# T-RAID: TEE-based Remote Attestation for IoT Devices

Roland Nagy, Márton Bak, Dorottya Papp, and Levente Buttyán(✉)

Laboratory of Cryptography and System Security (CrySyS Lab),
Department of Networked Systems and Services,
Budapest University of Technology and Economics, Budapest, Hungary
`buttyan@crysys.hu`
`https://www.crysys.hu/`

**Abstract.** The Internet of Things (IoT) consists of network-connected embedded devices that enable a multitude of new applications, but also create new risks. In particular, embedded IoT devices can be infected by malware. Operators of IoT systems not only need malware detection tools, but also scalable methods to reliably and remotely verify malware freedom of their IoT devices. In this paper, we address this problem by proposing T-RAID, a remote attestation scheme for IoT devices that takes advantage of the security guarantees provided by a Trusted Execution Environment running on each device.

**Keywords:** Internet of things · Embedded systems · Malware · Remote attestation · Trusted Execution Environment

## 1 Introduction

The Internet of Things (IoT) consists of network-connected embedded devices that enable a multitude of new applications in various domains, such as industrial automation, transportation, building automation, healthcare, and agriculture – just to mention a few of them. The use of IoT technologies can make applications *smarter*: they provide the technological foundations for transforming buildings into smart buildings, cities into smart cities, transportation systems into intelligent transportation systems, healthcare into personalized healthcare, agriculture into precision agriculture, and factories into smart factories.

However, as usual, new technologies also create new risks. In particular, due to the increasing levels of automation and connectedness, our new, smart and intelligent applications are now exposed to cyberattacks. To address the problem, academic researchers and industry alliances are actively working on IoT security solutions [1,12,13], standards [8,10], and guidelines [7,16,17], and regulatory

The presented work was carried out within the SETIT Project (2018-1.2.1-NKP-2018-00004), which has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2018-1.2.1-NKP funding scheme.

E. Gelenbe et al. (Eds.): EuroCybersec 2021, CCIS 1596, pp. 76–88, 2022.
https://doi.org/10.1007/978-3-031-09357-9_7

bodies are also making steps[1] to ensure that those solutions, standards, and guidelines are indeed used and followed in practice.

One particular security issue is that IoT devices can be infected by malware [2,4], which can alter their behavior, endangering the integrity and the availability of IoT systems, and undermining the trustworthiness of the smart applications based on them. Hence, system operators need malware detection solutions adapted to the constraints of IoT systems [20]. In addition, they also need scalable methods to reliably verify malware freedom of IoT devices in their systems. In this paper, we address this problem by proposing a remote attestation scheme for IoT devices that takes advantage of the security guarantees provided by a Trusted Execution Environment (TEE) running on the device.

Attestation is meant to be a process whereby a trusted verifier reliably checks the state of an untrusted prover, and remote attestation is when this verification is done remotely via a network. In our case, the prover is (a process running on) an IoT device, and it is untrusted, because the device may be compromised by a malware. The verifier is a trusted remote server operated by the system operator. We use remote attestation to prove the malware-free state of the IoT devices to the system operator. Doing this remotely means that the operator does not need physical access to the devices, allowing for large scale, automated verification of all devices in an entire IoT system.

The structure of the paper is the following: We start by giving a brief overview on different approaches to remote attestation in Sect. 2, serving as a review of relevant related work and also providing technical background for our proposal. Next, we introduce T-RAID, our TEE-based solution to secure remote attestation for IoT devices, by first providing a general overview of it in Sect. 3, and then presenting its protocols in more details in Sects. 4 and 5. Finally, we evaluate T-RAID and discuss its properties in Sect. 6, and conclude the paper in Sect. 7.

## 2 Approaches to Remote Attestation

There exist three general approaches to attestation: hardware-based, software-based, and hybrid attestation. Hardware-based attestation relies on a secure co-processor (e.g., a TPM chip[2]) that can produce a digitally signed summary of the hardware and software state of the device being verified. The summary is typically a hash computed by progressively combining the hashes of the system components and software modules started during the boot process. The key used to sign the summary is kept in the co-processor and protected by its logical access control and physical tamper resistance features. As this approach requires a co-processor, it is typically considered to be too expensive for embedded IoT devices.

