Requirements for Root of Trust Establishment

Virgil Gligor(✉) and Maverick Woo

CyLab, Carnegie Mellon University, Pittsburgh, USA
gligor@cmu.edu

Abstract. Root-of-Trust (RoT) establishment assures that either a state of an untrusted system contains all and only content chosen by an external verifier *and* the verifier's code begins execution in that state, or the verifier discovers the existence of unaccounted content. RoT establishment is sufficient to assure program booting in *malware-free system states*, and necessary for establishing *secure initial states* for any software system. In particular, it is necessary for software deployed in access control and cryptographic applications despite the presence of an adversary (e.g., persistent malware) that controls that system. In this paper, we define requirements for RoT establishment and their relationships. These requirements differ from those for software-based and cryptographic attestation protocols. We point out these differences and explain why these protocols cannot be expected to satisfy the defined RoT requirements. Then we argue that *jointly* satisfying all these requirements yields a secure solution for establishing malware-free states – a strictly weaker requirement than RoT establishment. However, to establish RoT, it is sufficient to load a family of almost universal hash functions in a malware-fee state and then verify their outputs when applied to state components.

1 Introduction

Suppose a user has a trustworthy program and attempts to boot it in a system state. The *system state* comprises the contents of all processor and I/O registers and primary memories of a chipset and peripheral device controllers at a particular time; e.g., before boot. If any malicious software (*malware*) can execute instructions anywhere in system state, the user wants to discover the presence of malware with high probability. This goal has not been achieved to date. System components that are *not* directly addressable by CPU instructions or by trusted hardware modules enable malware to become *persistent*; i.e., to survive in non-volatile memories of system states despite repeated power cycles, secure-, and trusted-boot operations [1], and to infect the rest of the system state. For example, persistent malware has been found in the firmware of peripheral controllers [2–5], network interface cards [3,6,7], disk controllers [8–11], routers and firewalls [11], as well as that of removable devices [12]. A remote adversary can retain long-term control of a user's system via persistent malware.

© Springer Nature Switzerland AG 2018
V. Matyáš et al. (Eds.): Security Protocols 2018, LNCS 11286, pp. 192–202, 2018.
https://doi.org/10.1007/978-3-030-03251-7_23

Suppose that the user attempts to initialize the system state to content that she chooses; e.g., she includes a small I/O program for loading a trustworthy microhypervisor or microkernel later, initializes the primary memory to chosen values, and reflashes device-controller firmware to malware-free code. Furthermore, her chosen content may also satisfy some security invariants; e.g., the system is disconnected from the Internet, and it has the configuration expected by the microhypervisor. Now the user wants to verify that the system, which may have been infected by malware and hence is untrusted, has been initialized to the content chosen and hence known by the user.

Root of trust (RoT) establishment on an untrusted system assures that a system state comprises *all* and *only* content chosen by, and known to, the user, and the user's code *begins execution* in that state. *All* implies that no content is missing, and *only* implies that no extra content exists. If a system state is initialized to content that satisfies security invariants and RoT establishment succeeds, a user's code begins execution in a *secure initial state*. Then trustworthy OS and application programs booted in a secure initial state can extend this state to include secondary storage. If RoT establishment fails, unaccounted content, such as malware, exists. Hence, RoT establishment is sufficient for (and stronger than) assuring malware freedom and necessary for all access control models and cryptographic protocols, since all need secure initial states.

In this paper, we answer the following questions:

- How can RoT be established without secrets and trusted hardware modules?
- Can past attestation protocols provide a viable solution to RoT? If not, what requirements are not satisfied?
- Can jointly satisfying these requirements lead to a sound RoT establishment protocols? If not, what additional mechanisms are necessary?

Specifically,

- we define requirements for RoT establishment without secrets and trusted hardware modules, and discuss their relationships;
- we show that past attestation protocols have had different goals than RoT establishment, and hence cannot be expected to satisfy the requirements defined herein;
- we argue that jointly satisfying these requirements leads to establishment of malware-free states.
- we argue that loading a simple family of universal hash functions [24] – one per system component – and verifying their outputs when applied to those components yields RoT establishment.

2 Software-Based Attestation - An Overview

To define the requirements for RoT establishment we review the basic steps of software-based attestation [25–29] for a simple untrusted system connected to a *local* trusted verifier.

