and arbitrary code execution capabilities in a high-privilege environment (e.g. a bootloader, hypervisor, or security coprocessor) [11, 8].

2.3 Earlier work on the DSi

Interestingly enough, it was the 3DS of which the security system was fully broken first. From software exploits in the operating system [26] to bootloader 'unlocks' [20]. Finally, the boot ROMs were extracted aswell [8], leading to the discovery of fatal vulnerabilities [30, 31].

The ROM extraction method presented by derrek et al. is very relevant to this work [8]. Their method relies on two quirks: SRAM is not cleared across resets, and some exception vectors in the ROM are hardcoded to jump into SRAM. Using fault injection, it is possible to cause an undefined instruction exception early during boot ROM execution. By first poisoning SRAM with a payload, resetting the SoC and then quickly injecting faults, an attacker can thus obtain code execution while the boot ROMs are executing [8]. Once ROM images had been obtained, it became clear that the boot ROMs contain a vulnerability in the PKCS#1 ASN.1 parsing code [31].

Meanwhile, the DSi had survived earlier attempts at breaking into its security system, and still stood strong at the time the 3DS was released. For example, Micah Elizabeth Scott built a setup to trace all DRAM accesses [32], but this only lead to exploits in specific games, not in the full system. As DRAM is initialized only by the second-stage bootloader before loading the System Menu, DRAM probing could not be used against any bootloader of the DSi. In addition, the `SCFG` control registers [15] are used to mitigate such attacks as well: they can be used to prohibit the CPU from accessing I/O registers related to eMMC, the SD card, WiFi, etc., until a reboot happens. This way, a cartridge-based game cannot access the eMMC filesystem, for example.

However, as the 3DS included a DSi backwards compatibility mode (including emulating the DSi bootchain starting from the second-stage bootloader), the defeat of the 3DS opened a new avenue for analyzing its predecessor. As it was now possible to decrypt and reverse-engineer the DSi's second-stage bootloader, a vulnerability was discovered here [17], allowing for persistent arbitrary code execution capabilities, at cold boot.

One downside of this jailbreak is that it targets the second-stage bootloader, rather than the ROM bootloader. This means the console's eMMC memory still needs to contain a valid cryptographic signature along with an intact second-stage bootloader. As already mentioned in section 1, this eMMC memory is prone to failure, rendering the console unable to boot.

3 ARM7 ROM extraction

The 3DS ROM extraction method described in the previous section can be used to extract one of the two boot ROMs of the Nintendo DSi. This section describes how we extracted this boot ROM, and what information is contained within.

3.1 Method

The method used here is similar to the one used for the 3DS. SRAM contents persist across resets, and the ROM is hardcoded to jump into SRAM when an

undefined instruction exception occurs. By first filling SRAM with a payload, then resetting the SoC and injecting a fault, an attacker can obtain arbitrary code execution capabilities on the ARM7.

3.2 Practical considerations

It was already theorized to be possible to extract the ARM7 boot ROMs using fault injection, though earlier attempts using VFI led to no results [23, 7]. Instead we opted to use EMFI here, as it seemed more practical with the wire-bond BGA package of the SoC, and the complex layer stackup and connection layout of the power supply rails on the PCB. The used fault injection setup consists of a NEWAE ChipSHOUTER as EMFI injector, and a NEWAE ChipShover as positioning stage.

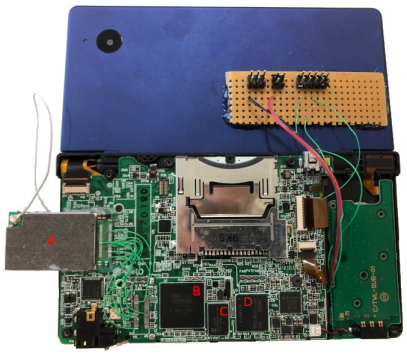
The jailbreak exploit described in section 2.3 is a prerequisite for this attack. It is used to run custom code that fills SRAM with payload code, from which the attack can be performed. More specifically, a region of memory called *WiFi RAM* is used to store the ROM extraction payload. This RAM serves as a queue for WiFi packets, and is untouched by any bootloader. It is thus guaranteed to survive during execution of the boot ROM. The rest of SRAM is filled with NOP sleds (valid as both ARM and Thumb code) that jump to the payload in WiFi RAM.

However, this setup comes with a few downsides. EMFI requires moving the WiFi daughterboard and shielding that are normally placed above the SoC and DRAM. However, the SPI bus of the daughterboard does need to remain connected, as it contains a SPI flash memory used by the boot ROM (cf. section 6.2). It is required to exist in order for the system to reach the state where the second-stage bootloader jailbreak attack is executed. This is worked around by moving the daughterboard to the side, and soldering thin wires to reconnect the SPI bus. A photo of the modified target can be found in Figure 1a.

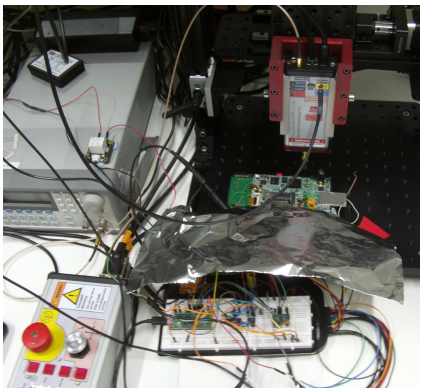
The Raspberry Pico was chosen as the controller for the setup. It is fast, can be controlled on a low level, supports USB as a communication method, and its PIO state machines allow for fast and precise control of glitch pulses. These pulses are sent to the ChipSHOUTER using signals from the target as trigger inputs. It is able to assert the reset line of the target, and watches the `GPIO330` and `CAM_LED` lines as trigger and success signals. The Pico also acts as a new device on the target's I2C bus as a backchannel for `printf`-style debugging of payloads. A desktop computer sends FI parameters to the Pico (using a USB-UART connection) and controls the ChipShover positioning.