[1] The California IoT cybersecurity law SB-327 became effective Jan 1, 2020.

[2] https://trustedcomputinggroup.org/work-groups/trusted-platform-module/ Last visited: Sep 12, 2021.

In contrast to the hardware-based approach, software-based attestation does not require any additional hardware in the prover device. Solutions following this approach (e.g., [18,19]) are typically based on executing a protocol in which the verifier probes the prover and the response of the prover to the probe convinces the verifier that the prover is in a malware-free state. To generate the response, the prover runs a checksum function, which traverses memory locations in a pseudo-random manner (seeded by the verifier's probe). The verifier checks the correctness of the prover's response by computing the same checksum function on the expected memory content of the device. If malware is hiding in the memory, either the checksum of the prover will differ from that computed by the verifier, or the response time of the prover will be longer than expected, as the malware needs to check and redirect memory accesses that refer to locations holding the malware code itself. So besides checking the correctness of the checksum, in this case, the verifier also checks the response time of the prover.

The main problem of software-based attestation is that, in practice, it does not really work over a network due to network jitter, which makes it practically impossible to remotely measure the exact checksum computation time of the prover [14]. Another problem is that a compromised prover can actually delegate checksum computation to a much faster attacker device, which cannot be detected by a remote verifier. In addition, even if we do not consider such delegation attacks, this approach assumes that the checksum computation on the prover cannot be performed faster than the speed of the actual implemention of the checksum function. Unfortunately, this assumption does not always hold [3], leading to attacks where a tricky faster way of computing the checksum leaves time for the malware to check and redirect memory references that would reveal its presence.

Given all these problems, hybrid approaches were proposed (e.g., [5,6]) that are largely software-based, but also require minimal hardware support. For example, in [6], the authors propose a scheme, applicable to attestation purposes, that uses a ROM-based checksum routine and relies on a secret key for authenticating the computed checksum that is kept in memory accessible only by ROM-based code. This latter property is provided by a hardware-based memory access logic, which verifies that the instruction pointer is in the ROM region when the secret key is being accessed. This actually ensures that the confidentiality of the key is preserved, even if the device is infected by a malware. In addition, as the ROM code cannot be modified, integrity of the checksum computation is also ensured. This means that a response authenticated by the secret key must be genuine, and such a response can be verified by a remote verifier.

At this point, a natural question could be the following: What is the minimum hardware support needed for a hybrid remote attestation solution to be secure? This question is investigated in [9], where the authors conclude that the following set of requirements is sufficient and necessary (hence minimal) for secure remote attestation of embedded devices:

1. Custom hardware to enforce exclusive access to a secret key;
2. Reliable and secure memory erasure;

3. Read-only-memory (ROM);
4. Instructions for enabling and atomically disabling interrupts;
5. Custom hardware to enforce that the attestation (checksum) routine can only be invoked by running its first instruction;
6. Secure reset mechanism.

It turns out that the guarantees provided by satisfying the requirements above can also be achieved in another way: In [5], the authors propose a hybrid remote attestation scheme, called HYDRA, that relies on security features provided by a formally verified seL4 microkernel to obtain similar properties. In HYDRA:

1. A privileged process handles the secret key and enforces proper access control to it;
2. Reliable and secure memory erasure is required by [9] to ensure that no information about the secret key is leaked after using it in the checksum authentication. However, the strict memory separation of the seL4 microkernel ensures the same property;
3. ROM is required by [9] to make sure that the checksum routine cannot be modified. Isolated process memory and code integrity checks in seL4 can provide the same property;
4. Prioritized interrupt handling of the microkernel ensures uninterruptable execution of the checksum code that runs with the highest possible priority;
5. Controlled invocation is enforced by operating system support;
6. Secure reset is initiated in [9] whenever an attempt is detected to execute the checksum function from the middle of its code. This is not needed if controlled invocation is enforced.

The authors of [5] claim that using seL4 imposes fewer hardware requirements on the underlying microprocessor, and building upon a formally verified software component increases confidence in security of the overall solution.

## 3 Overview of T-RAID

Our main idea is to follow the approach of HYDRA [5], but instead of a secure microkernel, we rely on a Trusted Execution Environment (TEE). A TEE provides an isolated environment for trusted processes where they can execute without being interfered by normal, potentially compromised processes. In addition, a TEE also provides secure storage for secrets that is not accessible from outside the TEE. Thus, checking malware-freedom can be implemented in a trusted process running in the TEE and the key used for authenticating the result of the check can be stored in TEE-protected secure memory.

