Subverting Linux' Integrity Measurement Architecture

Felix Bohling
6bohling@informatik.uni-hamburg.de
University of Hamburg, Germany

Tobias Mueller*
mueller@informatik.uni-hamburg.de
University of Hamburg, Germany

Michael Eckel
michael.eckel@sit.fraunhofer.de
Fraunhofer SIT
Darmstadt, Germany

Jens Lindemann
lindemann@informatik.uni-hamburg.de
University of Hamburg, Germany

ABSTRACT

Integrity is a key protection objective in the context of system security. This holds for both hardware and software. Since hardware cannot be changed after its manufacturing process, the manufacturer must be trusted to build it properly. However, it is completely different with software. Users of a computer system are free to run arbitrary software on it and even modify BIOS/UEFI, bootloader, or Operating System (OS).

Ensuring that only authentic software is loaded on a machine requires additional measures to be in place. Trusted Computing technology can be employed to protect the integrity of system software by leveraging a Trusted Platform Module (TPM). Measured Boot uses the TPM to record measurements of all boot software in a tamper-resistant manner. Remote attestation then allows a third party to investigate these TPM-protected measurements at a later point and verify whether only authentic software was loaded.

Measured Boot ends with loading and running the OS kernel. The Linux Integrity Measurement Architecture (IMA) extends the principle of Measured Boot into the OS, recording all software executions and files read into the TPM. Hence, IMA constitutes an essential part of the Trusted Computing Base (TCB).

In this paper, we demonstrate that the security guarantees of IMA can be undermined by means of a malicious block device. We validate the viability of the attack with an implementation of a specially-crafted malicious block device in QEMU, which delivers different data depending on whether the block has already been accessed. We analyse and discuss how the attack affects certain use cases of IMA and discuss potential mitigations.

CCS CONCEPTS

• **Security and privacy** → *Operating systems security*; Hardware attacks and countermeasures; Trusted computing.

KEYWORDS

System security, integrity measurement architecture, side-channel attack, Trusted Computing

ACM Reference Format:
Felix Bohling, Tobias Mueller, Michael Eckel, and Jens Lindemann. 2020. Subverting Linux' Integrity Measurement Architecture. In *The 15th International Conference on Availability, Reliability and Security (ARES 2020), August 25–28, 2020, Virtual Event, Ireland.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3407023.3407058

*Corresponding

ARES 2020, August 25–28, 2020, Virtual Event, Ireland

ACM ISBN 978-1-4503-8833-7/20/08…$15.00
https://doi.org/10.1145/3407023.3407058

1 INTRODUCTION

Software integrity is a key protection goal in computer system security. If an attacker manages to tamper with application software, the resulting computations cannot be trusted. This argument extends to the OS, firmware, and finally the CPU. While protecting the integrity of applications running on a trusted OS is feasible, protecting firmware of a motherboard or CPU is much harder.

A lot of research has gone into the subject of ensuring the integrity of a whole computer system, e. g. by making use of a Trusted Platform Module (TPM) for Measured Boot and Remote Attestation [24, 42, 51]. With Measured Boot the system measures all boot components, e.g. the BIOS, the bootloader and the OS kernel. As the measurements are anchored in the TPM, a third party can verify them against expected good measurements (Remote Attestation). The result of this verification can then be used for security relevant decisions, e. g. whether a passphrase for disk encryption can be handed out or whether a machine shall be allowed to access the network in a Network Admission Control (NAC) system [26].

Secure Boot, in contrast to Measured Boot, verifies digital signatures over software components in place, before passing control to them. In case a signature is not valid, the system stops booting.

IMA is part of the Linux Security Modules (LSM) subsystem and extends the principles of Measured Boot and Secure Boot into the Linux OS. For this purpose IMA measures executables before they are loaded into memory for execution. IMA can be configured to also measure other files upon access.

It is worth noting that IMA does not prevent all possible ways of introducing malicious code into the Linux OS on its own. For example, extended Berkeley Packet Filter (eBPF) program loaded into the kernel from user-space as well as simple file-like objects such as `stdin` are known to be so called "measurement gaps".

In this paper we present our findings of an inherent Time of Check, Time of Use (TOCTOU) vulnerability in Linux' IMA [11]. Instead of just finding another measurement gap, our attack exploits

a systemic weakness of IMA. With our attack an attacker is able to undermine the integrity of a target system by having a malicious file executed, despite the IMA measuring the correct value. We demonstrate the feasibility of the attack and evaluate chances of successfully launching it.

The problem exists because IMA requires files and executables to be read and hashed as a whole, before they are executed or read. However, if the OS cannot cache the entire file read by IMA, parts of it are read off the disk again by the OS loader; this time without involving IMA. Similarly, IMA only re-hashes a file if it thinks it has changed. In case a program is executed multiple times consecutively and is not present in the page cache, it will be read off the disk for execution without being hashed again by IMA.

In this paper, we exploit this systemic weakness to make the OS, and in fact any interested third party, believe that the machine has been executing benign code, while we actually caused the machine to execute our malicious code. Consequences and their severity vary with the way IMA is used. However, it is conceivable that certain industries, especially in the critical infrastructure domain, rely on IMA to assert the integrity of their systems [20, 39, 46, 52].

With our attack, a motivated attacker is able to stealthily compromise computing systems. In the domain of critical infrastructures, misbehaviour can have severe impact on the environment, infrastructure, or even humans. Considering a nuclear power plant which encounters undetected serious malfunction caused by malicious software, consequences can be disastrous.

With this paper, we make the following contributions:

- We review mechanisms and protocols to assert the integrity of a computer system,
- we analyse an inherent weakness in the implementation of Linux' IMA, and how it affects certain use cases of IMA,
- we evaluate the attack surface as well as performance and feasibility measurements, and
- discuss potential mitigations for the problem.

2 BACKGROUND

A trusted system boot process is a precondition for a trusted operating system [7]. The Trusted Computing Group distinguishes between two methods by which a trusted system can be booted [47], i. e. Measured Boot and Secure Boot, both of which are described in more detail in the following. Further, we introduce remote attestation and IMA.

2.1 Measured Boot

Measured Boot, also referred to as Trusted Boot, is the process in which a system measures all boot components consecutively after the system is powered on [29]. Boot components are, e. g. BIOS, bootloader, or the OS kernel. Measurements are recorded in one or more event logs and represent hashes of software binaries and files, generated using a cryptographic hash function, such as SHA256. Along with the hash of a component, an identifier is recorded in the event log, e. g. "UEFI", "Option ROM", or "Bootloader", which enables the log to be auditable at a later point.

The measurement process is hardened by using a TPM to anchor event log entries, making them tamper-resistant against unintended manipulation. Anchoring, in this context, means that a measurement hash is extended to a protected location called Platform Configuration Register (PCR) inside the TPM. A PCR realises a folding hash function internally and allows only to extend it, i. e. it is not possible to delete the PCR or set it to an arbitrary value. As a result, an attacker cannot tamper with the event log undetected in order to cover traces of malicious actions.

The measurement process is implemented as follows: (1) execute main logic of component, (2) measure next component in the boot chain, (3) append measurement to the event log, (4) anchor log entry in the TPM, (5) activate next component. This is to ensure a component becomes active only after it has been measured and recorded in the TPM-protected event log.