To find the optimal combination of fault injection parameters, a divide-and-conquer approach is used. A parameter sweep determines the probe positioning, coil voltage and pulse width to raise an undefined instruction exception in the ARM7. This sweep is done using a test payload injected using the attack from [17] that tries to to replicate the situation when the boot ROM is running. For these tests, the ARM7 runs a test payload that fills SRAM with payload code and sets up the undefined instruction exception handler to jump to it. The payload signals success using the `CAM_LED` line whenever an undefined instruction

exception occurs, while sending out a CPU register dump over I2C. The optimal timing to inject a fault is then discovered by simply sweeping through the entire range (between reset release and the first activity on the eMMC bus), using a payload that performs the ARM7 ROM extraction attack.



(a) The modified DSi with a relocated WiFi daughterboard (A). This configuration makes it possible to target the SoC using EMFI. The large square IC next to this connector footprint is the SoC (B). Next to the SoC are the DRAM (C) and eMMC (D).



(b) A photo of the EMFI setup to extract the boot ROM. The target DSi sits on the stepper table, with a ChipSHOUTER hanging above. In front sits a breadboard with a Raspberry Pico and supporting components (e.g. a level shifter), under which a logic analyzer is placed to inspect the whole system.

Fig. 1: Photos of the modified target and the EMFI setup

3.3 Results and analysis

The attack worked, and the ARM7 boot ROM has been extracted successfully. It was possible to obtain a dump approximately once every 90 seconds, with one attempt made per second.

Static analysis of this boot ROM reveals that it mostly contains driver code for various non-volatile memories. The second-stage bootloader is read from one of these memories, depending on a configuration byte in SPI flash. Additionally, it became clear that the cryptographic verification of the second stage bootloader is performed by the ARM9 processor. Communication between the two cores happens using the FIFO interface. The ARM9 ROM image is thus needed to perform a security analysis of the system.

Nevertheless, the ARM7 ROM contains some information that will be useful later on. Right before jumping to the code of the second-stage bootloader (after

the latter has been decrypted and verified by the boot ROMs), both ROMs are completely detached from the system memory buses. More specifically, the ARM7 ROM writes to an MMIO register to disable both ROMs [15], while the ARM9 waits for this transaction to be completed.

4 ARM9 ROM extraction

In this section, a method of extracting the ARM9 boot ROM is described. This method, using a second-order fault injection attack, is then used in practice.

4.1 Method

The fault injection exploit used to extract the ARM7 ROM can be used as a starting point. Once the console boots, a first fault is injected to take control of the ARM7. The payload then continues booting normally, but 'forgets' to detach the ROMs from the system buses. This will leave the ARM9 stuck in an infinite loop (cf. section 3.3). A second fault can be injected to break the ARM9 core out of this loop and continue the boot process normally, while both ROMs are still readable. As soon as execution ends up in an applet or game (running custom code inserted using a pre-existing exploit such as [17]), the ROM can be read out and e.g. be saved to the SD card.

4.2 Practical considerations

The method described above is a second-order fault injection attack, i.e. it requires *two* successful faults. While such attacks tend to be seen as difficult to pull off (e.g. the authors of [38] call it 'unrealistic'), and countermeasures rarely exist, they have already been performed successfully before [4, 13, 10].

The same EMFI setup used to extract the ARM7 ROM image is used here as well. Similarly, the ARM7 parameter sweeps for the ARM7 takeover part of the ARM9 ROM extraction attack can be reused as-is. Only the probe positioning, coil voltage and pulse width for breaking the ARM9 out of an infinite loop still needs to be determined. A test payload is used here as well: the ARM9 is placed in an infinite loop, while timer interrupts signal a 'heartbeat' message to the Pico to detect crashes. It is, just like for the ARM7 parameter sweep, injected using the Unlaunch exploit. Once these steps have been completed, the parameters can be combined for the final boot ROM extraction attack. A photo of the setup is provided in Figure 1b.

4.3 Results

The attack worked, and the ARM9 boot ROM has been extracted successfully. The timeline of a successful exploit is shown in Figure 2. The success rate was high enough to obtain a dump once every ≈ 90 minutes.

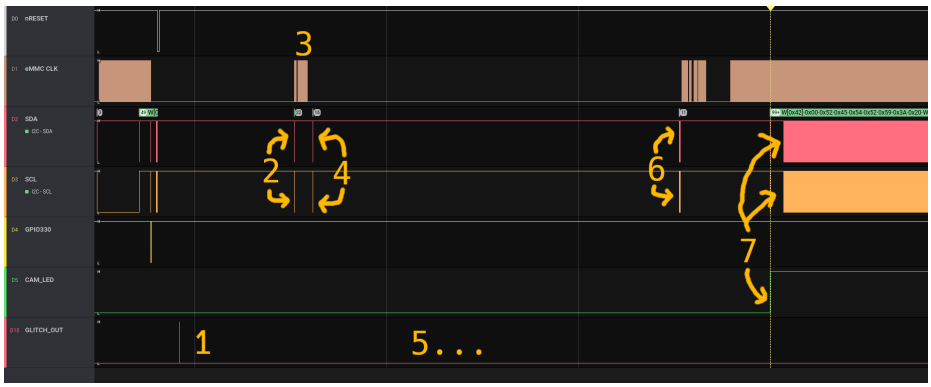


Fig. 2: Logic analyzer capture of the ARM9 boot ROM extraction process. Some time after reset release, a first fault is injected (1). The ARM7 payload then starts mimicking the regular boot ROM execution (2-4). After this, the SoC hangs as the ARM9 waits for the ROMs to be locked away (5), which the ARM7 payload does not do. After injecting a fault into the ARM9 successfully (not pictured on the `GLITCH_OUT` line), the console continues booting (6) and the result is transferred over I2C (7).