We note that embedded devices equipped with ARM or similarly powerful processors are typically capable of hosting such a TEE. We also note that TEEs usually rely on hardware support to provide their security guarantees.

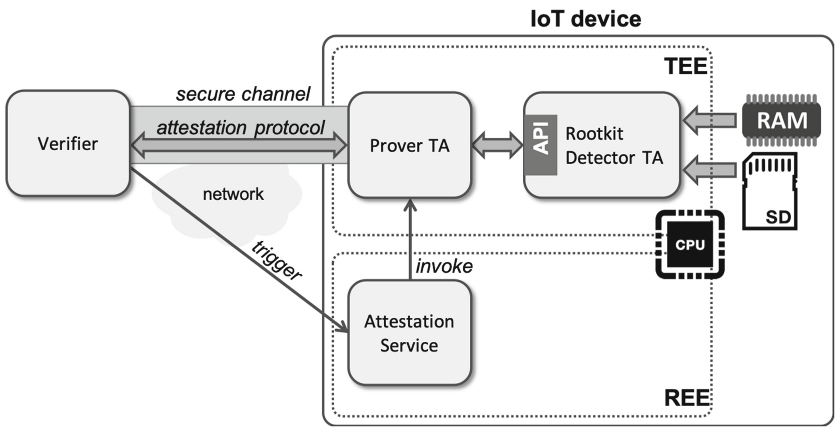

**Fig. 1.** Overview of the T-RAID architecture.

For example, TrustZone[3] enabled ARM processors support TEEs by offering hardware-enforced memory isolation. A special register in the processor keeps track whether it runs in the so called *Normal World* or in the *Secure World*. In the Normal World, access to certain memory regions and peripherals associated with the Secure World is denied, and this is enforced by the memory bus fabric. On the other hand, processes running in the Secure World have virtually unlimited access to any resources of the embedded device. In addition, switching between Worlds is possible only by invoking a special instruction that guarantees a proper context switch. TEE implementations usually take advantage of this low level support built into the processor itself.

The architecture of our TEE-based remote attestation scheme designed for IoT devices (T-RAID) is illustrated in Fig. 1. In T-RAID, the Verifier is assumed to be a remote entity that interacts with the Prover via a network. The Prover is a trusted application (TA) running in the TEE hosted by the processor of the IoT device. An Attestation Service is running as a normal (untrusted) process in the Rich Execution Environment (REE), also hosted by the processor. The term "rich" refers to the fact that the REE may be provided by a full-blown operating system (OS), such as Linux, that offers an abundance of services to the processes running in the REE. The TEE typically features a much more limited OS that provides only basic services to the trusted applications. Isolation between the TEE and the REE is supported by the processor and its memory management unit.

In order to initiate an attestation session, the Verifier calls the Attestation Service, which is assumed to be always available. If it is not, then this fact already proves that the IoT devices is not in its normal state, and it may be compromised by a malware. The Attestation Service invokes the Prover TA via

[3] https://developer.arm.com/ip-products/security-ip/trustzone Last visited: Sep 12, 2021.

a controlled invocation mechanism provided by the TEE (and supported by the processor). The Prover TA then establishes a secure connection to the Verifier, which is used to execute a remote attestation protocol securely.

In the remote attestation protocol, the Verifier challenges the Prover with a set of tasks. These tasks consists in the execution of certain integrity checks that are implemented by a Rootkit Detector TA, also running in the TEE. The integrity checks requested by the Verifier are invoked by the Prover TA via a well-defined API provided by the Rootkit Detector TA, and the results are sent back to the Verifier via the secure channel.

The Rootkit Detector TA has access to the entire memory of the IoT device and its persistent storage. The memory includes the memory of the processes and the OS kernel running in the REE and the persistent storage includes the file system that they use. The integrity checks implemented by the Rootkit Detector TA analyze this memory and file system, collect relevant data (e.g., the list of processes currently running in the REE), compute hashes (e.g., the hash of the text segment of the OS kernel in the REE), and try to detect anomalies that may signal the presence of a malware (e.g., hooked function pointers).

