Establishing Software Root of Trust Unconditionally

Virgil D. Gligor, Maverick S.L. Woo

November 7, 2018

CMU-CyLab-18-003

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

Establishing Software Root of Trust Unconditionally

Virgil D. Gligor Maverick S.L. Woo

Carnegie Mellon University
CMU-CyLab TR 18-003
November 7, 2018

***Abstract*—Root-of-Trust (RoT) establishment ensures either that the state of an untrusted system contains all and only content chosen by a trusted local verifier and the system code begins execution in that state, or that the verifier discovers the existence of unaccounted for content. This ensures program booting into system states that are free of persistent malware. An adversary can no longer retain undetected control of one's local system.**

We establish RoT *unconditionally*; i.e., without secrets, trusted hardware modules and instructions, or bounds on the adversary's computational power. The specification of a system's chipset and device controllers, and an external source of true random numbers, such as a commercially available quantum RNG, is all that is needed. Our system specifications are those of a concrete Word Random Access Machine (cWRAM) model – the closest computation model to a real system with a large instruction set.

We define the requirements for RoT establishment and explain their differences from past attestation protocols. Then we introduce a RoT establishment protocol based on a new computation primitive with concrete (non-asymptotic) optimal space-time bounds in adversarial evaluation on the cWRAM. The new primitive is a *randomized polynomial*, which has k-independent uniform coefficients in a prime order field. Its collision properties are stronger than those of a k-independent (almost) universal hash function in cWRAM evaluations, and are sufficient to prove existence of malware-free states before RoT is established. Preliminary measurements show that randomized-polynomial performance is practical on commodity hardware even for very large k.

To prove the concrete optimality of randomized polynomials, we present a result of independent complexity interest: a Horner-rule program is *uniquely optimal* whenever the cWRAM execution space and time are simultaneously minimized.

I. Introduction

Suppose a user has a trustworthy program, such as a formally verified micro-kernel [45] or a micro-hypervisor [89], and attempts to boot it into a specific system state. The *system state* comprises the contents of all processor and I/O registers and random access 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 the system state, the user wants to discover the presence of such 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 survive in non-volatile memories despite repeated power cycles, secure- and trusted-boot operations [67]; i.e., malware becomes *persistent*. For example, persistent malware has been found in the firmware of peripheral controllers [19], [53], [83], network interface cards [20], [21], disk controllers [5], [59], [71], [93], USB controllers [2], as well as routers and firewalls [5]. Naturally, persistent malware can infect the rest of the system state, and thus a remote adversary can retain long-term undetected control of a user's local system.

Now suppose that the user attempts to initialize the local system state to content that s/he chooses; e.g., malware-free code, or I/O register values indicating that the system is disconnected from the Internet. Then, the user wants to verify that the system state, which may have been infected by malware and hence is untrusted, has been initialized to the chosen content.

Root of trust (RoT) establishment on an untrusted system ensures that a system state comprises *all* and *only* content chosen by the user, and the user's code *begins execution* in that state. *All* implies that no content is missing, and *only* 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 programs booted in a secure initial state can extend this state to include secondary storage and temporarily attached (e.g., USB) controllers. If RoT establishment fails, unaccounted for content, such as malware, exists. Hence, RoT establishment is sufficient for (stronger than) ensuring malware freedom and necessary for all software that needs secure initial states, such as access control and cryptographic software. However, as with secure and trusted boot, the trustworthiness of the software booted in secure initial states is not a RoT establishment concern.

Unconditional Security. In this work we establish RoT unconditionally; i.e., without secrets, trusted hardware modules and special instructions (e.g., TPMs [87], ROMs [22], [38], SGX [18]), or polynomial bounds on an adversary's computing power. By definition, a solution to a security or cryptography problem is unconditional if it depends only on the existence of physical randomness [13] and the ability to harvest it [37], [70]. Unconditional security solutions have several fundamental advantages over conditional ones. For example:

- they are independent of any security mechanism, protocol, or external party whose trustworthiness is uncertain; e.g., a mechanism that uses a secret key installed in hardware by a third party depends on the unknowable ability and interest of that party to protect key secrecy.
- they limit any adversary's chance of success to provably

A short version of this paper will be presented at the
Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23170
www.ndss-symposium.org

low probabilities determined by the defender; i.e., they give a defender undeniable advantage over the adversary.
- they are independent of the adversary's computing power and technology used; e.g., they are useful in post-quantum computing.