5 Fault model analysis

This section looks into the faults injected in the previous sections, and proposes an explanation of the real faults occurring (rather than the ones aimed for in section 3.1), based on observations made during the parameter search.

5.1 Method

Observations come from two sources. One source is the influences of variations in fault parameters during the parameter search. The second is the state of the ARM7 CPU right after takeover: the payload used dumps the CPU and SCFG MMIO registers to the I2C backchannel.

5.2 Observations

One expects a fault on the ARM7 to corrupt an instruction, turning it into an undefined opcode. When decoding such an instruction, the CPU will then jump to the undefined instruction exception (UIE) handler, where the payload resides. At one point, the boot ROM clears and reinitializes the UIE vector. The fault must thus be injected before this point in time.

First of all, some irregularities occur in the register dumps. The dump would normally show a link register (`lr`, `r14`) pointing to an instruction that gets executed *before* the clearing of the UIE vector. Similarly, the mode field of `cpsr`

would indicate the CPU to be in undefined instruction mode (`0x1b`) when running the payload. However, `lr` sometimes points to code running *after* the UIE vector clear occurs. Similarly, the `cpsr` mode field is equal to `0x1f` (system mode) most of the time, rather than `0x1b`.

Secondly, the fault timing seems to have a rather large window in which successes are observed. More specifically, the window starts with a high peak in the success rate, after which a long trail can be seen. This is shown in Figure 3.

A third interesting pattern emerges in the influence of the pulse injector coil voltage (and thus H-field strength) on the success rate. As one would expect (c.f. [25]), a threshold exists below which no faults will occur. However, when increasing the voltage even further, the success rate seems to *decrease*.

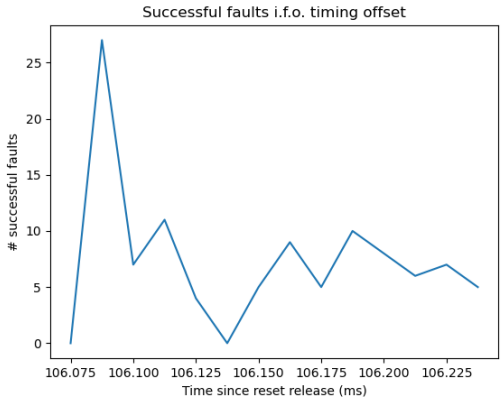


Fig. 3: Graph depicting the number of successful faults when attacking the ARM7 during boot ROM execution, in function of the time offset of the injected pulse. It starts out with zero successes, peaks slightly before 106.1 ms, after which it becomes much smaller again. Every possible moment was attempted 1400 times, the peak thus corresponds to a success rate of about 2%.

5.3 Explanation

From the `cpsr` information, it is clear that the real fault mechanism is *not* causing an UIE. Instead, a direct program counter corruption seems more likely. As the rest of SRAM is filled with NOP sleds that jump to the payload, it is not unlikely that a `pc` corruption would end up there.

The 'long tail' of the success rate i.f.o. the fault timing confirms this. The boot ROM clears SRAM upon starting up. The later the fault is injected (and thus, the more SRAM has been cleared), the lower the chance of a `pc` corruption ending up in the NOP sled.

Furthermore, the authors of [21] provide a possible explanation pointing towards `pc` corruption as well: increasing the coil voltage ends up corrupting more bits in an instruction word. This is desirable when trying to effect a large change (needed for e.g. an UIE), but less so for smaller ones (e.g. changing the destination register into the program counter, while keeping the rest intact). If the latter is what is needed instead of the former to cause successful faults and take over the ARM7 CPU core, increasing the voltage is counterproductive, which is what we observed.

6 ARM9 ROM analysis

This section demonstrates how the security analysis of a boot ROM can be conducted, and what vulnerabilities lie in the ARM9 boot ROM. As this is the first work to reverse-engineer this ROM, we provide a functional description as well, as a reference for others.

6.1 Method

To properly analyze the functioning of the boot ROMs, a combination of multiple tools is needed. Ghidra[4] serves as a base for static analysis to discover possible vulnerabilities. These are then tested in small 'unit tests' using Unicorn[5] and Python scripting. However, this environment does not suffice to emulate the full boot procedure with both ARM cores active at the same time. To overcome this, we extended the Nintendo DS emulator melonDS [6] to support booting from the boot ROMs and enable debugging using GDB [7]. This way, an exploit can be tested in the full boot process, with instruction stepping and memory inspection.

6.2 Functional description

The boot ROMs load, decrypt and verify the second-stage bootloader as follows: first, the ARM7 reads configuration bytes from an external SPI flash. Depending on this configuration, it will boot from either eMMC or the SPI flash itself. If a special button combination is pressed, the game cartridge will be booted from instead, just like the 3DS [31]. Then, a 512-byte boot header is read from the boot medium. This header contains information on the offset, size, load address, SRAM mapping configuration, and optional compression flags of the second-stage payload binaries, as well as an RSA-1024 signature. The ARM7 sends this boot header to the ARM9 over the FIFO interface, and the ARM9 then verifies the RSA signature, sending the result back to the ARM7.