Suppose that the simple system has a processor and a m-word memory comprising random-access memory, processor, and I/O registers. The verifier asks the system to initialize the m words to chosen content. Then the verifier challenges the system to perform a computation $C_{m,t}(\cdot)$ on a pseudorandom *nonce* in the m words and time t. Suppose that $C_{m,t}(\cdot)$ is space-time (i.e., $m - t$) optimal, $C_{m,t}(nonce)$ is unpredictable by an adversary, and the computation is non-interruptible. If the system returns $C_{m,t}(nonce)$ to the local verifier in time t, then after accounting for the local communication delay, the verifier concludes that the memory state contains all and only the chosen content.

When applied to multiple device controllers, this protocol proceeds from the faster controllers to the slower ones, repeating the attestation of the faster ones, so that they do not end execution early and act as *proxies* for the slow ones [3].

3 Adversary Definition

Our adversary can exercise all known attacks that insert persistent malware into a computer system, including having brief access to that system; e.g., an EFI attack by an "evil maid". Also, it can control malware remotely and extract all software secrets stored in the system, via a network channel. Malware can read and write the verifier's local I/O channel, which is always faster and has less transfer-time variability than the adversary's network channel. However, malware does not have access to the verifier's code nor to the true random number generator.

We assume the adversary *can break* all complexity-based cryptography but *cannot predict* the true random numbers to be received from the verifier. Also, the adversary can optimize $C_{m,t}$'s code in a system at no cost; i.e., it can encode small values of *nonces* and memory content into the immediate address fields of instructions to lower $C_{m,t}$'s space and/or time below m, t, in zero extra time and memory space. Furthermore, the adversary can output the result of a different computation that takes less time than t or space than m, or both. Why is the no-cost code change a prudent assumption in a timed protocol? First, some code changes can take place before a *nonce*'s arrival, which marks the beginning of the timed protocol. Second, to account for (e.g., cache and TLB) jitter caused by random memory access by typical $C_{m,t}(\cdot)$ computations, verifiers' time measurements typically build in some slack time; e.g., 0.2%–2.6% of t [2,26,29–31]. This could enable an adversary to exploit the slack and use unaccounted instructions; viz., [31].

4 Requirements

4.1 Concrete Optimality

Optimality of a $C_{m,t}$ computation means that its lower bounds match the upper bounds non-asymptotically[1] in both memory size, m, and execution time, t. If the

[1] Different constants of asymptotic lower and upper bounds of $C_{m,t}$ cause these bounds to differ for concrete values of m and t.

time bounds differ, the verifier faces a fundamental problem: its measurement becomes either useless or meaningless. If the measurement is checked against the theoretical lower bound, which is often unattainable in practice, then false positives as high as 100% would render attestation useless. If it's checked against a value that exceeds the theoretical lower bound or if it's checked against the upper bound, then it would be impossible to prove that an adversary's code could not produce better timing and render attestation meaningless. In contrast, if the optimal time bounds are non-asymptotic, the only challenge is to reduce the measurement slack in specific systems, which is an engineering, rather than a basic computational complexity, problem.

If the memory bounds differ, the adversary can exercise time-memory trade-offs and the verifier faces a similar measurement dilemma as above. For example, such trade-offs can be exploited in the evaluation of univariate polynomials of degree d in algebraic models of computation, where $t \cdot m^2 \geq d/8$ [32].

Optimality in a concrete computational model. Optimal bounds for any computation always depend on the model of computation used. For example, lower bounds differ with the instruction set architectures (ISA) of a *practical WRAM* model – which is close to a real computer – even for simple computations such as static dictionaries [33,34]. Few optimal bounds exist in these models, even if asymptotic, despite the fact that their variable word length allows the use of the circuit-based complexity hierarchy. Instead, lower bounds for more complex problems have been proved only in the *cell probe* model [40], where references to memory cells (i.e., bits) are counted, but *not* instruction executions. Unfortunately, these lower bounds cannot be used for any $C_{m,t}$ since they can never match upper bounds non-asymptotically, and are unreachable in reality.