In unconditional RoT establishment all the user needs is an external source of non-secret physical randomness, such as one of the many commercially available quantum random number generators, and correct system specifications. That correct system specifications are indispensable for solving any security and cryptography problem has been recognized for a long time. As security folklore paraphrases a well-known quote [92]: "a system without specifications cannot be (in)secure: it can only be surprising." For RoT establishment, specifications are necessarily low-level: we need a concrete Word Random Access Machine (cWRAM) model of computation (viz., Appendix A), which is the closest model to a real computer system. It has a constant word length, up to two operands per instruction, and a general instruction-set architecture (ISA) that includes I/O operations and multiple addressing modes. It also supports multiprocessors, caches, and virtual memory.

Contributions and Roadmap. We know of no other protocols that establish RoT provably *and* unconditionally. Nor do we know any other software security problem that has been solved unconditionally in any realistic computational model. This paper is organized as follows.

Requirements Definition (Section II). We define the requirements for RoT establishment, and provide the intuition for how to jointly satisfy them to establish malware-free states and then RoT. In Section VIII we show that these requirements *differ* from those of past attestation protocols; i.e., some are stronger and others weaker than in past software-based [7], [47], [77], [78], [80], cryptographic-based [8], [22], [27], [38], [46], [64], and hybrid [53], [94] attestation protocols.

New Primitive for establishing malware-free states (Section IV). To support establishment of malware-free system states, we introduce a new computation primitive with optimal space (m)-time (t) bounds in adversarial evaluation on cWRAM, where the bounds can scale to larger values. The new primitive is a *randomized polynomial*, which has k-independent uniform coefficients in a prime order field. It also has stronger collision properties than a k-independent (almost) universal hash function when evaluated on cWRAM. We use randomized polynomials in a new verifier protocol that assures deterministic time measurement in practice (Section VI). Preliminary measurements (Section VII) show that their performance is practical on commodity hardware even for very large k; i.e., k = 64.

RoT establishment (Section V). Given malware-free states, we provably establish RoT and provide secure initial states for all software. This requirement has not been satisfied since its identification nearly three decades ago; e.g., see the NSA's Trusted Recovery Guideline [62], p. 19, of the TCSEC [61].

Optimal evaluation of polynomials (Section III). We use Horner's rule to prove concrete optimal bounds of randomized polynomials in the cWRAM. To do this, we prove that a Horner-rule program is *uniquely optimal* whenever the cWRAM execution space and time are simultaneously minimized. This result is of independent complexity interest since Horner's rule is uniquely optimal only in infinite fields [12] but is *not* optimal in finite fields [23], [43].

Appendix A provides a description of the cWRAM model. Appendix B contains the proofs of the theorems, lemma, and corollary of Section IV. Appendix C illustrates the implementation of the cWRAM encoding of Horner-rule steps in real processors, the selection of k for these processors, and practical ways to map strings of program words to strings of $\mathbf{Z_p}$ integers; i.e., to the inputs of randomized polynomials.

II. Requirements Definition

To define the requirements for RoT establishment we use a simple *untrusted* system connected to a *trusted* local verifier.

Suppose that the system has a processor with register set R and a random access memory M. The verifier asks the system to initialize M and R to chosen content. Then the verifier sends a random *nonce*, which selects C_{nonce} from a family of computations $\mathbf{C_{m,t}}(M, R)$ with space and time bounds m and t, and challenges the system to execute computation C_{nonce} on input (M, R) in m words and time t. Suppose that $\mathbf{C_{m,t}}$ is space-time (i.e., m-t) optimal, result $C_{nonce}(M, R)$ is unpredictable by an adversary, and C_{nonce} is non-interruptible. If $\mathbf{C_{m,t}}$ is also second pre-image free and the system outputs result $C_{nonce}(M, R)$ in time t[1], then after accounting for the local communication delay, the verifier concludes that the system state (M, R) contains all and only the chosen content. Intuitively, second pre-image freedom and m-t optimality can jointly prevent an adversary from using fewer than m words or less time than t, or both, and hence from leaving unaccounted for content (e.g., malware) or executing arbitrary code in the system.