The RSA signature format is rather peculiar: instead of using PKCS#1, a custom format is used. The RSA signature appendix contains the hashes of the boot header and of the decrypted second-stage binaries, a partial AES key, and

[4] `https://ghidra-sre.org/`
[5] `https://www.unicorn-engine.org/`
[6] `https://melonds.kuribo64.net/`
[7] Available at `https://github.com/melonDS-emu/melonDS/pull/1583`

a hash of all the previous items concatenated. The RSA public key resides in the ARM9 ROM. PKCS#1 v1-style padding is used, but without ASN.1 encoding. A diagram of the boot header and signature formats is shown in Fig. 4.

The payload binaries are encrypted using AES-128-CTR. The key is derived from two 128-bit partial keys (`keyX` and `keyY`). `keyX` is hardcoded in the boot ROM, `keyY` comes from the RSA signature appendix as described above. The IV is 96 bits in size, and consists of the corresponding binary's size repeated three times (the first time as-is, the second time its binary complement, the third time its two's complement).

Using this construction, the RSA public key (stored in the protected half of the ARM9 boot ROM) is needed to obtain the AES key for decrypting the second bootstage. Nintendo leaked the RSA public key, `keyX` and `keyY` by having the 3DS be less secure than its predecessor, as described in 2.3. Furthermore, with any system update, Nintendo can also change this `keyY` in the RSA signature appendix, rendering previous leaks useless as long as the public key remains secret.

The ARM7 then sets up its AES accelerator peripheral and DMA engine to read and decrypt the first payload (i.e. the ARM7 binary of the second-stage bootloader). The plaintext is then sent to the ARM9, which then calculates its SHA-1 hash and compares it against the corresponding one in the RSA signature appendix. The procedure is repeated for the second payload (the ARM9 binary). Once the two hashes have been checked, the ARM9 communicates back the result to the ARM7. If everything verified correctly, both ROMs then jump to their respective second-stage binaries.

Both second-stage payloads can optionally be compressed using an LZ variant, this can be configured separately per payload binary. Normally, the payload is sent by writing it to a specific SRAM bank that can be mapped privately between both CPU cores. However, when this compression option is used, another option becomes available. It allows the data to be sent over the FIFO, and the ARM9 then decompresses it upon reception (instead of the ARM7 after reading from its AES accelerator). This new option applies to both binaries at once. The compression options are ignored when booting from a game cartridge, such payloads are always uncompressed. This implementation detail will become relevant in section 7.

6.3 Vulnerabilities

The cryptography of the ROMs seems rather interesting, with its custom signature format and use of primitives that have now become outdated [3, 22]. However, no straightforward way of forging such a signature appears possible, without either factoring the modulus or creating a second preimage of a SHA-1 hash. Only *collision* attacks against SHA-1 have been demonstrated in practice [36]. The method of deriving the AES IV could be problematic: if both binaries are equal in size, the CTR keystream would be reused, making decryption much easier. Though, this does not occur in practice. Secondly, the verification code never confirms whether the padding and data block together span the entire

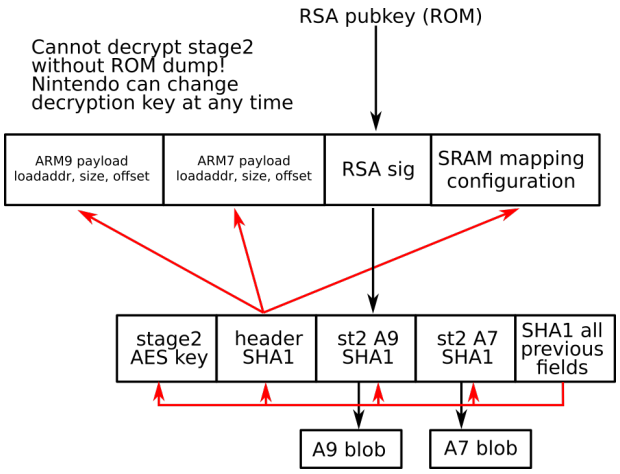


Fig. 4: Header format of the second bootstage and verification chain of the boot ROM: the RSA signature's message contains several hashes, not only for the code of the next bootstage, but also of the header information, and for the other information in the RSA message. Next to these hashes it also contains the partial AES used to decrypt the next bootstage.

128-byte RSA message. This again does not spell doom of the scheme, as the data block itself is 116 bytes in size, and the padding must be at least 8 bytes. This leaves 32 bits of data inside the RSA message that can be ignored. This on its own is too little to give an adversary any significant practical advantage to forge a signature.

The ROM software does not seem to include any 'obvious' vulnerabilities, as (unlike the 3DS ROM) it performs no parsing. The boot header format is a C struct with fixed offsets and sizes, and no ASN.1 encoding is used in the RSA signatures. There seem to be a few oversights, though on their own they do not compromise the security of the system. The first one is that the return value of the function checking the RSA signature padding is ignored, as shown in Figure 5. Improper padding checks were exploited in the 3DS [31] by brute-forcing an RSA signature with a corresponding appendix that triggers the bug. However, the 'hash of hashes' used here (which is not present in the 3DS) makes this computationally infeasible. The second oversight is that the payload binary sizes and load addresses are never sanity-checked. This was also exploited for the 3DS in a later stage of the jailbreak exploit [31]. The validity of these load addresses rely entirely on the authenticity of the RSA signature.