There must be a very first component in the system which becomes active immediately after the system is powered on. This component is called Root of Trust for Measurement (RTM), and must be trusted implicitly. The RTM typically is immutable and often implemented in hardware.

2.2 Secure Boot

Secure Boot, also referred to as Verified Boot [3], was first described by Tygar and Yee [49] and is similar to Measured Boot. However, instead of only recording measurements of the next component in the boot chain, it verifies the hash in place by comparing it against an expected one, a reference measurement. Reference measurements can be embedded as digital signatures inside the boot component binary itself, and can be verified, e. g. using a key in the TPM. In case the expected hash does not match the current, the boot process is aborted, resulting in a system that is unable to boot [43].

Early implementations of Secure Boot, which also support *Authenticated Boot*, used specialised hardware and co-processors [19, 49]. Arbaugh et al. first practically applied Secure Boot to a PC platform with minor modifications to the BIOS and by adding a PROM chip. Secure Boot is supported on PCs since UEFI 2.3.1 [37]. As of today, different implementations of Secure Boot exist.

Secure Boot constructs a chain of integrity checks from power on, over the boot process, until control is finally transferred to the operating system [7]. According to Sailer et al. [43], Secure Boot is not practical for systems which from time to time need remote updates of security policies, without making the system vulnerable to denial of service attacks. Therefore, instead of simply aborting the boot process, *Measured Boot* [8] can be used to measure the boot process. Secure Boot and Measured Boot can be combined and operated in parallel.

2.3 Remote Attestation

Remote attestation is a process in which a system, the attestor, reports cryptographically signed evidence about all executed software and files read. Produced evidence is reported to an investigating system, the verifier, which in turn verifies the evidence. The result of this verification is a cryptographically signed claim about the integrity of the system.

Remote attestation requires a trustworthy Measured Boot process to be in place on the attested system. This is typically ensured by a system incorporating a RTM and a TPM in its hardware design, as well as TPM-aware boot components.

```
10 1d8d[..] ima-ng sha1:0000[..] boot_aggregate
10 5cfa[..] ima-ng sha1:a947[..] /init
10 290f[..] ima-ng sha1:5955[..] /bin/sh
10 0d41[..] ima-ng sha1:91bb[..] /etc/ld.so.cache
10 a5c1[..] ima-ng sha1:f598[..] /conf/arch.conf
10 2bf3[..] ima-ng sha1:c98a[..] /conf/initramfs.conf
10 e8d0[..] ima-ng sha1:9b32[..] /scripts/functions
```

Figure 1: (Shortened) IMA Stored Measurement Log (SML) of a booted Linux system, obtained via `/sys/kernel/security/ima/ascii_runtime_measurements` and using IMA template `ima-ng`

2.4 Integrity Measurement Architecture (IMA)

IMA extends the principles of Measured Boot and Secure Boot to the Linux OS, by applying it to applications and accessed files [43]. IMA has been part of the *Linux Integrity Subsystem* since 2009 (kernel 2.6.30) [55, 57] and enables protection of system integrity by *collecting* and *storing* measurements, and *appraising* them locally.

Further, IMA enables *remote attestation* and *protecting* measurements stored in extended attributes of files [42, 43, 55, 56]. For extending Secure Boot and Measured Boot to the operating system and its applications, IMA provides two mechanisms called Appraisal and Measurement.

2.4.1 IMA Appraisal extends the principle of *Secure Boot* into the OS and its applications. Known good hash values of each file are put into the extended file attributes. IMA uses these values to compare them against actual measurements of a file. In case of a measurement mismatch, IMA denies access to the file. Instead of only storing hash values in the extended file attributes, digital signatures can be used.

IMA appraisal can be run in four different modes [21, 56]: (1) `fix` collects and attaches the measurements as good hash values to the file's extended attributes., (2) `log` logs mismatches between the measured hash and the hash in the file's extended attributes, (3) `enforce` denies access in case of a mismatch, and (4) `off` disengages IMA appraisal.

2.4.2 IMA Measurement extends *Measured Boot* into the OS. It enables for Remote Attestation, where a third party challenges the system in order to verify its integrity [43]. For that purpose IMA maintains an ordered list of hash values in the kernel, the Stored Measurement Log (SML), for all files measured since the OS took over control.

Although IMA can be used without a TPM, it is usually used in conjunction with a TPM to provide a much stronger, hardware-rooted chain of trust. The SML can then be used in a remote attestation process in order to provide an investigating remote party with evidence of all software executions. The remote party can then verify the SML against a database with known good reference measurements, and check for deviations from the intended integrity state [43].

IMA provides the SML in both ASCII and binary representations in `/sys/kernel/security/ima/`: `ascii_runtime_measurements` and `binary_runtime_measurements` [56]. The format of the SML is defined by an IMA template (cf. Fig. 1). With the standard template `ima-ng`, the columns are, from left to right [21, 56]: (1) TPM PCR, (2) *template hash*, which is a hash computed over the file's path name, content hash, and other components which are defined by the template, (3) name of the used IMA template, (4) hash of the file content, and (5) file.

2.4.3 IMA Policies define which files to measure. IMA has three built-in policies which are passed as kernel boot arguments: (1) *tcb* measures kernel modules, executed software, files which are loaded into memory for execution with *mmap*, and files opened for reading by the root user. (2) *appraise_tcb* appraises in place, instead of only measures, the same components as the *tcb* policy. It denies access if files do not match their known good hash. (3) *secure_boot* appraises only the kernel, its modules, and the IMA policies.

Other concepts, such as "Trusted Cloud" [9, 10], TPM-based network endpoint assessment [44], Trusted Software Defined Networks [25], and integrity verification of Docker containers [17], build on IMA for detecting or protecting against manipulation of systems. IMA has also been applied to the critical infrastructure domain, e. g. for protecting sensors, actuators, and other controller and monitoring class devices [15, 20, 39, 52].

3 SUBVERTING IMA

In this section we describe our attack that overcomes IMA with a malicious block device. We first explain the underlying threat model which will be considered throughout the rest of the paper. Subsequently, we present our findings regarding the inherent weaknesses of Linux' current implementation of IMA.

3.1 Threat Model

For the purpose of our attack, we assume that an attacker has the ability to either insert a malicious block device into an IMA-protected system or modify the block device's firmware. Those manipulations could be performed at the manufacturing site, e. g. the manufacturer places an additional chip or a modified version of the chip for the customer's device. We note that such attacks have been documented [2, 13, 41]. Note that it is not necessary for an attacker to have full physical access to the target system – it is sufficient for them to be able to manipulate the block device itself.

The supply chain of a device constitutes another prominent target for attack. On its way from the manufacturer to the customer, the device can be intercepted and compromised hardware or software be implanted. This attack is known to be executed by the NSA through their "Tailored Access Operations" (TAO) [6, 23, 54].

Finally, the attack could be launched while the device is being maintained by a regular service operator, e. g. by replacing either the hard disk or its firmware with a malicious copy. Such a service operator could either be coerced into misbehaving or be infected with a malware which replaces other device's firmware, e.g. through BadUSB [35], without the service operator even noticing.

We assume that the attacker intends to modify an arbitrary file to either change a data value or manipulate the control flow of a program executed by the IMA-protected OS. In order to make the attack more severe, we concentrate on executable files and note that the attacker may either target the text section of the executable directly to manipulate the program code contained therein, or the

data section to manipulate the data processed by the program. We further assume that the program is either larger than the machine's memory or that the system is performing other tasks which causes the system to thrash.