Optimality retention in $C_{m,t}$ composition. In abstract WRAM models program optimality is considered without regard of whether extra system code and data in memory could invalidate the program's lower bounds. Input data and optimal programs simply exist in system registers and memory, and *I/O* operations and register initialization (*Init*) are assumed to be done already.

In contrast, a concrete WRAM model must be implemented in real systems, and hence it must include I/O registers and instructions (e.g., for data transfers, interrupt handling, busy/done status) and instructions that initialize special registers and configure processors; e.g., clear/evict cache and TLB content, disable VM. Thus $C_{m,t}$'s code must be composed with I/O and *Init* code, which could invalidate $C_{m,t}$'s lower bounds. Hence, proving its optimality must account for *all* extra code and data in memory when $C_{m,t}$'s code runs, and hence the less extra code and data the better.

Unpredictability of $C_{m,t}(nonce)$. Most optimality results are obtained assuming honest execution of $C_{m,t}$'s code. An execution is *honest* if the $C_{m,t}$ code is fixed (i.e., committed) before it reads any inputs, and returns correct results for all inputs. Unfortunately, the optimality of $C_{m,t}$'s code in honest execution does not necessarily hold in adversarial execution since an adversary can change $C_{m,t}$'s code both before and after receiving the *nonce*, or simply guess $C_{m,t}(nonce)$ without executing any instructions. Before the *nonce*'s arrival, the adversary

can modify the code independent of the verifier's *nonce*. After a *nonce*'s arrival, the adversary can check its value, and determine the best possible code modification, at no cost. For example, she can encode a small-value *nonce* into immediate address fields of instructions, and save register space and instruction executions. More insidiously, an adversary could change the entire code to that of another function $C'_{m',t'}(\cdot)$ and $nonce'$, such that $(C'_{m',t'}, nonce') \neq (C_{m,t}, nonce)$, $(m', t') < (m, t)$, and $C'_{m',t'}(nonce') = C_{m,t}(nonce)$.

The adversary's goal is to write the result $C_{m,t}(nonce)$ into the output register after executing fewer instructions, if any, and/or using less memory than the honest optimal code. If adversary succeeds with better than low probability over the pseudo-random choice of *nonce*, then she could execute unaccounted instructions that arbitrarily modify system state before returning the result, which would remain undetected.

To counter all possible adversary behaviors, we require that the adversary succeeds in writing $C_{m,t}(nonce)$ to the output register with low probability over the *nonce*, after executing fewer instructions, if any, and/or using less memory than the honest optimal code. We call this requirement the *unpredictability of* $C_{m,t}(nonce)$. Thus, the correctness and timeliness of $C_{m,t}(nonce)$ must imply unpredictability.

4.2 Protocol Atomicity

The *verifier's protocol* begins with the input of the *nonce* challenge in a system and ends when the verifier receives the system's output; e.g., $C_{m,t}(nonce)$. Protocol atomicity requires integrity of control flow across the instructions of the verifier's protocol with *each* system component; i.e., each device controller and the (multi)processor(s) of the chipset. Asynchronous events, such as future-posted interrupts, hardware breakpoints on instruction execution or operand access [29], and inter-processor communication, can violate control-flow integrity *outside* of $C_{m,t}(\cdot)$'s execution. For instance, a malware instruction can post a *future* interrupt *before* the verifier's protocol begins execution. The interrupt could trigger *after* a correct and timely $C_{m,t}(nonce)$ result is sent to the verifier, and execute code that undetectably corrupts system state [31]. Clearly, optimality of $C_{m,t}(\cdot)$ is insufficient for control-flow integrity. Nevertheless, optimality is necessary: otherwise, a predictable $C_{m,t}(nonce)$ would allow time and space for an *interrupt-enabling* instruction to be executed undetectably.

Verifiable control flow. Instructions that disable asynchronous events must be executed before $C_{m,t}(\cdot)$. Their execution inside $C_{m,t}(\cdot)$ would violate optimality bounds, and after $C_{m,t}(\cdot)$ would be ineffective: asynchronous events could trigger during the execution of the last $C_{m,t}(\cdot)$ instruction. However, verification that an instruction is located before $C_{m,t}(\cdot)$ in memory (e.g., via a digital signature or a MAC) does *not* guarantee its execution. The adversary code could simply skip it before executing $C_{m,t}(\cdot)$. Hence, verification must address the apparent cyclic dependency: on the one hand, the execution of the event-disabling instructions before $C_{m,t}(\cdot)$ requires control-flow integrity, and on the other, control-flow integrity requires the execution of the event-disabling instructions before $C_{m,t}(\cdot)$.