One particular spot in the ARM9 ROM code seems to be particularly vulnerable to fault injection. Right after the RSA signature verification, the ROM checks the 'hash of hashes' and the hash of the boot header. The corresponding assembly code is constructed such that, upon hash mismatch, a single 'load-bearing' `mov` instruction changes the return value in both cases, as shown in

```
NANDboot_RSA_verify(&rsa_verif_outbuf,boothdr_NAND);
// ^ return value not checked! rsa_verif_out in .bss
if (!NANDboot_check_boothdr_sha1_hashes(&rsa_verif_out,boothdr_NAND)) {
  ipc_notifyID(0xf);
  return -2;
}
// continue...
```

(a) Part of the `NANDboot_do_verify_header` ARM9 boot ROM routine, which verifies the RSA signature and the hashes inside the RSA message.

```
bool swi_RSA_Decrypt_Unpad(RSA_heap *heap,byte *dest,byte *src,byte *key)
{ // snip: local variable declarations
  memset(dest_with_pad,0,0x80);
  memset(unpad_output,0,0x80);
  dest_ptrnfo.dst = (byte *)dest_with_pad;
  dest_ptrnfo.src = src;
  dest_ptrnfo.key = key;
  if (swi_RSA_Decrypt(heap,&dest_ptrnfo,&lenout) &&
      RSA_parse_padding((byte *)unpad_output,&len_unpad_out, (byte *)
          ↪ dest_with_pad,lenout, 0x80)) {
    memcpy(dest,unpad_output,len_unpad_out);
    // if bad padding: not copied, dest is zero-initialized
    return true;
  }
  return false;
}
```

(b) `swi_RSA_Decrypt_Unpad` routine, called by `NANDboot_RSA_verify`

Fig. 5: Code snippets of the ARM9 boot ROM showing how the ROM code forgets to check whether the RSA signature had well-formed padding. If this is not the case, it ends up using zero-initialized memory as the signature appendix.

Figure 6. If this `mov` instruction were to be faulted, *both* hash checks can be skipped. This however still leaves the second-stage payload binary hashes in place.

```
bool NANDboot_check_boothdr_sha1_hashes(RSA_appendix *sig,NAND_boothdr *)
{ // snip...
  result = true;
  if (!swi_SHA1_Compare(sig->SHA1_boothdr,boothdr_digest)
      || !swi_SHA1_Compare(sig->SHA1_hash_of_hash,all_flds_digest)) {
    result = false; // skip this -> bypass both checks
  }
  return result;
}
```

Fig. 6: Code snippets of the ARM9 boot ROM showing how the ROM's checks for both SHA1 hashes in the RSA message end up in the same code path. This makes it such that only one instruction needs to be skipped in order to bypass both hash checks. `NANDboot_check_boothdr_sha1_hashes` is called by the code in Figure 5a.

7 Practical exploitation

With the elements from the previous sections, we can now devise an exploitation method. Once this exploit has been tested, we can design a cheap modchip that performs the attack. We then select suitable crowbar MOSFETs and perform another parameter search. After this is done, we insert the modchip into the DSi and evaluate the performance of this modchip.

7.1 Method

This attack uses a single injected fault to take control over both ARM cores. When the attack succeeds, both boot ROMs are still mapped into the address space. It works as follows:

1. Use fault injection to bypass both the boot header hash check and the 'hash of hashes' check at once.
2. Set the load address of the second-stage ARM7 binary (which gets loaded first) to an ARM9 stack address (in its DTCM, Data Tightly Coupled Memory). Let this binary be LZ-compressed, with the FIFO option enabled. This will make the ARM7 send the decrypted, compressed binary to the ARM9 over the FIFO interface. The ARM9 then decompresses it and writes it to the destination address in its own address space (i.e. the DTCM). This allows an attacker to control return addresses (and thus the program counter)

of the ARM9. This all happens before the ARM9 has a chance to calculate the hash of the decrypted and decompressed second-stage binary.
3. DTCM is marked as no-execute by the MPU. Thus, use ROP [33] to copy the payload to ITCM (Instruction Tightly Coupled Memory, which is read-write-execute), and jump to it. The attacker now fully controls the ARM9.
4. The 'ARM7' payload (running on the ARM9) uses the FIFO interface to instruct the ARM7 to now load the next second-stage binary (meant for the ARM9).
5. Set the load address of the second-stage ARM9 binary to an ARM7 stack address, and do not use compression for this binary.
6. The ARM7 will load the decrypted ARM9 binary into its address space. It intends to send it to the ARM9 by remapping the SRAM bank backing it. Instead, it will take over control of the ARM7. The ARM7 stack is executable, thus no further steps need to be taken. The attacker is now in control of both cores, with the boot ROMs still mapped into memory.

This attack has been tested to work in melonDS and with the EMFI setup. The following sections deal with how a modchip can be built using this attack.

7.2 Design

Before a PCB can be designed and firmware can be written, several questions still need to be answered.

The first is the choice of the boot medium. A custom game cartridge — akin to a flashcart — would be ideal in terms of practicality, but this is not possible: compression cannot be used with a game cartridge, while the exploit relies on its use. The eMMC memory is also not a possibility: modifying its contents in-situ is difficult (it requires BGA rework skills). The final option is thus to use the SPI flash as boot medium. Luckily, this IC resides on a daughterboard instead of being directly soldered onto the motherboard. Replacing the daughterboard with one with malicious contents is thus relatively easy.