The benign version of the file is assumed to have been obtained and stored by the victim before. As the manipulated version of the file has a different hash value than the benign version, the attacker needs to be able to expose the changed file in a way that IMA will not notice the changed hash value. Thus, even though appraisal is activated, the manipulated version of the file will be executed. For remote attestation, the hash value of the original version of the file is stored in the SML, while the manipulated version was in fact executed.

To this end, we consider two scenarios, in which the malicious block device provides the benign executable for the first read operation and the manipulated executable thereafter:

(1) The program is executed only once by the victim, e.g. during the boot process. In this scenario, it is conceivable that the program may finish its execution within a short period of time or that it may be executed continuously for a long time, e. g. as a daemon.
(2) The program is executed multiple times. Between the subsequent executions, some time passes and the machine is used for other purposes which cause the system to thrash.

We do not require our attacker to be able to change the boot parameters, e. g. manipulate the way the system boots. Further, we do not require the attacker to manipulate the TPM or any other hardware other than the hard disk or rather its firmware.

3.2 Structural Problems

A key insight into the vulnerability of IMA is that the check it performs is susceptible to a TOCTOU attack. Before Linux executes a file, it needs to compute its hash. If the file is large enough to not fit into the OS buffer cache, e. g. bigger than the machine's RAM, then the file will be read twice: First the file is read by IMA to compute its hash value. Then the file is read again for execution. For that second read, the file is not measured again by IMA. Hence, IMA will not notice if the block device provides a modified version of the file on the second read (cf. Fig. 2).

If the executable is not bigger than the caches, it will be in the page cache after IMA measured it. Hence, Linux does not issue a read request to the underlying block device, and thus, the block device sees only one read request from the OS rather than two. Hence, if the block device wanted to attack the host it would have to render the malicious executable on the first read. That, however, would be detected by IMA as it sees a change in the hash value. For optimisation reasons, however, IMA measures a file only if it thinks that it has not been hashed yet or that it has changed since the last read. This leads to the behaviour that if an executable has been evicted from the cache before it is executed again, IMA will not re-measure it, but assume the already measured value to be correct. Under these circumstances, the malicious block device can thus provide a modified version of the executable on subsequent reads.

We present an implementation of the attack in Sect. 3.3 and evaluate the conditions that make the attack possible in Sect. 4.

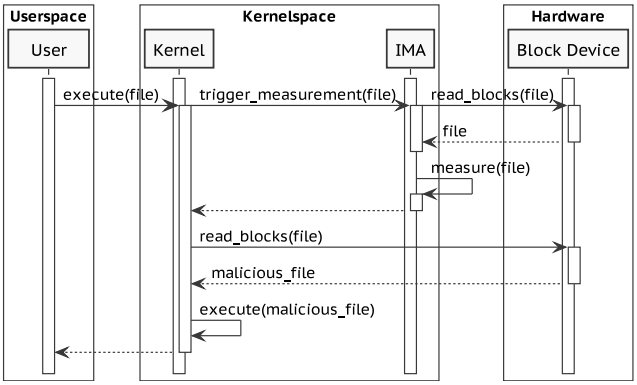

Figure 2: Exploiting the inherent TOCTOU bug in Linux' IMA implementation

3.3 Implementation

To simulate our attack, we need a malicious block device that can be used to trick IMA. To this end we modified the implementation of the virtual block devices in the QEMU hypervisor. Our implementation is based on version 4.1.0 of QEMU. We modified the (virtual) block device to (1) detect read access to security related information on the file-backed storage device, (2) return unmodified security-relevant information on the first read, and finally (3) return modified security-relevant information on subsequent reads.

The target of our modification was the `raw_co_preadv` function of QEMU. This function is a thin wrapper around the generic `raw_co_prw` function which implements logic for reading from a file-backed block device. Our intention is to intercept all `read` calls to the block device, and deliver the benign version of the file on the first read and a malicious version of the file, where a part of the file is manipulated.

Figure 3 shows the patched function used for the simulated attack. Originally, the function returns with the result of line 2, i.e. it is just thinly wrapping the other function. Thus, all further lines show our modifications. In line 4 the offset of the attacked file on the block device is defined. Line 5 defines where the part of the file starts that is manipulated during the attack. Line 6 defines where the part to be manipulated ends. The offsets in lines 4, 5 and 6 must be adjusted according to the target file and its position on the file system. By adding the previously mentioned offsets, the actual beginning and end of the manipulated area on the device is computed in lines 7 and 8. Line 9 defines a counter, that is used to count the number of reads that have happened. The `if` statement in lines 11 to 13 check whether the current read happens on our target block device and if the read is in the area that is to be attacked. If the read affects the part of the file that is to be attacked and the information has been read at least once before, the benign information will be replaced with the malicious information in lines 15 to 28. For this a buffer is created and filled with the value `0xFF` and copied into the io vector that is used by the hypervisor to read from the backing file of the simulated block device in lines 21 to 24. Before that the size and placement of the buffer within the vector is calculated in lines 15 to 19. When finally the end of the area to be manipulated has been read, the read counter in line 27 is incremented.

```
static int coroutine_fn
↪ raw_co_preadv(BlockDriverState *bs, uint64_t
↪ offset, uint64_t bytes, QEMUIOVector *qiov, int
↪ flags) {
    int retval = raw_co_prw(bs, offset, bytes, qiov,
    ↪ QEMU_AIO_READ);

    const uint64_t file_abs_pos = 0x001e2000;
    const uint64_t data_rel_bgn = 0x00001020;
    const uint64_t data_rel_end = data_rel_bgn +
    ↪ 0x00100000;
    const uint64_t data_abs_bgn = file_abs_pos +
    ↪ data_rel_bgn;
    const uint64_t data_abs_end = file_abs_pos +
    ↪ data_rel_end;
    static int dev_reads = 0;

    if (       offset <= data_abs_end
            && offset + bytes > data_abs_bgn
            && strstr(bs->filename, "fat") != NULL) {
        if (dev_reads > 0) {
            int cpy_bgn = data_abs_bgn - offset;
            if (cpy_bgn < 0) { cpy_bgn = 0; }
            int cpy_end = data_abs_end - offset;
            if (cpy_end > bytes) { cpy_end = bytes; }
            ptrdiff_t cpy_bytes = cpy_end - cpy_bgn;
            char evil_buf[cpy_bytes];
            for (int i = 0; i < cpy_bytes; i++) {
                evil_buf[i] = 0xFF;
            }
            qemu_iovec_from_buf(qiov, cpy_bgn,
            ↪ evil_buf, cpy_bytes);
        }
        if (offset + bytes > data_abs_end) {
            dev_reads += 1;
        }
    }
    return retval;
}
```

Figure 3: Patch for the QEMU hypervisor to return malicious data on subsequent reads.

We make our modifications to QEMU and all auxiliary scripts available under: https://github.com/muelli/subverting-ima.

4 EVALUATION

This section describes under which conditions our attack works.