Concurrent all-or-nothing transaction. Protocol atomicity also requires that the verifier's protocol with n device controllers and CPUs of the (multiprocessor) chipset is implemented as a concurrent all-or-nothing transaction. That is, *all* optimal $C_{m_1,t_1}, \ldots, C_{m_n,t_n}$ codes for the n components must execute concurrently and pass verification. This prevents register and memory modification of already attested devices (e.g., reinfection) by yet-to-be attested devices, not only *proxy* attacks [3]. Note that powering off individual devices and powering them one-at-a-time before performing individual attestation is inadequate because some (e.g., chipset) devices cannot be powered-off without system shutdown, and insufficient because malicious firmware can still corrupt an already attested controllers after power-on and before attestation starts.

Concurrent all-or-nothing execution requires that for distinct *fixed* t_i's, the faster C_{m_j,t_j} computations be performed $k_j \geq \lceil max(t_i)/t_j \rceil$ times, where $max(t_i)$ is the optimal time bound of the slowest device controller. As shown in the next section, a protocol does not exist that uses a *fixed* t_i for a given m_i and produces concurrent all-or-nothing execution, and at the same time retains both C_{m_i,t_i}'s optimality m and result unpredictability. Hence, atomicity requires a *scalable* time bound t; i.e., t can be increased independent of the constant memory bound m and yet preserves $C_{m,t}$'s optimality[2].

5 Past Attestation Protocols and RoT Establishment

Past attestation protocols, whether software-based [25,26,29,35,38], cryptographic-based [17,19–21,39], or hybrid [3,41], have *different* security goals than those of RoT requirements defined here: some are weaker and some are stronger. For example, whether these protocols are used for single or multiple devices, they typically aim to verify a weaker property, namely the integrity of software – not system – state. However, they also satisfy a stronger property: in all cryptographic and hybrid attestation protocols the verification can be remote and can be repeated after boot, rather than local and limited to pre-boot time as here.

Given their different goals, it is unsurprising that past protocols do not satisfy some RoT establishment requirements defined here, even for bounded adversaries and secret-key protection in trusted hardware modules. For example, these protocols need not be concerned with system's *register content* (e.g., for general processor and I/O registers), since they cannot contain executable code. Furthermore, they need not satisfy the concurrent all-or-nothing *atomicity* (see Sect. 4.2) of the verifier's protocol since they need not establish any state properties, such as secure initial state in multi-device systems. Finally, since none of these systems aim to satisfy security properties *unconditionally*, they do not require that verifiers are equipped with true random number generators; e.g., pseudo-random numbers are sufficient for *nonce* generation. Beyond these common differences, past protocols exhibit some specific differences.

[2] This is the opposite of *perfect* universal hash functions, which seek a *constant* t independent of the scalable m.

Software-based attestation. Some applications in which software-based attestation is beneficially used do not require control-flow integrity [37], and naturally this requirement need not always be satisfied [31,36]. Here we illustrate a more subtle challenge that arises *if* one uses traditional checksum designs for RoT establishment in a multi-device system, where the concurrent all-or-nothing requirement becomes important. That is, some past designs cannot jointly satisfy concurrent all-or-nothing atomicity and either code optimality or result unpredictability. Software-based attestation models [38] also face this challenge.

Some past $C_{m,t}$ computations are checksums that have a *fixed* bound t for a given m. Let a concurrent all-or-nothing transaction comprise checksums $C_{m_1,t_1}, \ldots, C_{m_n,t_n}$ for n devices. This implies that some $C_{m,t}$ must be executed $k \geq \lceil max(t_i)/t \rceil$ times and its executions $C_{m,t}(nonce_0), \ldots, C_{m,t}(nonce_{k-1})$ must be linked to eliminate idle waiting [3]. Suppose that linking is done by the verifier: optimal $C_{m,t}(nonce_j)$ cannot end execution until it inputs $nonce_{j+1}$ from the verifier. Then $C_{m,t}$ can no longer be optimal, since the variable input-synchronization delays within $C_{m,t}$ invalidate the optimal time bounds t[3]. If synchronization buffers of $nonce_{j+1}$, m also becomes invalid.