More information about the remote attestation protocol and the integrity checks of T-RAID is provided below in Sects. 4 and 5, respectively.

We implemented T-RAID as a prototype running in QEMU[4]. The target architecture of our prototype is the ARM processor, and we use OP-TEE[5] as the TEE implementation and Linux as the OS in the REE. Cryptographic functions in our protocols use the mbedTLS cryptographic library[6]. In the sequel, we assume this setup when implementation specific details are described.

## 4 Remote Attestation Protocol

The remote attestation protocol of T-RAID assumes a secure channel between the Prover TA and the Verifier, thus, a prerequisite for running the protocol is to establish such a secure channel. One can use TLS for this purpose if the TEE implementation supports TLS-protected sockets and the resource constraints of the IoT device permit the use of such a complex protocol as TLS. In our prototpye implementation, we could not use TLS, because OP-TEE does not support TLS-based sockets in the TEE. Instead, we designed and implemented a lightweight secure channel protocol over a raw socket. Our protocol uses an authenticated version of the Diffie-Hellman protocol to establish a shared secret between the Prover TA and the Verifier (where authentication is based on ECDSA signatures); the PBKDF2 key derivation function to derive a 32-byte symmetric encryption key and a 32-byte message authentication key from the shared secret; the AES cipher in CBC mode with PKCS#7 padding to encrypt messages; message sequence numbers to protect against replay attacks;

[4] https://www.qemu.org/ Last visited: Sep 12, 2021.
[5] https://www.op-tee.org/ Last visited: Sep 12, 2021.
[6] https://github.com/ARMmbed/mbedtls Last visited: Sep 12, 2021.

and HMAC with SHA-256 as the hash function to authenticate (encrypted and numbered) messages.

The attestation protocol consists of the exchange of a single attestation request and response. The secure channel guarantees the freshness, integrity, authenticity, and confidentiality of these messages, and most importantly, the Prover's response. The request of the Verifier may contain multiple challenges, each triggering the call of a specific integrity check function of the Rootkit Detector TA. The response of the Prover contains the results of the integrity check functions called. The integrity check functions may return a status code (e.g., 0 for a successful check and 1 for a failure), a hash value (e.g., hash of the REE OS kernel's text segment or recursive hash of some part of the REE file system), or a list of process IDs and process names extracted from various OS kernel data structures (e.g., the process list, the process tree, and the run queues).

The Verifier must be able to verify the results of the checks received in the response of the Prover. For this, we assume that the Verifier stores the hash values expected in correct responses. In case of file system checks, the computed hash value depends on what parts of the file system are actually hashed, therefore, we assume that the Verifier has a mirror of the file system of the IoT device and performs the same hash computation on this mirror to obtain the expected correct hash value. Finally, if the response contains a list of process names, the Verifier can compare that to a pre-defined white list of process names allowed on the IoT device.

## 5 Integrity Checks

Our integrity checks perform rootkit detection on the IoT device; hence, their successful execution is an assurance of malware freedom. The software component implementing the checks is capable of accessing components of the REE, such as the memory and the file system. The latter is not supported by OP-TEE, so we had to extend and slightly modify the OP-TEE kernel and the `tee-supplicant` daemon (a component of OP-TEE running in the REE). More details on this can be found in our earlier paper [15]. Using the aforementioned capabilities, we implemented numerous checks, each aiming at detecting a different rootkit technique or ensuring the integrity of REE components that our checks rely upon. In this section, we present the integrity checks that the Prover TA can invoke.

### 5.1 Process Listing

The most important functionality of any kernel is to manage and schedule processes. In order to achieve these goals, the Linux kernel uses so called tasks. A task is approximately equivalent to a thread. Single-threaded processes consist of one task, while multi-threaded processes have one task for each thread, sharing the same address space. These tasks are organized into multiple dynamic data structures. Here we present the ones used by the 5.1 version of the Linux kernel.

Our solution uses these data structures to enumerate processes on the system. Currently, we can list process IDs and names of the processes.

The oldest and simplest data structure in the kernel holding process-related information is the so called process list. This is a doubly-linked circular list of task structures; each task has a `next` and `prev` pointer, pointing to the next and previous tasks in the list, respectively. Using these pointers, we can easily traverse the whole list, starting from the `init_task`; this is the first kernel thread, started at boot.