In order to see whether our attack works we have created a scenario that resembles an embedded controller in the critical infrastructure domain. That is, we have a (virtual) machine that boots into Linux with IMA protection, i. e. the IMA policy set to `appraise_tcb` (cf. Sect. 2.4.3). The operating system then starts an executable during boot. This could be a program for controlling some industrial hardware that the system is connected to.

```
static char hello[] = "Hello, World!";
static int foo[1024 * 1024 * SIZE_IN_MB] = {0};
static char bye[] = "Good bye!";

int main() {
  printf("%s\n", hello);

  // Here we wait for the thrashing
  system("systemctl start stress-ng.service");

  for (int i = 0; i < sizeof (foo); i++) {
    if (foo[i] != 0) {
        printf("%i is wrong: %i\n", i, foo[i]);
    }
  }
  printf("%s\n", bye);
  return 0;
}
```

Figure 4: Crafted program in our critical infrastructure scenario that an attack is trying to tamper with (simplified).

We assume that the operator of the machine has set up Linux such that it makes use of IMA in order to protect against integrity compromising attacks. That means, in particular, that executable files have their hashes added as extended attributes and that Linux is booted in enforcement mode. Additionally, the hashes need to be signed and the key bound to the TPM so that it can neither be extracted nor replaced easily.

Our attacker attempts to make the operating system execute a file that has been tampered with. Our attacker has successfully launched their attack if neither the operating system nor an attestor notice any difference in the measured IMA values.

For evaluation, we set up a system that resembles this scenario. This setup consists of the already described QEMU hypervisor on x86-64 running a virtual machine of the same architecture with 512 MiB of RAM. On the host, we use an Ubuntu 18.04 with its default Linux 4.15 kernel. In the guest, we run an Ubuntu 20.04 with its default Linux 5.4 kernel. In line with our scenario, IMA is active. The main Linux system is booted from a normal, unmanipulated QEMU block device. Another block device is attached to the VM, which uses our modified, malicious version of the QEMU virtual block device (cf. Sect. 3.3). This block device is then used to launch an executable during boot, so that it can be used as the target of our attack. Instead of a complex program, we use a relatively simple one which iterates over an array and checks its contents, so that we can verify whether a manipulation took place (cf. a simplified version in Fig. 4). The size of the `foo` array in this code can be changed to produce executables of different sizes. The array can be placed either into the data or the text section at compile time. GCC places the array in the text section if it is marked with `__attribute__((section(".text#")))`. This allows us to test whether we can modify the respective section using the resulting executables.

When this program is executed without any attack taking place it is executed and running as expected. If the executable is replaced by

a manipulated version on a normal block device without adjusting the signed hash value, IMA will detect the manipulation and refuse execution of the executable, as the system log shows:

```
audit: type=1800 audit(...): ... op=appraise_data
↪  cause=invalid-hash comm=(check) name=check
↪  dev=vda1 ...
```

We first tested whether we could attack a relatively small version of our test program in our *first scenario*, as described in Sect. 3.1. We tuned the array size so that the executables containing it in either the text or data section were 1 MiB in size, each. After starting, the program immediately started checking the array. When these executables were started from the malicious block device, IMA did not notice any attack taking place. Both printed the same output as they would have printed when executed from a benign block device, i. e. we were not successful in manipulating the executable.

We then used a larger version of our test program. We produced an executable of 768 MiB in size, which exceeds the amount of available memory (512 MiB). Interestingly, we could make Linux execute ELF files with a *text* section bigger than the available memory, but not files with a *data* section bigger than the available memory. With a data section too big to fit into memory, Linux fails execution with a segmentation fault. With a big text section, IMA did not notice any attack taking place and allowed the executable to be run. However, the output did not match that of the program's benign version. When the array was placed in the text section, 824 123 392 of its 824 180 736 bytes were successfully manipulated by the attack, which corresponds to 201 202 out of 201 216 memory pages. In other words, 14 pages were held in memory while the rest was evicted. By running the test multiple times, we could observe that the first 13 pages as well as the last page which contained the array were always retained while the corresponding areas of the other pages on the block device were read twice.

While this attack allows us to modify nearly all pages of the text section of the executable section, it requires the benign executable to be larger than the available memory. It is questionable whether such a large executable which overloads the system's main memory, would actually be used by the victim.

Therefore, we modified our scenario to determine whether we can perform a similar attack on a smaller executable. In the modified scenario, we cannot rely on the executable being read once by IMA and then immediately again for its execution. However, parts of the executable may have been removed from memory over time if the memory is needed for other purposes. In case of a file-backed mapping whose contents have not been modified, the corresponding memory pages will not be written out to swap space, but rather be loaded again from the original file.

To evaluate whether an attack works in the modified scenario, we introduced a wait time into our program: After starting, the program now sleeps for two minutes. Only after this period, the program checks whether the array has been successfully manipulated. We varied the size of the executable to evaluate the impact of the size on the attackability. The file sizes tested were (in MiB): 1, 2, 4, 8, 16, 32, 64, 128, 256, and 384. For each file size, both executables containing the array within the text section as well as one containing it in the data section was compiled.

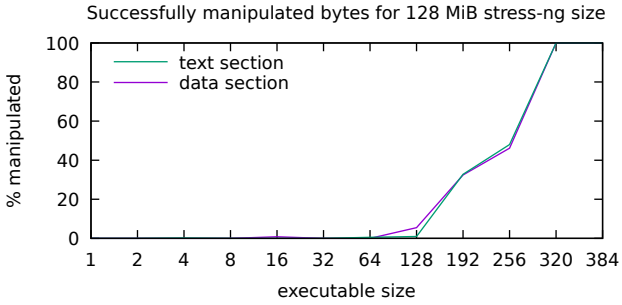

Figure 5: Diagram showing the percentage of successfully manipulated bytes for different executable sizes, and a fixed allocation size of the stress-ng benchmark of 128 MiB.

While the program was sleeping, we executed a stress-ng benchmark. stress-ng was configured to allocate 1 (32, 64, 128, 192, 256, 320, and 384) MiB of memory, by growing the heap in 1 MiB increments. In particular, the command we executed was `stress-ng --bigheap 1 --bigheap-ops memsize --bigheap-growth 1M --timeout 30s` with stress-ng version 0.11.07. By varying the size of the memory allocation, we simulated different intensities of memory pressure exerted by other processes running on the same system as our target executable.

For each of the resulting 160 combinations of array placement, file size, and stress-ng allocation size, the test was executed 20 times. Table 1 shows the average percentage of memory successfully manipulated in the array for each configuration.

The results indicate that an increase in the size of the executable leads to a larger part of the executable's image in memory being manipulated by our attack. For instance, when stress-ng was configured to use 128 MiB of memory, no page of the text section nor of the data section of a small executable of 1 MiB could be manipulated. For a large executable of 256 MiB, 48.01 % of the text section and 46.15 % of the data section could be manipulated (cf. Fig. 5).

A similar trend can be observed for the stress-ng memory allocation size: The more memory stress-ng was configured to use, the larger was the proportion of the executable's text and data sections successfully manipulated. For example, 0 % of the text section and 0 % of the data section from a 64 MiB executable could be manipulated when stress-ng was configured to use 1 MiB of memory. When we increased the memory usage of stress-ng to 256 MiB, 63.84 % of the text section and 64.05 % of the data section could be manipulated (cf. Fig. 6).

Finally, we evaluated whether it is also possible to attack a program that is executed multiple times instead of being executed continuously for a long time, as in our *second scenario* (cf. Sect. 3.1). For this experiment, we used small version of our test program, i. e. the executable was 1 MiB in size and thus significantly smaller than the amount of available memory. The executable did not sleep before checking the array. When the executable was first started, results were in line with our first experiment: IMA did not notice an attack taking place and thus allowed the program to be executed, while the program's output matched that of the benign version.