Alternatively, suppose that $C_{m,t}$'s executions are linked through nonces: $nonce_{j+1} = C_{m,t}(nonce_j)$. However, $C_{m,t}(nonce_{j+1})$'s unpredictability requires that its input $nonce_{j+1}$ is pseudo-random. This would no longer be guaranteed since $C_{m,t}$ need *not* be a pseudo-random function; e.g., Pioneer's checksum [26] and its descendants (e.g., [29]) are not.

Despite their differences from RoT establishment, software-based attestation designs met their goals [2,26,35], and offered deep insights on how to detect malware on peripheral controllers [3], embedded devices [31,36], mobile phones [30], and specialized processors; e.g., TPMs [29].

Cryptographic attestation. Cryptographic protocols for remote attestation typically require a trusted hardware module, which can be as simple as a ROM module [18], to protect a secret key for computing digital signatures or MACs. If used in applications that require control-flow integrity for the signature or MAC computation, as in RoT establishment, a trusted hardware model in each device must protect *both* the secret key *and* the signature/MAC generation code. Otherwise, these applications would exhibit similar control-flow vulnerabilities as software-based attestation.

More importantly, cryptographic attestation relocates the trust to the third parties who install the cryptographic keys in each device and those who distribute them to verifiers. The trustworthiness of these parties can be uncertain; e.g., a peripheral-controller supplier operating in jurisdictions that can compel the disclosure of secrets could not guarantee the secrecy of the protected cryptographic key. Similarly, the integrity of the distribution channel for the signature-verification certificate established between the device supplier/integrator and

---

[3] Input synchronization delays for $nonce_{j+1}$ within a $checksum_j$ computation on a network interface card (Netgear GA 620) that takes time t can be as high as $0.4t$ with a standard deviation of about $0.0029t$; see [3], Sects. 5.4.2-5.4.4.

verifier can be compromised, which enables known attacks; e.g., see the Cuckoo attack [13]. Thus, these protocols aim to offer only conditional security.

Nevertheless if the risk added when third parties manage one's system secrets is acceptable *and* protocol atomicity requirements can be met, then cryptographic protocols for remote attestation could be used in RoT establishment.

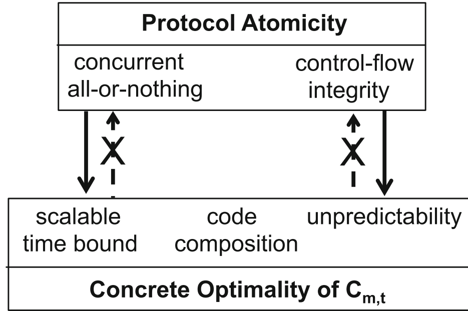

Fig. 1. Requirements for RoT establishment

6 Satisfying RoT Requirements – Overview

Necessity. Figure 1 summarizes the relationships among the requirements for RoT establishment. Atomicity of the verifier's protocol has unavoidable dependencies on both $C_{m,t}(\cdot)$'s scalable time bounds and unpredictability. As illustrated above, identifying dependencies is important because they show which requirements must be jointly satisfied to discharge proof obligations for establishing malware-free states. It is also useful since unnecessary dependencies can introduce cycles that often rule out proofs; e.g., the spurious optimality dependency on atomicity [31].

The concrete optimality requirements must be jointly satisfied independent of protocol atomicity. First, unpredictability must not depend on control-flow integrity: even if an adversary can trace the execution of each instruction of optimal evaluation code, she cannot write the correct result into the output register by executing fewer instructions or using less memory than optimal, except with very small probability over guessing.

Second, unpredictability enables verifiable execution of instructions that disable asynchronous events durably, which achieves control-flow integrity.

Third, a scalable optimal time bound and the implementation of concurrent all-or-nothing transactions based on them will complete the support for establishing malware-free states.

Sufficiency. Note that jointly satisfying the requirements presented above only yields malware-free states – a *strictly weaker* property than RoT establishment. However, it is sufficient to load a family of almost universal hash functions [24] in a malware-fee state and then verify their outputs when applied to state components to establish RoT unconditionally.