When applied to multiple device controllers, the verifier's protocol must ensure that a controller cannot help another undetectably circumvent its bounds by executing some part of the latter's computation; e.g., act as an *on-board proxy* [53].

A. Adversary

Our adversary can exercise all known attacks that insert persistent malware into a computer system, including having brief access to that system to corrupt software and firmware; e.g., an extensible firmware interface (EFI) attack [63] 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, but does not have access to the verifier's device and external source of true random numbers.

For unconditional security, we assume that the adversary *can break* all complexity-based cryptography but *cannot* predict the true random numbers received from the verifier. Also, the adversary's malware can optimize $\mathbf{C_{m,t}}$'s code on-the-fly and at *no cost*; e.g., without being detected by the verifier. Furthermore, the adversary can output the result of a different computation that lowers t or m, or both, while attempting to return a correct $C_{nonce}(M, R)$ result.

[1]The verifier is trusted to obtain the correct result $C_{nonce}(M, R)$ from the execution of C_{nonce} on a trusted computer, or equivalently a trusted simulator of the trusted computer, having the same configuration as the untrusted system device. Also, optimal time bound t may vary among different computer systems in reality, and hence the trusted verifier obtains it from the trusted computer.

B. Code Optimality in Adversary Execution

Concrete-Optimality Background. Recall that a computation's upper time and space bounds are given by an algorithm for that computation whereas the lower bounds are given by a proof that holds for all possible algorithms for it. An algorithm is space-time optimal if its bounds match the space and time lower bounds of its computation.

Note that a verifier can use neither $\mathbf{C_{m,t}}$ computations that have *asymptotic* lower bounds nor ones that have only *theoretical* ones; i.e., bounds that cannot be matched by any program, as illustrated below. If $\mathbf{C_{m,t}}$'s lower bounds are asymptotic, a verifier can never prove that an adversary is unable to find an algorithm with better concrete bounds, by improving the constants hidden in the asymptotic characterizations. If the verifier measures the computation time against a theoretical lower bound, it returns 100% false positives and renders verification useless. If it measures time against a value that exceeds the theoretical lower bound, it can never prove that an adversary's code couldn't execute faster than the measured time, which renders verification meaningless. If the memory lower bound is theoretical and the adversary can exercise space-time (m-t) trade-offs, a time measurement dilemma may arise again: if m is scaled up to a practical value, t may drop to a theoretical one.

A verifier needs $\mathbf{C_{m,t}}$ algorithms with *concrete* (i.e., non-asymptotic) *space-time optimal* bounds in *realistic* models of computers; e.g., models of general ISAs, caches and virtual memory, and instruction execution that accounts for *I/O* and interrupts, multiprocessors, pipelining. If such algorithms are available, the only verifier challenge is to achieve precise space-time measurements, which is an engineering, rather than a basic computation complexity, problem; viz., Section VI. In practice, finding such $\mathbf{C_{m,t}}$ algorithms is far from a simple matter. For example, in Word Random Access Machine (WRAM) models, which are closest to real computers (e.g., Appendix A), the lower bounds of even simple computations such as static dictionaries are asymptotic even if tight [1], [60]. For more complex problems, such as polynomial evaluation, lower bounds in WRAM have been purely theoretical. That is, they have been proved in Yao's *cell (bit) probe* model [91], where only references to memory cells are counted, but *not* instruction execution time. Hence, a WRAM program can never match these lower bounds[2]; see *Related Work*, Section VIII.

Concretely optimal algorithms exist for some classic problems in computation models that are limited to very few operations; e.g., Horner's rule for polynomial evaluation. However, lower bounds in such models do not hold in a WRAM model with a general ISA or a real processor. For instance, lower bounds for integer *gcd* programs obtained using integer division (or *exact division* and *mod* [57]) can be lowered in modern processors where an integer division by a known constant can be performed much faster by integer multiplication [34], [39]; also a right shift can replace division by a power of two. Furthermore, a $\mathbf{C_{m,t}}$ program must retain its optimality when *composed* with system code; e.g., initialization and *I/O* code. Its lower bounds must not be invalidated by the composition.