Size (MiB)	Section	stress-ng Allocation Size (MiB)							
		1	32	64	128	192	256	320	384
1	text	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	16.62 (± 5.87)	78.56 (± 3.89)	100.00 (± 0.00)
	data	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.70 (± 4.92)	14.54 (± 4.41)	79.18 (± 7.45)	99.61 (± 0.00)
2	text	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	25.94 (± 9.84)	86.53 (± 3.87)	100.00 (± 0.00)
	data	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	21.41 (± 4.76)	86.89 (± 2.86)	99.80 (± 0.00)
4	text	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.24 (± 1.73)	0.00 (± 0.00)	31.29 (± 5.30)	89.73 (± 2.02)	100.00 (± 0.00)
	data	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	29.72 (± 8.70)	90.44 (± 1.96)	99.90 (± 0.00)
8	text	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	32.98 (± 4.47)	93.47 (± 0.87)	100.00 (± 0.00)
	data	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	35.33 (± 5.00)	93.27 (± 1.01)	99.95 (± 0.00)
16	text	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.58 (± 4.07)	31.60 (± 1.17)	91.57 (± 0.43)	100.00 (± 0.00)
	data	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.79 (± 5.60)	0.00 (± 0.00)	43.40 (± 7.76)	95.02 (± 0.62)	99.98 (± 0.00)
32	text	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	52.01 (± 5.67)	94.66 (± 0.36)	100.00 (± 0.00)
	data	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	2.91 (± 14.40)	51.98 (± 1.81)	95.43 (± 0.67)	99.99 (± 0.00)
64	text	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.57 (± 4.01)	9.73 (± 2.37)	63.84 (± 1.04)	97.15 (± 0.50)	100.00 (± 0.00)
	data	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	10.38 (± 2.56)	64.05 (± 1.34)	96.21 (± 0.49)	99.99 (± 0.00)
128	text	0.00 (± 0.00)	0.00 (± 0.00)	2.22 (± 10.98)	0.96 (± 0.32)	43.60 (± 0.78)	75.50 (± 0.78)	97.27 (± 0.44)	100.00 (± 0.00)
	data	0.00 (± 0.00)	0.00 (± 0.00)	0.00 (± 0.00)	5.46 (± 1.03)	43.29 (± 0.54)	75.34 (± 0.69)	97.09 (± 0.25)	100.00 (± 0.00)
256	text	11.88 (± 12.98)	13.91 (± 0.79)	24.91 (± 0.34)	48.01 (± 7.52)	65.80 (± 0.43)	84.24 (± 2.29)	98.76 (± 0.28)	100.00 (± 0.00)
	data	12.16 (± 1.51)	16.39 (± 12.10)	24.87 (± 0.36)	46.15 (± 0.36)	66.90 (± 4.78)	84.51 (± 0.28)	98.78 (± 0.25)	100.00 (± 0.00)
384	text	99.99 (± 0.00)	99.99 (± 0.00)	99.99 (± 0.00)	99.99 (± 0.00)	99.99 (± 0.00)	99.99 (± 0.00)	99.99 (± 0.00)	99.99 (± 0.00)
	data	99.99 (± 0.00)	99.99 (± 0.00)	99.99 (± 0.00)	99.99 (± 0.00)	99.99 (± 0.00)	99.99 (± 0.00)	99.99 (± 0.00)	99.99 (± 0.00)

Table 1: Average percentage and standard deviation of 50 runs of text and data section successfully manipulated by our attack, depending on the size of the executable and the allocation size of the stress-ng benchmark.

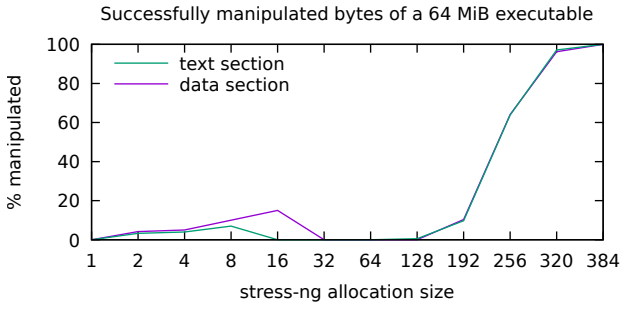

Figure 6: Diagram showing the percentage of successfully manipulated bytes for a 64 MiB executable, in relation to the allocation size of the stress-ng benchmark.

As the program is now no longer being executed, there is no need for the OS to keep it in memory, except for caching purposes. Contents of the cache will be overwritten by the OS if memory is needed by applications or if file I/O leads to other data being cached. Eventually, the executable will therefore disappear from memory over time. To simulate this, we manually evicted the page cache on our system (`echo 1 > /proc/sys/vm/drop_caches`).

We then started the executable again. IMA did not detect any changes in the file and allowed the program to be executed. However, the file was read from our malicious block device again and now produced a different output than the benign version. The full array could be manipulated successfully both when it was placed in the text section as well as when it was placed in the data section.

5 RELATED WORK

This section briefly presents mechanisms to protect the integrity of systems. Some of them are employed by different operating systems, some are subject to research.

Apple developed a technology called System Integrity Protection (SIP) which prevents modification of system files, even as `root` user. Unlike IMA, however, SIP lacks the capability of convincing a third party that the system has been booted into an acceptable state [4]. Additionally, macOS defaults to use Secure Boot (cf. Sect. 2) on certain devices [5].

Windows has two mechanisms to protect the integrity of the system. One is called "Code Integrity" and ensures that only digitally signed files are loaded into memory [34]. The other is called "Memory Integrity" and makes use of virtualisation technologies in order to isolate kernel-mode processes at run-time [31]. Both mechanisms, however, lack the ability to convince a verifier that the machine has been booted cleanly.

Chrome OS, based on Linux, makes use of dm-verity [16] which is an integrity mechanism for read-only data that works at block level using the Linux Device Mapper. Since checking an entire block device is susceptible to taking a very long time, each block is hashed separately and hashes are stored in a Merkle hash tree [27]. With

this design each block can be validated with reading a maximum of $log(n)$ other blocks.

Android uses fs-verity to ensure the integrity of files [28]. It is similar to dm-verity [27], but works on filesystem level, rather than block device level. A fs-verity enabled filesystem stores a hash-tree as additional data for a file, and evaluates it when the file or parts of it are read. "[…] fs-verity also re-verifies data each time it's paged in. This ensures that malicious disk firmware can't undetectably change the contents of the file at runtime" [28]. dm-verity and fs-verity use a block size which is equal to the system page size, i. e. usually 4096 [27, 28]. Measuring and verifying blocks upon every access represents the main difference to IMA.

Swierczynski et al. implement a real-world FPGA hardware Trojan insertion on a FIPS-140-2 level 2 certified USB flash drive from Kingston [45]. This shows that hardware attacks on a block level are a real threat, and that even on cryptographic hardware.