This replacement daughterboard can thus serve as the modchip. For practicality reasons, voltage glitching will have to be the fault injection mechanism, EMFI or LFI modchips have never been made, as far as the authors know. This can be done by using a crowbar MOSFET [19] on the 1.2V SoC core power rail. As the Raspberry Pico was used in the ROM extraction setup, reusing the RP2040 here is also a straightforward choice. Several other modchips use this microcontroller as well [40, 39].

7.3 Evaluation

The modchip is capable of injecting faults successfully to take over both CPU cores of the target. Though, the success rate is low enough that success occurs roughly once every ten minutes. A photo of the modchip can be seen in Figure 7.

The success rate could be increased by experimenting more with the setup, such as using more different MOSFETs, using MOSFET drivers, activating multiple MOSFETs at once, and so on. Nintendo Switch modchips [39] use two

IRFHS8342 MOSFETs, for example. These are also positioned very close to the supply rails, while here, it is placed on the modchip PCB.

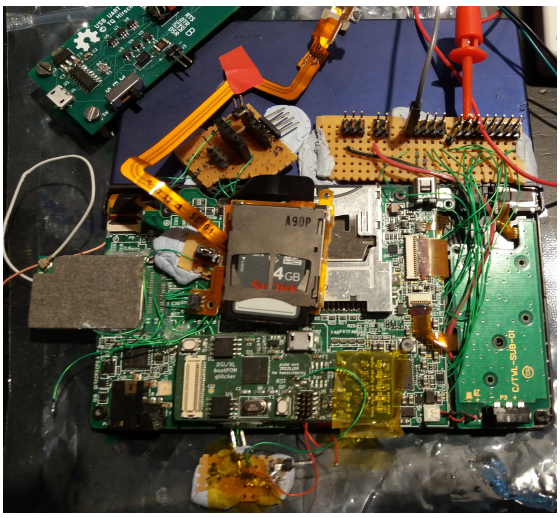


Fig. 7: A photo of the modchip installed on the target DSi. The modchip is placed on top of the motherboard, in the front of the photo. The beige connector on the modchip could be used to mount the WiFi board on top of the modchip. Though, doing so would make it impossible to close the plastic shell of the console. The loose SD card connector and breadboards are remnants from the boot ROM extraction process. In this photo, the crowbar MOSFET has been placed on a separate small breadboard as a workaround for signal integrity issues.

8 Conclusion

The DSi used a much more 'ad-hoc' and 'home-built' security system compared to the 3DS, with its custom signature format and lack of operating system being able to enforce security boundaries. Nevertheless, it ended up being more difficult to actually break, from finding an entrypoint to extracting and exploiting the boot ROMs. This stands in contrast to the fact that, especially with its use of outdated primitives, it looks much weaker on paper.

Though, this does not mean using the DSi's security system in a different context is a good idea: it is very specialized towards gaming, its main concern is piracy. The security system depends on custom hardware features not present in typical SoCs to achieve its security objectives. For example, the use of two different CPUs with different memories working in tandem on the same task, and the `SCFG` registers able to disallow games to access certain hardware. Getting one of these elements slightly wrong would have had bad consequences. For new

designs, using trusted execution environments e.g. ARM TrustZone seems like a better idea. These systems are much better understood and provide less room for mistakes. Similarly, the boot ROM was hard to attack simply because it has little attack surface. More modern devices typically implement a USB-based 'recovery mode' in their boot ROM, with all the consequences stemming from implementing this complex protocol (see e.g. [11]).

Secondly, this work shows that second-order fault injection attacks on complex SoCs are feasible without too much trouble. Such attacks should thus also be part of the attacker model (when applicable), instead of dismissing them as 'unfeasible' or 'impractical'.

Thirdly, countermeasures against such attacks are difficult to implement. As a rather complex fault model is used to extract the ROM image, it cannot easily be mitigated in software. Furthermore, as this work has shown second-order attacks to be feasible, such countermeasures against simple models (such as instruction skips) can also still be defeated. Similarly, ASLR, the typical countermeasure against ROP, would be very difficult to implement in the context of a boot ROM. Implementing code relocations would add more attack surface than it would remove. Instead, countermeasures at a lower level need to be used: fault detectors can detect FI attacks, builtin redundancy and error correction can stop them, and microarchitectural security techniques such as CFI and pointer authentication can stop ROP attacks as well. Finally, though, much of this work was only possible thanks to [17], which only happened because the 3DS leaked information about the security system of the DSi. Vendors leaking hardware secrets is very difficult to defend against.

Overall, this work shows how to resurrect consoles with a broken eMMC chip by breaking the security system. This was needed because normally, due to the secure boot implementation, the console would not be able to boot with such a broken component.

While in this setting, the user can still be seen as an attacker, it is not necessarily a *bad thing* that the user can perform such attacks. This question is relevant in the context of IoT security, where users may want to use such attacks to disable features that harm their privacy or safety [29]. This is already demonstrated by their reluctance to even let such devices be connected to the Internet [27].

An archive of the source code of the ROM extraction setup, modchip firmware and exploit payload, can be found at
`https://gitlab.ulyssis.org/pcy/dsi-hacking-stuff`.

Acknowledgments

We would like to thank Arthur Beckers for his practical help with setting up and conducting the fault injection run against the ARM7 boot ROM.