Adversary execution. Most optimality results assume *honest* execution of $\mathbf{C_{m,t}}$ code. An execution is honest if the $\mathbf{C_{m,t}}$ code is fixed before it reads any variables or input $nonce$, and returns correct results for all inputs. Unfortunately, the optimality in honest execution does not necessarily hold in *adversarial* execution since an adversary can change $\mathbf{C_{m,t}}$'s code both before and after receiving the $nonce$, or simply guess the $C_{nonce}(M, R)$ result without executing any instructions. For example, the adversary can encode a small $nonce$ into immediate address fields of instructions to save register space and instruction execution. More insidiously, an adversary can change $\mathbf{C_{m,t}}$'s code and $nonce$ to that of $\mathbf{C'_{m',t'}}$ and $nonce'$ where $(C'_{nonce'}, M', R') \neq (C_{nonce}, M, R)$, such that $C'_{nonce'}(M', R') = C_{nonce}(M, R)$ and $t' < t, m' = m$ or $t' = t, m' < m$ or $t' < t, m' < m$. If the adversary can output correct result $C'_{nonce'}(M', R')$ with only low probability over the choices of $nonce$, we say that result $C_{nonce}(M, R)$ is *unpredictable*. Otherwise, the adversary wins.

The adversary can also take advantage of the optimal *code composition* with initialization and *I/O* programs. For instance, if the input of $\mathbf{C_{m,t}}$'s variables and $nonce$ requires multiple packets, the adversary can pre-process input in early packet arrivals and circumvent the lower time and/or space bounds; viz., end of Section III for an example. Also, in a multi-device system, a device can perform part of the computation of another device and help the latter undetectably circumvent its optimal bounds, as illustrated below.

C. Verifier Protocol Atomicity in Adversary Execution

The *verifier's protocol* begins with the input into the system and ends when the verifier checks the system's output; i.e., result-value correctness and timeliness. Protocol atomicity requires integrity of control flow across the instructions of the verifier's protocol with *each* system device; 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 [47], and inter-processor communication, can violate control-flow integrity *outside* of $\mathbf{C_{m,t}}$'s code execution. For instance, malware instructions in initialization code can post a future interrupt *before* the verifier's protocol begins execution. The interrupt could trigger *after* the correct and timely $C_{nonce}(M, R)$ result is sent to the verifier, and its handler could undetectably corrupt the system state [52]. Clearly, optimality of $\mathbf{C_{m,t}}$ code is insufficient for control-flow integrity. Nevertheless, it is necessary: otherwise, a predictable $C_{nonce}(M, R)$ result 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 the $\mathbf{C_{m,t}}$ code. Their execution inside $\mathbf{C_{m,t}}$ code would violate optimality bounds, and after $\mathbf{C_{m,t}}$ would be ineffective: asynchronous events could trigger during the execution of the last instruction. However, verification that an instruction is located before $\mathbf{C_{m,t}}$ code in memory (e.g., via computing digital signatures/MACs over the code) does *not* guarantee the instruction's execution. The adversary code could simply skip it before executing $\mathbf{C_{m,t}}$'s code. Hence, verification must address the apparent cyclic dependency: on the one hand, the execution of the event-disabling instructions before $\mathbf{C_{m,t}}$ code requires control-flow integrity, and on the other, control-flow integrity requires the execution of those instructions before $\mathbf{C_{m,t}}$ code.

Concurrent-transaction order and duration. Let a system comprise c connected devices, where device i has random

[2]The irrelevance of the cell-probe model *in practice* has not escaped complexity theorists: "the true model is the [W]ord RAM, but the bitprobe model is sometimes interesting as a mathematical curiosity [68]."