With DRIVE, Rein describes and implements a concept to detect runtime attacks on loaded software [40]. DRIVE continuously monitors the actual memory image of a binary, and compares it with the loaded binary code. All measurements are anchored in the TPM and, thus, are integrity-protected by cryptographic hardware. The expected memory image is predicted by using the binary file, the loading mechanism, and allocated memory addresses. This is why binary files themselves are used as references. During verification, loading of the binary is emulated using the allocated memory addresses. As a result, an expected in-memory reference measurement is produced and compared to the actual in-memory measurement. DRIVE reduces the attack surface for sophisticated adversaries who target volatile memory. It complements the load-time integrity concept of IMA. However, DRIVE is complex and hardware-specific, and no open-source implementations exist.

Another approach which targets runtime attack detection and prevention, is Control-Flow Integrity (CFI) [1]. It assumes that many current software attacks leverage exploits to subvert machine code execution. CFI detects runtime attacks by monitoring the branching behaviour of executed software, and its guarantees can be established formally. There is even hardware support for CFI, including shadow call stacks and access control for memory regions [18]. Moreover, CFI is available for low-end MCUs [36].

Especially for lower-end embedded devices, an approach exists, which uses hardware-specific features to assess how likely firmware is benign or malicious [50]. Similarly, it has been reported that power-related measurements are able to reveal firmware with unwanted modifications [12, 33]. The notion of pre-boot time attestation of the whole system, including "the contents of all processor and I/O registers and primary memories of a chipset and peripheral device controllers" [22], could work to detect modified firmware. Given the capacity of contemporary drives, the approach can only help to defend in certain environments with small storage.

6 DISCUSSION

In this section we discuss the scope of our attack as well as potential mitigations and their viability.

6.1 Attack Scope

We note that our subversion exploits a condition that is out of scope for the original IMA description [43]. We argue, however, that maliciously acting block devices are a real threat as it is known that hardware or its firmware is being manipulated while it is in transit [6, 23]. Firmware of block devices can be re-programmed [14, 53] through standard protocols [35]. Also, attacks on firmware of devices belonging to critical infrastructure have been subject to research [30].

The precondition for our attack is that an executable either is too large to fit into the caches of the machine, or that it gets evicted from the caches and must be re-read off the disk. We acknowledge that this scenario is not universally relevant, but we argue that it is common for embedded controllers to be resource-constrained and that it is hence more likely for those types of hardware to be susceptible to our attack.

While our attack works reliably in our test-bed, we have noticed that, sometimes, the target area with security sensitive information is read while the partition is mounted. Hence, we noticed three reads until our binary was executed. We hypothesise that Linux enumerates directories and that it helps to have security sensitive information not in the front of the partition.

6.2 Mitigations

The way how IMA is currently implemented in the Linux kernel is not sufficient. Just measuring a file once cannot work reliably if attacks on block device firmware are considered, given that a block device unilaterally decides to change a file. Hence, mechanisms as found in dm-verity and fs-verity should be evaluated as to whether they can alleviate the problem. In fact, making IMA aware of fs-verity mechanisms and allow pages to be verified as they are used has been discussed in the community[1].

We did not investigate Apple's Secure Boot as the public documentation is sparse. If Apple includes the firmware of the hard disk in the Secure Boot process, then they should effectively thwart our attack. Further, we did not look into Windows TPM Base Services (TBS) and Windows System Monitor (Sysmon).

An efficient countermeasure for our attack would be authenticated encryption of data at rest, i. e. full disk encryption, to protect data while the system is offline. With our attack, it is still possible to tamper with data, but it would become infeasible to construct valid manipulated data since the encryption key is not known.

Probabilistic multiple reads could also help to ensure the integrity of files which are partially read off the disk consecutively. For that, the operating system could randomly remember blocks read off the disk, or a hash of the block, respectively, and then schedule a re-read of the same block while it has not been changed by the OS. If the re-read block then has changed without the OS knowing, an attack is more likely.

Considering CFI [1] as a defence to our attack turns out not to be sufficient. In fact, an attacker could craft a malicious CFI-enabled binary and launch it using our attack. Pointer Authentication (PA) as deployed in ARMv8.3 helps to narrow down the attack surface [38]. It is susceptible to pointer reuse attacks, though [32]. PA can

[1] e.g. https://lore.kernel.org/linux-fsdevel/CAHk-=wh2j+Yy28H_QxEdsP=k9xcHxjCG1PqKAF2Uv=ckK8oPug@mail.gmail.com/

be used together with CFI in order strengthen defence mechanisms against function pointer manipulation [32]. Consequently, it becomes hard, but not impossible, for a sophisticated attacker to craft a malicious binary, and deploy it using our attack.

It is possible to discover our IMA subversion attack on executables with DRIVE [40]. It continuously monitors the memory image and reports changes into a TPM-protected log file, so that a remote party can verify it at al later point, and discover deviations from a known good binary. Further, DRIVE can compare the memory image in place with the initially loaded binary image, and in case of deviations, raise a notification. Given that with our attack we load different code on subsequent reads, DRIVE would detect the modification on its next check, or a remote verifier at a later point, respectively. DRIVE is only available for executables, so normal files would still be susceptible to our attack.

Both dm-verity and fs-verity can be used to detect our attack under certain circumstances because they verify every page before loading it into memory. However, in contrast to IMA they do not support a TPM as a trust anchor. This could be due to the rather slow TPM, which would be used very frequently. As a result of not involving a TPM, the assurance level of measurements for a verifier is substantially weaker. Further, dm-verity and fs-verity only work on read-only files, which may not be sufficient in some use cases.

Therefore, IMA serves some purposes that dm-verity and fs-verity cannot serve. It can be improved, though, to counter our discovered weakness. One approach could be to introduce blockwise file measurements based on Merkle hash trees, like dm-verity and fs-verity. That is, IMA would maintain multiple hashes per file, one for each block, and maintain multiple records per file in the SML. However, Merkle hash trees cannot be mapped to a TPM, per se.

As an alternative, simple block-wise measurements could be considered, where files are divided into static sized blocks, and then measured and recorded. Hence, another requirement would be an appropriate block size for hashing. dm-verity and fs-verity both use 4096 bytes. Taking into account the rather slow TPM, too small block sizes could render the system unusable, due to the latency added by the TPM. Bigger blocks, e.g. 4 MiB may alleviate the latency. One prerequisite for that approach to work is that, exactly like dm-verity and fs-verity, IMA must be invoked whenever parts of a file are read. That, however, is not the case in current implementations. Moreover, the whole approach would not only change the measurement behaviour of IMA substantially, but also that of a verifier investigating reported measurements in a remote attestation process.

7 CONCLUSION

This paper has shown how a maliciously acting block device can undermine the security guarantees IMA seeks to provide. By presenting a concrete attack scenario we have shown how IMA is relevant for critical infrastructures and how the attack breaks the expectations its operators had when setting up the system. The stealthiness of the attack prevents it from being detected and adds to the severity of the issue.

In our evaluation we showed the feasibility of our attack. We analysed the attack surface by evaluating how many bytes of different sized executables can be manipulated, depending on available system main memory. We found out that the bigger the executable and the bigger the already used main memory, the more bytes of both text and data section of the executable could be successfully manipulated. The text section was easier to manipulate, though, starting at smaller executable sizes and less already occupied main memory.

Future directions of research include the application and evaluation of our discovered attack to similar integrity measurement implementations, such the System Monitor (SysMon) Services in Microsoft Windows OS, and System Integrity Protection (SIP) in Apple macOS. Further interesting research targets improving the implementation of IMA in the Linux kernel, in order to remedy our attack. This may be possible by introducing block-wise file measurements based on Merkle hash trees, as already used by dm-verity.