Another data structure is the process tree. When a process starts another process, it becomes its child, while the new process refers to the original one as its parent. Via this relation, processes form a tree, whose root is the `init_task`. The Linux kernel uses lists to implement this tree. Each process has a pointer to its first and last child, while the children are linked into a doubly-linked circular list. We traverse this data structure recursively in a depth-first manner.

Pid namespaces are used by the kernel as an isolation mechanism. There is an initial pid namespace containing every process. These namespaces use radix trees to account the process IDs in use. These radix trees store `pid` structures with pointers to the tasks using the specific ID. To traverse the initial pid namespace, we implemented a function capable of finding the corresponding `pid` structure in the tree for a given ID.

Finally, we extract process related information from runqueues. These queues are used by the scheduler, and unlike the previous data structures, not every process is accessible from these, only the runnable ones. These are the processes not waiting for anything and not stopped, they can continue their execution, if assigned to a CPU core. Each core has its own runqueue, and runqueues implement data structures for every scheduling policy. For Linux 5.1, this means 3 subschedulers, using lists, red-black trees and nested red-black trees.

### 5.2 Memory Integrity Checks

We check the integrity of two memory areas of the Linux kernel that are frequently targeted by rootkits: the system call table and the text segment of the kernel itself. System calls are the interface the kernel offers to user-space processes. When processes need to perform actions that are the kernel's responsibility, they invoke the appropriate system call. Such actions include interactions with files, network sockets, etc. The kernel uses an array of function pointers, known as the system call table. Rootkits often replace pointers in this array and re-implement certain system calls. Therefore, we compute the hash of the system call table using the SHA-256 hash algorithm.

Another common and similar rootkit technique is *inline hooking* [11]. In this case, the attacker modifies the code of an existing function, usually by rewriting the first few instructions to a jump such that during execution, the code jumps to the desired replacement. To detect inline hooks, we compute the SHA-256 hash of the entire text segment of the kernel, which contains all the code of the Linux kernel.

### 5.3 File Integrity Checks

By-default OP-TEE does not provide access to the file systems of the REE, however, it is capable of using it for Secure Storage. The API written for this does not aim to be a general purpose API for file access, so we had to extend it and apply some patches in order to be able to access arbitrary files. Again, for implementation details, the reader is referred to [15].

Our implementation provides a simple interface which can be used to check the integrity of any part of the REE file systems. This can be done invoking two functions, namely `hash_file` and `hash_dir`. The former one expects a filename as parameter, and an output buffer, where the computed hash of the file will be stored. If the file exists, it opens it, reads it by 4096-byte-long blocks, and feeds these blocks into a hash context. After reaching the end of the file, the SHA-256 hash is written to the output buffer. The latter one expects more parameters: a directory name, an output buffer, a boolean indicating if it should hash the directory recursively, and an optional blacklist. In case of non-recursive hashing, all subdirectories will be ignored. If a blacklist is supplied, all elements are checked against it, thereby the hashing files or directories with volatile content can be avoided. The contents of directories are sorted alphabetically.

### 5.4 Network Checks

We also implemented checks targeting the network stack of the kernel. In this subsystem, rootkits typically implement two kind of attacks: hiding open connections and implementing "magic packet" functionality. In the case of rootkits, this means performing a predefined action, when the infected system receives a specially crafted network packet. So far, we identified one way to hide open sockets and three mechanisms what can be abused by attackers to implement magic packets. For these checks to work properly, we assume that every necessary driver is compiled into the kernel.

The most common way to implement magic packets is using the Netfilter subsystem, the backend of Linux firewall solutions. Netfilter stores firewall rules in so-called chains. Each supported protocol (like IPv4, IPv6, ARP, etc.) has five chains, one for each stage of packet processing. Each chain acts as an arraylist, containing Netfilter hooks, storing function pointers. When a packet is checked against a specific chain, all hooks in the chain are invoked, and the packet is accepted only, if all hooks accept it. These hook functions, however, can have side effects, so an attacker can implement a firewall rule which executes his payload, if certain conditions are met. Our solution traverses all the hooks of every chain, and if any of the function pointers store a value that is not pointing into the text segment of the kernel, it is considered to be a sign of rootkit infection.