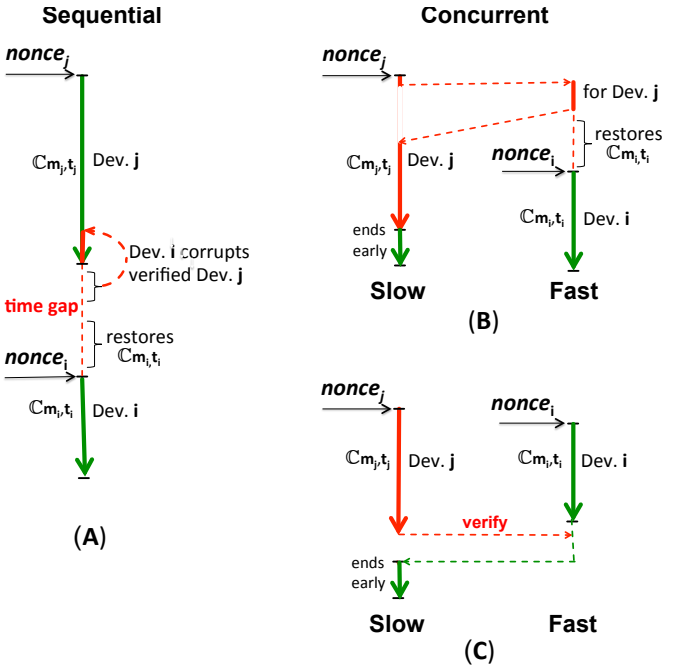

Fig. 1: Sequential and Concurrent Transaction Execution

access memory M_i and processor registers set R_i. Assume for the moment that space-time optimal $\mathbf{C_{m_1,t_1}}, \ldots, \mathbf{C_{m_c,t_c}}$ programs exist and that the control-flow integrity of the verifier's protocol is individually verifiable for each device i. Then the verifier protocol must be *transactional*: either *all* $C_{nonce_i}(M_i, R_i)$ result checks pass or the verification fails. In addition, it must prevent two security problems.

First, the protocol must prevent a time gap between the end of the $\mathbf{C_{m_j,t_j}}$'s execution and the beginning of $\mathbf{C_{m_i,t_i}}$'s, $i \neq j$. Otherwise, a time-of-check-to-time-of-use (TOCTTOU) problem arises. A malicious yet-to-be-verified device controller can perform an unmediated peer-to-peer I/O transfer [54], [55] to the registers of an already verified controller, corrupt system state, and then erase its I/O instruction from memory before its attestation begins[3]. This is shown in Figure 1(A) and implies that $\mathbf{C_{m_1,t_1}}, \ldots, \mathbf{C_{m_c,t_c}}$ must execute *concurrently*: none may end before another starts[4].

Second, the protocol must assure correct execution *order* and *duration* of $\mathbf{C_{m_i,t_i}}$ programs. That is, the difference between the start times and/or end times of any two programs $\mathbf{C_{m_i,t_i}}$ and $\mathbf{C_{m_j,t_j}}$ must be small enough so that neither device i nor j can undetectably perform any computation for the other enabling it to lower its bounds and circumvent attestation.

For instance, if the verifier challenges fast device i to start $\mathbf{C_{m_i,t_i}}$ a lot later than slower device j to start $\mathbf{C_{m_j,t_j}}$, device i can execute some of $\mathbf{C_{m_j,t_j}}$'s instructions faster, or even act as an *on-board proxy* [53], for j. Then device i can undetectably restore its correct $\mathbf{C_{m_i,t_i}}$ code before its challenge arrives. This is illustrated in Figure 1(B). Or, if $\mathbf{C_{m_i,t_i}}$ ends well before $\mathbf{C_{m_j,t_j}}$ ends, malicious device j can act as the verifier and fool attested device i into completing $\mathbf{C_{m_j,t_j}}$'s execution faster. This is illustrated in Figure 1(C). (Recall that, even if attested, devices cannot securely authenticate and distinguish unattested-device requests from verifier's requests and deny them.) Note that slower devices can also help faster ones lower their bounds. Nevertheless, the faster $\mathbf{C_{m_i,t_i}}$ to slower $\mathbf{C_{m_j,t_j}}$ execution order [53] helps ensure that start-time and end-time differences are small enough; e.g., see small differences δ_{start} and δ_{end} in Figure 2.