Much like IMA, Windows can prevent executing binaries based on their hashes[2]. To the best of our knowledge, unlike with IMA, the hashes cannot be cryptographically signed, such that an attacker simply can provide appropriate hashes along with the files themselves. Although it is related, we leave it for future work to explore how susceptible the Windows mechanism is to our attack.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Trans. Inf. Syst. Secur.* 13, 1, Article Article 4 (Nov. 2009), 40 pages. https://doi.org/10.1145/1609956.1609960

[2] Uzair Amir. 2018. Schneider Electric Shipped USB Drives Loaded with Malware. https://www.hackread.com/schneider-electric-shipped-usb-drives-loaded-with-malware/

[3] Android Open Source Project. 2020. Implementing Dm-Verity. https://source.android.com/security/verifiedboot/dm-verity

[4] Apple. 2019. About System Integrity Protection on Your Mac. https://support.apple.com/en-us/HT204899

[5] Apple. 2020. About Secure Boot. https://support.apple.com/en-gb/HT208330

[6] Jacob Applebaum, Laura Poitras, Marcel Rosenbach, Christian Stöcker, and Jörg Schindler. 2013. The NSA Uses Powerful Toolbox in Effort to Spy on Global Networks - DER SPIEGEL - International. https://www.spiegel.de/international/world/the-nsa-uses-powerful-toolbox-in-effort-to-spy-on-global-networks-a-940969.html

[7] W.A. Arbaugh, D.J. Farber, and J.M. Smith. 1997. A Secure and Reliable Bootstrap Architecture. In *Proceedings. 1997 IEEE Symposium on Security and Privacy (Cat. No.97CB36097)*. 65–71. https://doi.org/10.1109/SECPRI.1997.601317

[8] Marty Hernandez Avedon, Duncan Mackenzie, Andres Mariano Gorzelany, Tina Burden, and Nick Schonning. 2018. Secure the Windows 10 Boot Process. https://docs.microsoft.com/en-us/windows/security/information-protection/secure-the-windows-10-boot-process

[9] Stefan Berger, Ramón Cáceres, Dimitrios Pendarakis, Reiner Sailer, Enriquillo Valdez, Ronald Perez, Wayne Schildhauer, and Deepa Srinivasan. 2008. TVDc: Managing Security in the Trusted Virtual Datacenter. (Jan. 2008).

[10] Stefan Berger, Kenneth Goldman, Dimitrios Pendarakis, David Safford, Enriquillo Valdez, and Mimi Zohar. 2015. Scalable Attestation: A Step toward Secure and Trusted Clouds. *IEEE Cloud Computing* 2, 5 (Sept. 2015), 10–18. https://doi.org/10.1109/MCC.2015.97

[11] Felix Bohling. 2020. *Subverting Linux' Integrity Measurement Architecture (IMA)*. Master's thesis. University of Hamburg, Germany.

[12] Dane Brown, Owens Walker, Ryan Rakvic, Robert W. Ives, Hau Ngo, James Shey, and Justin Blanco. 2018. Towards Detection of Modified Firmware on Solid State Drives via Side Channel Analysis. In *Proceedings of the International Symposium on Memory Systems (MEMSYS '18)*. Association for Computing Machinery, Alexandria, Virginia, USA, 315–320. https://doi.org/10.1145/3240302.3285860

[13] Catalin Cimpanu. 2018. Ships Infected with Ransomware, USB Malware, Worms. https://www.zdnet.com/article/ships-infected-with-ransomware-usb-malware-worms/

[14] Lucian Cojocar, Kaveh Razavi, and Herbert Bos. 2017. Off-the-Shelf Embedded Devices as Platforms for Security Research. In *Proceedings of the 10th European*

[2] https://github.com/MicrosoftDocs/windowsserverdocs/blob/master/WindowsServerDocs/identity/software-restriction-policies/software-restriction-policies-technical-overview.md

Workshop on Systems Security (EuroSec'17). Association for Computing Machinery, Belgrade, Serbia, 1–6. https://doi.org/10.1145/3065913.3065919

[15] Luigi Coppolino, Michael Dr Jaeger, Nicolai Kuntze, and Roland Rieke. 2012. A Trusted Information Agent for Security Information and Event Management. In *ICONS 2012*.

[16] Jonathan Corbet. 2011. Dm-Verity. https://lwn.net/Articles/459420/

[17] Marco De Benedictis and Antonio Lioy. 2019. Integrity Verification of Docker Containers for a Lightweight Cloud Environment. *Future Generation Computer Systems* 97 (Aug. 2019), 236–246. https://doi.org/10.1016/j.future.2019.02.026

[18] Ruan de Clercq and Ingrid Verbauwhede. 2017. A survey of Hardware-based Control Flow Integrity (CFI). arXiv:cs.CR/1706.07257

[19] Joan.G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert van Doorn, and Sean W. Smith. 2001. Building the IBM 4758 Secure Coprocessor. *Computer* 34, 10 (Oct. 2001), 57–66. https://doi.org/10.1109/2.955100

[20] Apostolos P. Fournaris, Lidia Pocero Fraile, and Odysseas Koufopavlou. 2017. Exploiting Hardware Vulnerabilities to Attack Embedded System Devices: A Survey of Potent Microarchitectural Attacks. *Electronics* 6, 3 (Sept. 2017), 52. https://www.mdpi.com/2079-9292/6/3/52

[21] Gentoo Foundation, Inc. 2019. Integrity Measurement Architecture. https://wiki.gentoo.org/wiki/Integrity_Measurement_Architecture

[22] Virgil Gligor and Maverick Woo. 2018. Requirements for Root of Trust Establishment. In *Security Protocols XXVI (Lecture Notes in Computer Science)*, Vashek Matyáš, Petr Švenda, Frank Stajano, Bruce Christianson, and Jonathan Anderson (Eds.). Springer International Publishing, Cham, 192–202.

[23] Glenn Greenwald. 2014. *No Place to Hide: Edward Snowden, the NSA, and the U.S. Surveillance State* (first ed.). Metropolitan Books/Henry Holt, New York, NY.

[24] Bare J. Christopher. 2006. Attestation and Trusted Computing. https://courses.cs.washington.edu/courses/csep590/06wi/finalprojects/bare.pdf

[25] Ludovic Jacquin, Adrian L. Shaw, and Chris Dalton. 2015. Towards Trusted Software-Defined Networks Using a Hardware-Based Integrity Measurement Architecture. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. IEEE, London, UK, 1–6. https://doi.org/10.1109/NETSOFT.2015.7116186

[26] Rick Kennell and Leah H. Jamieson. 2003. Establishing the Genuinity of Remote Computer Systems. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12 (SSYM'03)*. USENIX Association, Washington, DC, 21.