This work was supported by CyberSecurity Research Flanders with reference number VR20192203 and by the Research Council KU Leuven C1 on Security and Privacy for Cyber-Physical Systems and the Internet of Things

with contract number C16/15/058. In addition this work was supported by the European Commission through the Horizon 2020 research and innovation program under grant agreement Belfort ERC Advanced Grant 101020005 695305, through H2020 Twinning SAFEST 952252, through the Horizon Europe research and innovation program under grant agreement HORIZON-CL3-2021-CS-01-02 101070008 ORSHIN.



References

1. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The Sorcerer's Apprentice Guide to Fault Attacks, Cryptology ePrint Archive, Paper 2004/100 (2004). https://eprint.iacr.org/2004/100.
2. Barenghi, A., Breveglieri, L., Koren, I., Naccache, D.: Fault Injection Attacks on Cryptographic Devices: Theory, Practice, and Countermeasures. Proceedings of the IEEE **100**(11), 3056–3076 (2012). https://doi.org/10.1109/JPROC.2012.2188769
3. Barker, E., Dang, Q.: NIST Special Publication 800-57 Part 3 Revision 1: Recommendation for Key Management: Application-Specific Key Management Guidance. National Institute of Standards and Technology (2015). https://doi.org/10.6028/NIST.SP.800-57pt3r1. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57Pt3r1.pdf
4. Blömer, J., Silva, R.G.d., Günther, P., Krämer, J., Seifert, J.-P.: A Practical Second-Order Fault Attack against a Real-World Pairing Implementation. In: 2014 Workshop on Fault Diagnosis and Tolerance in Cryptography, pp. 123–136 (2014). https://doi.org/10.1109/FDTC.2014.22
5. Copetti, R.: Nintendo 3DS Architecture - A Practical Analysis, Archived at https://archive.li/cGFtC. (2023). https://www.copetti.org/writings/consoles/nintendo-3ds/#anti-piracy-and-homebrew (visited on 09/28/2023)
6. Copetti, R.: Nintendo DS Architecture - A Practical Analysis, Archived at https://archive.ph/28Jmb. (2020). https://www.copetti.org/writings/consoles/nintendo-ds/%5C#security-mechanisms (visited on 03/27/2022)
7. dark samus, Worklog - Getting the DSi bootroms — BitBuilt, Online forum (2017). https://bitbuilt.net/forums/index.php?threads/.948/ (visited on 03/27/2022). Archived at https://archive.ph/AvDsQ.
8. derrek, nedwill, naehrwert, Nintendo Hacking 2016: Game Over. In: 33rd Chaos Communications Congress: 'Works for me' (2016). https://media.ccc.de/v/33c3-8344-nintendo_hacking_2016 (visited on 03/27/2022)
9. Dusart, P., Letourneux, G., Vivolo, O.: Differential Fault Analysis on A.E.S. Cryptology ePrint Archive, Paper 2003/010 (2003). https://eprint.iacr.org/2003/010.
10. fail0verflow, PS4 Aux Hax 2: Syscon, Archived at https://archive.ph/mt3YK. (2018). https://fail0verflow.com/blog/2018/ps4-syscon/ (visited on 10/09/2022)

11. Galauner, A., Bazanski, S.: Glitching the Switch. In: OpenChaos 2018 (2018). https://media.ccc.de/v/c4.openchaos.2018.06.glitching-the-switch (visited on 10/05/2022)
12. Gerlinsky, C.: Breaking Code Read Protection on the NXP LPC-family Microcontrollers. In: REcon 2017 Brussels Hacking Conference (2017). https://doi.org/10.5446/32392
13. den Herrewegen, J.V., Oswald, D., Garcia, F.D., Temeiza, Q.: Fill your Boots: Enhanced Embedded Bootloader Exploits via Fault Injection and Binary Analysis. IACR Transactions on Cryptographic Hardware and Embedded Systems **2021, Issue 1**, 56–81 (2020). https://doi.org/10.46586/tches.v2021.i1.56-81. https://tches.iacr.org/index.php/TCHES/article/view/8727
14. Korth, M.: GBATEK, Archived at https://archive.ph/Ws1cO. (2021). https://problemkaputt.de/gbatek.htm (visited on 10/04/2022)
15. Korth, M.: GBATEK DSi Control Registers (SCFG), Archived at https://archive.ph/rPwKB. (2021). https://problemkaputt.de/gbatek-dsi-control-registers-scfg.htm (visited on 10/04/2022)
16. Korth, M.: GBATEK DSi SD/MMC Internal NAND layout, See 'boot info blocks'. Archived at https://archive.li/I7S9E. (2021). https://problemkaputt.de/gbatek-dsi-sd-mmc-internal-nand-layout.htm (visited on 03/27/2022)
17. Korth, M.: Unlaunch, Archived at https://archive.ph/g5Qv0. (2018). https://problemkaputt.de/unlaunch.htm (visited on 03/27/2022)
18. Lu, Y.: Attacking Hardware AES with DFA, (2019). https://doi.org/10.48550/ARXIV.1902.08693. https://arxiv.org/abs/1902.08693. Supplementary text available at https://yifan.lu/2019/02/22/attacking-hardware-aes-with-dfa/. Accessed on 2022-10-06. Archived at https://archive.ph/oQlE7.
19. Lu, Y.: Injecting Software Vulnerabilities with Voltage Glitching, (2019). https://doi.org/10.48550/ARXIV.1903.08102. https://arxiv.org/abs/1903.08102.
20. McClintic, M., Maloney, D., Scires, M., Marcano, G., Norman, M., Wright, A.: Keyshuffling Attack for Persistent Early Code Execution in the Nintendo 3DS Secure Bootchain, (2018). https://doi.org/10.48550/ARXIV.1802.00092. https://arxiv.org/abs/1802.00092.
21. Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic Fault Injection: Towards a Fault Model on a 32-bit Microcontroller. (2013). https://doi.org/10.1109/FDTC.2013.9. http://arxiv.org/abs/1402.6421
22. National Institute of Standards and Technology, NIST Retires SHA-1 Cryptographic Algorithm, Archived at https://archive.ph/zUJQk. (2022). https://www.nist.gov/news-events/news/2022/12/nist-retires-sha-1-cryptographic-algorithm (visited on 04/27/2023)
23. nocash, ApacheThunder, dark samus, Get BOOTROM/Key Scrambler? - 4dsdev, Online forum (2016). https://4dsdev.kuribo64.net/thread.php?id=130 (visited on 03/27/2022). Archived at https://archive.ph/qdu9x.
24. oranav, eMMC hacking, or: how I fixed long-dead Galaxy S3 phones. In: 34th Chaos Communications Congress: 'tuwat' (2017). https://media.ccc.de/v/34c3-8784-emmc_hacking_or_how_i_fixed_long-dead_galaxy_s3_phones (visited on 02/24/2023)
25. Ordas, S., Guillaume-Sage, L., Maurine, P.: Electromagnetic fault injection: the curse of flip-flops. Journal of Cryptographic Engineering **7**(3), 183–197 (2017). https://doi.org/10.1007/s13389-016-0128-3. https://hal-lirmm.ccsd.cnrs.fr/lirmm-01430913