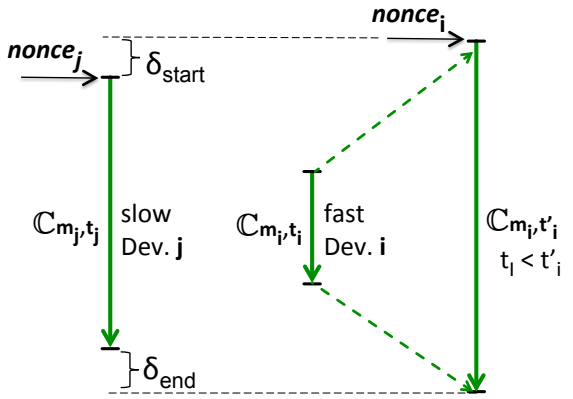

Fig. 2: Concurrent Execution with Bounds Scaling

Scalable bounds. Given an optimal $\mathbf{C_{m,t}}$ program, one must be able to obtain other optimal $\mathbf{C_{m_i,t_i}}$ programs from it, where $m_i > m$, $t_i > t$. Furthermore, given an optimal $\mathbf{C_{m_i,t_i}}$ program for a fast device i, one must be able to obtain an optimal program $\mathbf{C_{m_i,t'_i}}$ for it, where time bound $t'_i \geq \max(t_i)$, $i = 1, \ldots, c$, *independent* of m_i; see Figure 2. This is intended to prevent the on-board proxy attacks as described above. Neither scaling is obvious.

For example, an intuitive scaling of $\mathbf{C_{m,t}}$ to $\mathbf{C_{m_i,t_i}}$ might copy $\mathbf{C_{m,t}}$ code $k \geq \lceil m_i/m \rceil$ times in M_i and then challenge the k optimal copies sequentially; and the scaling from $\mathbf{C_{m_i,t_i}}$ to $\mathbf{C_{m_i,t'_i}}$ might execute $\mathbf{C_{m_i,t_i}}$ on $k' \geq \lceil t'_i/t_i \rceil$ *nonces*. Neither achieves optimal bounds in adversary execution. Consider the second scaling; the first has similar drawbacks. The k' executions $C_{nonce_0}(M_i, R_i), \ldots, C_{nonce_{k'-1}}(M_i, R_i)$ must be linked to avoid exploitable time gaps, as noted above. If linking is done by the verifier, $C_{nonce_j}(M_i, R_i)$'s code cannot end its execution until it inputs the next nonce, $nonce_{j+1}$, from the verifier [53]. Then $\mathbf{C_{m_i,t_i}}$ can no longer be optimal, since the variable input-synchronization delays in $\mathbf{C_{m_i,t_i}}$ invalidate the optimal t_i[5]. If the synchronization buffers $nonce_{j+1}$, optimal m_i also becomes invalid. The alternate linking whereby $nonce_{j+1} = C_{nonce_j}(M_i, R_i)$ is inadequate since nonces are no longer random, or even pseudo-random [47], [77].

Figure 3 summarizes the relationships among requirements for RoT establishment on an untrusted system. The engineering requirements for *time-measurement security* and a new mechanism that satisfies them are presented in Section VI.

[3]Powering off all *stateful* devices and then powering them individually to perform one-at-a-time verification is inadequate because some (e.g., chipset) devices cannot be powered-off without system shutdown. The TOCTTOU problem is still *not* solved because malicious firmware can still corrupt already verified devices in the time gap between device power-on and attestation start.

[4]TOCTTOU attacks generalize to remote attestation in networks [22], [27]: a yet-to-be-attested host can reinfect an already-attested host and then reboot itself to a clean software copy before its attestation. Reinfection is possible because attestation does not guarantee correctness of the attested software [67]. *Duqu* 2 illustrates similar malware mobility [42].

[5]Synchronization delays for $nonce_{j+1}$ input in a $checksum_j$ computation on a network interface card (Netgear GA 620) can be as high as $0.4t$ with a standard deviation of about $0.0029t$; see [53], Sections 5.4.2-5.4.4.