We also check structures called `icmp_control`. The kernel uses an array of these to determine what handler function should be executed for different kinds of ICMP packets. The packet's `type` field is used to index this array. We check all function pointers the same way as we did in the case of Netfilter hooks.

The kernel uses `net_protocol` structures to register handler functions for different protocols, like UDP, TCP and IGMP. These structures contain handler, error handler and demultiplexer functions, which can be hooked and used to implement magic packet functionality. We perform the same integrity check on these pointers as described above.

Finally, we implemented a check targeting hidden network connections. Files in the `/proc/net` directory give information about open connections. The content of these files is generated on-demand using `seq_ops`. These objects store function pointers to iterate a specific data structure and display information about its elements. Rootkits often target these to hide open connections, therefore we check these function pointers the same way as we checked the others.

## 6 Evaluation and Discussion

The presented TEE-based remote attestation scheme, T-RAID, provides security guarantees similar to those of HYDRA [5]:

1. A trusted application in the TEE, the Prover TA, handles the private key used to set up a secure channel with the Verifier. Every message, including the Prover's responses to the Verifier's challenges are authenticated by this channel, which means that the Verifier can be sure that the responses come from the given Prover. In addition, the private key of the Prover is stored in the secure storage of the TEE, hence, the key remains protected and invisible from the potentially compromised REE.
2. Strict separation of the secure memory used by the TEE from processes in the REE prevents the leakage of the private key after it has been used.
3. TEE-based integrity protection of TAs prevents their illegitimate modification by untrusted processes of the REE. This property ensures similar guarantees for TAs as a ROM would ensure. Thus, neither the Prover TA nor the Rootkit Detector TA can be modified, and hence, the integrity checks are performed and their results are reported correctly to the Verifier.
4. Interrupts can be disabled and re-enabled in the TEE. Disabling them when the Rootkit Detector TA is invoked ensures the uninterruptable execution of our integrity checks, with some caveats that we discuss later in this section.
5. TEE-based invocation mechanism of TAs enforces that the execution of a TA always begins at its entry point. This contributes to the correct execution of the integrity checks and correct reporting of their results.
6. Secure reset is not needed, as the TEE enforces the controlled invocation of every TA.

Unfortunately, our current prototype has two known weaknesses. The first one is that file operations in OP-TEE are delegated to the REE side where they can actually be interrupted. Moreover, as file operations usually take a long time, they will almost surely be interrupted by the task scheduler of the OS. This is a limitation of OP-TEE, other TEE implementations may implement file operations within the TEE itself. Nevertheless, if T-RAID is implemented using OP-TEE as the TEE, one has to be aware that property 4 may not hold. Disabling the file system related integrity checks would make property 4 satisfied, but then malware could clear itself from memory and hide its components in persistent storage, from which it may be potentially reloaded later on.

The second weakness is that on multi-core processors, such as most ARM processors, other, potentially untrusted processes may run in parallel to our TAs on different cores. Those processes may interfer with the execution of our integrity checks. For instance, a malware running on a different core could remove a hook from the system call table before it is hashed by our Rootkit Detector TA and put the hook back once the hashing is completed. The only reliable countermeasure to this is disabling all but one cores during the entire attestation process. At the time of this writing, we are experimenting with the implementation of this idea.

## 7 Conclusions

In this paper, we proposed T-RAID, a TEE-based remote attestation scheme for IoT devices. T-RAID follows the hybrid approach to remote attestation: it is mostly based on software and uses only limited hardware support. Notably, T-RAID relies only on the hardware support provided for the TEE itself by the processor and its memory management unit. Considering that TEEs are already widely supported by certain classes of embedded devices, T-RAID is an affordable solution for IoT systems built from such devices.

We showed that T-RAID has similar security properties to those of HYDRA, a secure remote attestation scheme proposed in the past. T-RAID, however, performs more complex integrity checks on the device aiming at detecting rootkits both in memory and in persistent storage. While our integrity checks effectively detect malware, unfortunately, our current prototype implementation of T-RAID has some weaknesses stemming from limitations of OP-TEE, the TEE implementation that we use, and the inherent parallelism provided by multi-core processor architectures.