26. plutoo, derrek, smea, Console Hacking: Breaking the 3DS. In: 32nd Chaos Communications Congress: 'Gated Communities' (2015). https://media.ccc.de/v/32c3-7240-console_hacking (visited on 10/04/2022)
27. Purdy, K.: Appliance makers sad that 50% of customers won't connect smart appliances. Ars Technica (2023). https://arstechnica.com/gadgets/2023/01/half-of-smart-appliances-remain-disconnected-from-internet-makers-lament/ (visited on 05/17/2023)
28. Raelize, Espressif ESP32: Controlling PC during Secure Boot, Archived at https://archive.li/6vEgT. (2020). https://raelize.com/blog/espressif-systems-esp32-controlling-pc-during-sb/ (visited on 10/09/2022)
29. Ren, J., Dubois, D.J., Choffnes, D., Mandalari, A.M., Kolcun, R., Haddadi, H.: Information Exposure From Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach. In: Proceedings of the Internet Measurement Conference. IMC '19, pp. 267–279. Association for Computing Machinery, Amsterdam, Netherlands (2019). https://doi.org/10.1145/3355369.3355577
30. Scire, M., Mears, M., Maloney, D., Norman, M., Tux, S., Monroe, P.: Attacking the Nintendo 3DS Boot ROMs, (2018). https://arxiv.org/abs/1802.00359. https://doi.org/10.48550/ARXIV.1802.00359
31. SciresM, Myria, Normmatt, TuxSH, Hedgeberg, Sighax and Boot9strap, Archived at https://web.archive.org/web/20211105063611/https://sciresm.github.io/33-and-a-half-c3/. (2017). https://sciresm.github.io/33-and-a-half-c3/ (visited on 03/27/2022)
32. Scott, M.E.: DSi RAM tracing, Archived at https://archive.ph/lhMYa. (2009). https://scanlime.org/2009/09/dsi-ram-tracing/ (visited on 06/27/2023)
33. Shacham, H., Buchanan, E., Roemer, R., Savage, S.: Return-Oriented Programming: Exploits Without Code Injection. In: Black Hat USA 2008 Briefings (2008)
34. Shepherd, C., Markantonakis, K., van Heijningen, N., Aboulkassimi, D., Gaine, C., Heckmann, T., Naccache, D.: Physical fault injection and side-channel attacks on mobile devices: A comprehensive analysis. Computers & Security **111**, 102471 (2021). https://doi.org/10.1016/j.cose.2021.102471
35. Sidorenko, A., van den Berg, J., Foekema, R., Grashuis, M., de Vos, J.: Bellcore attack in practice, Cryptology ePrint Archive, Paper 2012/553 (2012). https://eprint.iacr.org/2012/553.
36. Stevens, M., Bursztein, E., Karpman, P., Albertini, A., Markov, Y.: The First Collision for Full SHA-1. In: Katz, J., Shacham, H. (eds.) Advances in Cryptology – CRYPTO 2017, pp. 570–596. Springer International Publishing, Cham (2017)
37. Timmers, N., Mune, C.: Escalating Privileges in Linux Using Voltage Fault Injection. 2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC) (2017)
38. Whelan, C., Scott, M.: The Importance of the Final Exponentiation in Pairings When Considering Fault Attacks. In: Pairing-Based Cryptography (2007)
39. Wololo, Picofly: The $3 Nintendo Switch hacking modchip is real, and it's now available, Archived at https://archive.li/Puo2C. (2023). https://wololo.net/2023/03/21/picofly-the-3-nintendo-switch-hacking-modchip-is-real-and-its-now-available/ (visited on 04/17/2023)
40. Wouters, L.: Glitched on Earth by Humans: A Black-Box Security Evaluation of the SpaceX Starlink User Terminal. In: DEF CON 2022 (2022)
41. Yuce, B., Schaumont, P., Witteman, M.: Fault Attacks on Secure Embedded Software: Threats, Design and Evaluation. CoRR **abs/2003.10513** (2020). arXiv: 2003.10513. https://arxiv.org/abs/2003.10513