Acknowledgment. Comments received from Gene Tsudik and Adrian Perrig helped clarify the differences between RoT establishment and past attestation protocols.

References

1. Parno, B., McCune, J.M., Perrig, A.: Bootstrapping Trust in Modern Computers. Springer Briefs in Computer Science, vol. 10. Springer, New York (2011). https://doi.org/10.1007/978-1-4614-1460-5
2. Li, Y., McCune, J.M., Perrig, A.: SBAP: software-based attestation for peripherals. In: Acquisti, A., Smith, S.W., Sadeghi, A.-R. (eds.) Trust 2010. LNCS, vol. 6101, pp. 16–29. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13869-0_2
3. Li, Y., McCune, J.M., Perrig, A.: VIPER: verifying the integrity of PERipherals' firmware. In: Proceedings of the 18th ACM Conference on Computer and Communications Security, pp. 3–16. ACM Press (2011)
4. Cui, A., Costello, M., Stolfo, S.: When firmware modifications attack: a case study of embedded exploitation. In: Proceedings of the 2013 Network and Distributed Systems Security Symposium, ISOC (2013)
5. Stewin, P.: Detecting Peripheral-based Attacks on the Host Memory. T-Lab Series in Telecommunication Services. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-319-13515-1
6. Delugre, G.: Closer to metal: reverse engineering the broadcom NetExtreme's firmware. In: Sogeti ESEC Lab. (2010)
7. Duflot, L., Perez, Y.-A., Morin, B.: What if you can't trust your network card? In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 378–397. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23644-0_20
8. Zaddach, J., et al.: Implementation and implications of a stealth hard-drive backdoor. In: Proceedings of the 29th Annual Computer Security Applications Conference (ACSAC). ACM (2013)
9. Mearian, L.: There's no way of knowing if the NSA's spyware is on your hard drive. Computerworld **2** (2015)
10. Raiu, C.: Equation: The Death Star of the Malware Galaxy, February 2015
11. Applebaum, J., Horchert, J., Stocker, C.: Catalog reveals NSA has back doors for numerous devices, vol. 29 (2013)
12. Greenberg, A.: Why the security of USB is fundamentally broken. In: Wired Magazine, Number July (2014)
13. Parno, B.: Bootstrapping trust in a trusted platform. In: Proceedings of the 3rd Conference on Hot Topics in security, pp. 1–6. USENIX Association (2008)
14. Lone-Sang, F., Nicomette, V., Deswarte, Y.: I/O attacks in intel-pc architectures and countermeasures. In: Proceedings of the Symposium for the Security of Information and Communication Technologies SSTIC (2011)