[27] The kernel development community. 2019. *dm-verity*. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/verity.html

[28] The kernel development community. 2019. *fs-verity: read-only file-based authenticity protection*. https://www.kernel.org/doc/html/latest/filesystems/fsverity.html

[29] Obaid Khalid, Carsten Rolfes, and Andreas Ibing. 2013. On Implementing Trusted Boot for Embedded Systems. In *2013 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, Austin, TX, USA, 75–80. https://ieeexplore.ieee.org/document/6581569

[30] Charalambos Konstantinou and Michail Maniatakos. 2015. Impact of Firmware Modification Attacks on Power Systems Field Devices. In *2015 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. 283–288. https://doi.org/10.1109/SmartGridComm.2015.7436314

[31] Beth Levin and Marty Hernandez Avedon. 2019. Memory Integrity. https://docs.microsoft.com/en-us/windows/security/threat-protection/device-guard/memory-integrity

[32] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. 2019. PAC It up: Towards Pointer Integrity Using ARM Pointer Authentication. In *Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19)*. USENIX Association, USA, 177–194.

[33] Ryan S. McDowell, Hau Ngo, Ryan Rakvic, Owens Walker, Robert W. Ives, and Dane Brown. 2019. Using Current Draw Analysis to Identify Suspicious Firmware Behavior in Solid State Drives. In *2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. 92–97. https://doi.org/10.1109/CSE/EUC.2019.00027

[34] Microsoft. 2019. WDAC and Virtualization-Based Code Integrity (Windows 10) - Windows Security. https://docs.microsoft.com/en-us/windows/security/threat-protection/device-guard/introduction-to-device-guard-virtualization-based-security-and-windows-defender-application-control

[35] Karsten Nohl and Jakob Lell. 2014. BadUSB - On Accessories That Turn Evil. https://www.blackhat.com/us-14/briefings.html#badusb-on-accessories-that-turn-evil

[36] Thomas Nyman, Jan-Erik Ekberg, Lucas Davi, and N. Asokan. 2017. CFI CaRE: Hardware-Supported Call and Return Enforcement for Commercial Microcontrollers. In *Research in Attacks, Intrusions, and Defenses*, Marc Dacier, Michael Bailey, Michalis Polychronakis, and Manos Antonakakis (Eds.). Springer International Publishing, Cham, 259–284.

[37] Magnus Nyström, Martin Nicholes, and Vincent J Zimmer. 2011. UEFI Networking and Pre-Os Security. *Intel Technology Journal* 15, 1 (2011), 80–102. https://www.intel.com/content/dam/www/public/us/en/documents/research/2011-vol15-iss-1-intel-technology-journal.pdf

[38] Qualcomm Technologies, Inc. 2017. *Pointer Authentication on ARMv8.3*. Technical Report.

[39] Tobias Rauter, Andrea Höller, Johannes Iber, Michael Krisper, and Christian Kreiner. 2017. Integration of Integrity Enforcing Technologies into Embedded Control Devices: Experiences and Evaluation. In *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*. Christchurch, New Zealand, 155–164. https://doi.org/10.1109/PRDC.2017.29

[40] Andre Rein. 2017. DRIVE: Dynamic Runtime Integrity Verification and Evaluation. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIA CCS '17)*. Association for Computing Machinery, New York, NY, USA, 728–742. https://doi.org/10.1145/3052973.3052975

[41] Jordan Robertson and Michael Riley. 2018. China Used a Tiny Chip in a Hack That Infiltrated U.S. Companies. *Bloomberg.com* (Oct. 2018). https://www.bloomberg.com/news/features/2018-10-04/the-big-hack-how-china-used-a-tiny-chip-to-infiltrate-america-s-top-companies

[42] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. 2004. Attestation-Based Policy Enforcement for Remote Access. In *Proceedings of the 11th ACM Conference on Computer and Communications Security - CCS '04*. ACM Press, Washington DC, USA, 308. https://doi.org/10.1145/1030083.1030125

[43] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. 2004. Design and Implementation of a TCG-Based Integrity Measurement Architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium (SSYM'04)*, Vol. 13. USENIX Association, San Diego, CA, USA, 16.

[44] Andreas Steffen. 2012. The Linux Integrity Measurement Architecture andTPM-Based Network Endpoint Assessment. In *Linux Security Summit*. San Diego, CA, USA. https://www.strongswan.org/lss2012.pdf

[45] Pawel Swierczynski, Marc Fyrbiak, Philipp Koppe, Amir Moradi, and Christof Paar. 2017. Interdiction in practice–Hardware Trojan against a high-security USB flash drive. *Journal of Cryptographic Engineering* 7, 3 (2017), 199–211. https://doi.org/10.1007/s13389-016-0132-7

[46] Trusted Computing Group. 2018. Standards for Securing Industrial Equipment. https://trustedcomputinggroup.org/wp-content/uploads/04_TCG_StdsSecureEquip_2018_Web.pdf

[47] Trusted Computing Group. 2018. TCG Guidance for Securing Network Equipment Using TCG Technology. https://trustedcomputinggroup.org/wp-content/uploads/TCG_Guidance_for_Securing_NetEq_1_0r29.pdf

[48] Trusted Computing Group. 2020. Welcome To Trusted Computing Group. https://trustedcomputinggroup.org/

[49] J. Douglas Tygar and Bennet Yee. 1991. Dyad: A System for Using Physically Secure Coprocessors. *Proceedings of the Joint Harvard-MIT Workshop on Technological Strategies for the Protection of Intellectual Property in the Network Multimedia Environment* (May 1991).

[50] Xueyang Wang, Charalambos Konstantinou, Michail Maniatakos, Ramesh Karri, Serena Lee, Patricia Robison, Paul Stergiou, and Steve Kim. 2016. Malicious Firmware Detection with Hardware Performance Counters. *IEEE Transactions on Multi-Scale Computing Systems* 2, 3 (July 2016), 160–173. https://doi.org/10.1109/TMSCS.2016.2569467

[51] Zhaohui Wang, Ryan Johnson, and Angelos Stavrou. 2012. Attestation & Authentication for USB Communications. In *2012 IEEE Sixth International Conference on Software Security and Reliability Companion*. 43–44. https://doi.org/10.1109/SERE-C.2012.43

[52] Bo Yang, Yu Qin, Yingjun Zhang, Weijin Wang, and Dengguo Feng. 2016. TMSUI: A Trust Management Scheme of USB Storage Devices for Industrial Control Systems. In *Information and Communications Security (Lecture Notes in Computer Science)*, Sihan Qing, Eiji Okamoto, Kwangjo Kim, and Dongmei Liu (Eds.). Springer International Publishing, Cham, 152–168.

[53] Jonas Zaddach, Anil Kurmus, Davide Balzarotti, Erik-Oliver Blass, Aurélien Francillon, Travis Goodspeed, Moitrayee Gupta, and Ioannis Koltsidas. 2013. Implementation and Implications of a Stealth Hard-Drive Backdoor. In *Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC '13)*. Association for Computing Machinery, New Orleans, Louisiana, USA, 279–288.

[54] Kim Zetter. 2015. How the NSA's Firmware Hacking Works and Why It's So Unsettling. *Wired* (Feb. 2015). https://www.wired.com/2015/02/nsa-firmware-hacking/

[55] Mimi Zohar and Dmitry Kasatkin. 2018. Integrity Measurement Architecture (IMA). https://sourceforge.net/p/linux-ima/wiki/Home/

[56] Mimi Zohar and David Safford. 2010. An Overview of the Linux Integrity Subsystem. http://downloads.sf.net/project/linux-ima/linux-ima/Integrity_overview.pdf

[57] Mimi Zohar, David Safford, and Reiner Sailer. 2009. Using IMA for Integrity Measurement and Attestation. https://blog.linuxplumbersconf.org/2009/slides/David-Stafford-IMA_LPC.pdf