## References

1. Alrawi, O., Lever, C., Antonakakis, M., Monrose, F.: SoK: security evaluation of home-based IoT deployments. In: IEEE Symposium on Security and Privacy, pp. 1362–1380 (2019). https://doi.org/10.1109/SP.2019.00013
2. Antonakakis, M., et al.: Understanding the Mirai botnet. In: USENIX Security Symposium, pp. 1093–1110. USENIX Association, August 2017
3. Castelluccia, C., Francillon, A., Perito, D., Soriente, C.: On the difficulty of software-based attestation of embedded devices. In: ACM Conference on Computer and Communications Security (CCS), pp. 400–409 (2009). https://doi.org/10.1145/1653662.1653711
4. Cozzi, E., Vervier, P.A., Dell'Amico, M., Shen, Y., Bigle, L., Balzarotti, D.: The tangled genealogy of IoT malware. In: Annual Computer Security Applications Conference (ACSAC) (2020)

5. Eldefrawy, K., Rattanavipanon, N., Tsudik, G.: HYDRA: hybrid design for remote attestation (using a formally verified microkernel). In: ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec), pp. 99–110 (2017). https://doi.org/10.1145/3098243.3098261
6. Eldefrawy, K., Tsudik, G., Francillon, A., Perito, D.: SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In: Network and Distributed Systems Symposium (NDSS) (2012)
7. ENISA: Guidelines for securing the Internet of Things. ENISA study, November 2020
8. ETSI: CYBER; Cyber security for consumer Internet of Things: Baseline requirements. ETSI TS 103 645 v2.1.2, June 2020
9. Francillon, A., Nguyen, Q., Rasmussen, K.B., Tsudik, G.: A minimalist approach to remote attestation. In: Conference on Design, Automation & Test in Europe (DATE), pp. 1–6 (2014)
10. Global Platform: Security evaluation standard for IoT platforms v1.1 (SESIP). Global Platform Standard, June 2021
11. Gu, J., Xian, M., Chen, T., Du, R.: A Linux rootkit improvement based on inline hook. In: Proceedings of the 2nd International Conference on Advances in Mechanical Engineering and Industrial Informatics, pp. 793–798. Atlantis Press (2016). https://doi.org/10.2991/ameii-16.2016.155
12. Hassija, V., Chamola, V., Saxena, V., Jain, D., Goyal, P., Sikdar, B.: A survey on IoT security: application areas, security threats, and solution architectures. IEEE Access **7**, 82721–82743 (2019). https://doi.org/10.1109/ACCESS.2019.2924045
13. Kumar Jain, V., Gajrani, J.: IoT security: a survey of issues, attacks and defences. In: Sharma, H., Saraswat, M., Kumar, S., Bansal, J.C. (eds.) CIS 2020. LNDECT, vol. 61, pp. 219–236. Springer, Singapore (2021). https://doi.org/10.1007/978-981-33-4582-9_18
14. Li, Y., Cheng, Y., Gligor, V., Perrig, A.: Establishing software-only root of trust on embedded systems: facts and fiction. In: International Workshop on Security Protocols, pp. 50–68 (2015). https://doi.org/10.1007/978-3-319-26096-9_7
15. Nagy, R., Németh, K., Papp, D., Buttyán, L.: Rootkit detection on embedded IoT devices. Acta Cybernetica, August 2021. https://doi.org/10.14232/actacyb.288834
16. NIST: Considerations for managing Internet of Things (IoT) cybersecurity and privacy risks. NISTIR 8228, June 2019
17. NIST: Baseline security criteria for consumer IoT devices. NIST draft white paper, August 2021
18. Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.: SWATT: software-based attestation for embedded devices. In: IEEE Symposium on Security and Privacy, pp. 272–282 (2004). https://doi.org/10.1109/SECPRI.2004.1301329
19. Seshadri, A., Luk, M., Shi, E., Perrig, A., van Doorn, L., Khosla, P.: Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. ACM SIGOPS Oper. Syst. Rev. **35**(5), 1–16 (2005). https://doi.org/10.1145/1095809.1095812
20. Tamás, C., Papp, D., Buttyán, L.: SIMBIoTA: similarity-based malware detection on IoT devices. In: Proceedings of the 6th International Conference on Internet of Things, Big Data and Security - IoTBDS, pp. 58–69. SciTePress (2021). https://doi.org/10.5220/0010441500580069