15. Lone-Sang, F., Nicomette, V., Deswarte, Y.: A tool to analyze potential I/O attacks against PCs. In: IEEE Security and Privacy, pp. 60–66 (2014)
16. Kaspersky Lab: The Duqu 2.0 - Technical Details (version 2.1). Technical report (2015)
17. Eldefrawy, K., Perito, D., Tsudik, G.: SMART: Secure and minimal architecture for (establishing a dynamic) root of trust, February 2012
18. Koeberl, P., Schulz, S., Sadeghi, A.-R., Varadharajan, V.: TrustLite: a security architecture for tiny embedded devices. In: Proceedings of the Ninth European Conference on Computer Systems, EuroSys 2014 (2014)
19. Asokan, N., et al.: SEDA: scalable embedded device attestation. In: Proceedings of the 2015 ACM Conference on Computer and Communications Security. ACM (2015)
20. Ibrahim, A., Sadeghi, A.R., Tsudik, G., Zeitouni, S.: DARPA: device attestation resilient to physical attacks. In: Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks. WiSec 2016, pp. 171–182. ACM (2016)
21. Ibrahim, A., Sadeghi, A.R., Zeitouni, S.: SeED: secure non-interactive attestation for embedded devices. In: Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks. WiSec 2017, pp. 64–74 (2017)
22. Lipton, R., Ostrovsky, R., Zikas, b.: Provable virus detection: using the uncertainty principle to protect against malware. Cryptology ePrint Archive, Report 2015/728 (2015). http://eprint.iacr.org/2015/728
23. Lipton, R., Ostrovsky, R., Zikas, V.: Provably secure virus detection: using the observer effect against malware. In: 43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, 11–15 July 2016, Rome, Italy, pp. 32:1–32:14 (2016)
24. Thorup, M.: High speed hashing for integers and strings. CoRR arXiv:1504.06804, September 2015
25. Spinellis, D.: Reflection as a mechanism for software integrity verification. ACM Trans. Inf. Syst. Secur. **3**(1), 51–62 (2000)
26. 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. In: Proceedings of the 20th ACM Symposium on Operating Systems Principles, pp. 1–16. ACM (2005)
27. Seshadri, A., Luk, M., Perrig, A., van Doorn, L., Khosla, P.: SCUBA: secure code update by attestation in sensor networks. In: Proceedings of the 5th ACM Workshop on Wireless Security, pp. 85–94. ACM (2006)
28. Seshadri, A., Luk, M., Qu, N., Perrig, A.: SecVisor: a tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In: Proceedings of 21st ACM Symposium on Operating Systems Principles, pp. 335–350. ACM (2007)
29. Kovah, X., Kallenberg, C., Weathers, C., Herzog, A., Albin, M., Butterworth, J.: New results for timing-based attestation. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy, pp. 239–253. IEEE (2012)
30. Jakobsson, M., Johansson, K.A.: Retroactive detection of malware with applications to mobile platforms. In: Proceedings of the 5th USENIX Workshop on Hot Topics in Security, USENIX (2010)
31. Li, Y., Cheng, Y., Gligor, V., Perrig, A.: Establishing software-only root of trust on embedded systems: facts and fiction. In: Christianson, B., Švenda, P., Matyáš, V., Malcolm, J., Stajano, F., Anderson, J. (eds.) Security Protocols 2015. LNCS, vol. 9379, pp. 50–68. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-26096-9_7

32. Aldaz, M., Heintz, J., Matera, G., Montaa, J., Pardo, L.: Time-space tradeoffs in algebraic complexity theory. J. Complex. **16**(1), 2–49 (2000)
33. Miltersen, P.B.: Lower bounds for static dictionaries on RAMs with bit operations but no multiplication. In: Meyer, F., Monien, B. (eds.) ICALP 1996. LNCS, vol. 1099, pp. 442–453. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61440-0_149
34. Andersson, A., Miltersen, P.B., Riis, S., Thorup, M.: Static dictionaries on AC^0 RAMs: query time $\Theta(\sqrt{(}log\ n/log\ log\ n))$ is necessary and sufficient. In: Proceedings of 37th FOCS, pp. 441–450 (1996)
35. Seshadri, A., Perrig, A., van Doorn, L., Khosla, P.: SWATT: software-based attestation for embedded devices. In: Proceedings of the IEEE Symposium on Security and Privacy, pp. 272–282. IEEE (2004)
36. Castelluccia, C., Francillon, A., Perito, D., Soriente, C.: On the difficulty of software-based attestation of embedded devices. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, pp. 400–409. ACM (2009)
37. Perrig, A., van Doorn, L.: Refutation of "on the difficulty of software-based attestation of embedded devices" (2010)
38. Armknecht, F., Sadeghi, A.R., Schulz, S., Wachsmann, C.: A security framework for the analysis and design of software attestation. In: Proceedings of the 2013 ACM Conference on Computer and Communications Security, pp. 1–12. ACM (2013)
39. Francillon, A., Nguyen, Q., Rasmussen, K.B., Tsudik, G.: A minimalist approach to remote attestation. In: Proceedings of the Conference on Design, Automation & Test in Europe. DATE 2014, 3001 Leuven, Belgium, Belgium, pp. 244:1–244:6. European Design and Automation Association (2014)
40. Yao, A.C.-C.: Should tables be sorted? J. ACM **28**(3), 615–628 (1981)
41. Zhao, J., Gligor, V., Perrig, A., Newsome, J.: ReDABLS: revisiting device attestation with bounded leakage of secrets. In: Christianson, B., Malcolm, J., Stajano, F., Anderson, J., Bonneau, J. (eds.) Security Protocols 2013. LNCS, vol. 8263, pp. 94–114. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41717-7_12