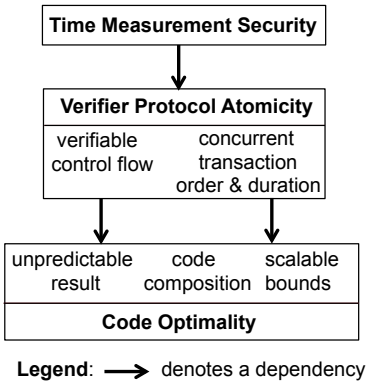

Fig. 3: RoT Establishment on an Untrusted System

D. Satisfying the requirements – Solution Overview

Individually, the two properties presented below are necessary but insufficient to satisfy the $C_{m,t}$ requirements. However, jointly they do satisfy all of them.

1. *k-independent (almost) universal hash functions*. The soundness of the verifier's result-value check requires that $\mathbf{C_{m,t}}$ is second pre-image free in a one-time evaluation. That is, no adversary can find memory or register words whose contents differ from the verifier's choice and pass its check, except with probability close to a random guess over *nonces*. Also, inputting the $\mathbf{C_{m,t}}$ variables and $nonce$ into an untrusted device must use a small constant amount of storage. k-independent (almost) universal hash functions based on polynomials satisfy both requirements. Their memory size is constant for constant k [16], [65] and they are second pre-image free. We introduce the notion of *randomized polynomials* to construct such functions for inputs of $d+1$ log p-bit words independent of k; i.e., degree d polynomials over $\mathbf{Z_p}$ with k-independent, uniformly distributed coefficients; see the Corollary in Section IV-D.

2. *Optimal polynomial evaluation*. The soundness of the verifier's result-timeliness check requires a stronger property than second pre-image freedom. That is, no computation $C'_{m',t'}$ and $nonce'$ exists such that $C'_{nonce'}(M', R') = C_{nonce}(M, R)$ and either one of its bounds, or both, are lower than $\mathbf{C_{m,t}}$'s in a one-time cWRAM evaluation, except with probability close to a random guess over $nonces$. Concrete space-time optimality of randomized polynomials in adversary evaluation on cWRAM yields this property; viz., Section IV-D. Its proof is ultimately based on a condition under which a Horner-rule program for polynomial evaluation is uniquely optimal in an honest one-time cWRAM evaluation; see Theorem 1 below.

Why are these combined properties sufficient for RoT establishment? Randomized polynomials enable a verifier to check the *integrity of control flow* in the code it initializes on an untrusted cWRAM device (Theorem 6). In turn, this helps implement *time-measurement security*; viz., Section VI. They also assure bounds scalability[6], which enables the verifier to satisfy the transaction *order* and *duration* requirement and leads to the establishment of malware-free states on a multi-device system (Theorem 7). Finally, the verifier uses ordinary universal hash functions to establish RoT in malware-free states (Theorem 8).

[6] k-independent universal hash functions with *constant time* bounds and very good *space-time trade-offs* exist in a standard WRAM model [17]. However, these bounds aren't (concretely) optimal and don't allow *independent* time-bound scalability. Hence these functions are impractical for this application.

III. Foundation: Optimal Polynomial Evaluation

In this section we provide the condition under which a Horner-rule program for polynomial-evaluation is uniquely optimal in the concrete WRAM (cWRAM) model, which we use for proving the optimality of randomized-polynomial evaluation in Section IV. We begin with a brief overview of the cWRAM model and illustrate the challenges of proving optimality of universal hash functions in it. A detailed description of cWRAM is in Appendix A.

A. Overview of the cWRAM model

The cWRAM model is a concrete variant of Miltersen's *practical RAM* model [60]; i.e., it has a *constant* word length and at most *two operands* per instruction. It also extends the *practical RAM* with higher-complexity instructions (e.g., mod, multiplication), as well as *I/O* instructions, special registers (e.g., for interrupt and device status), and an execution model that accounts for interrupts. The cWRAM includes all known register-to-register, register-to-memory, and branching instructions of real system ISAs, as well as all integer, logic, and shift/rotate computation instructions. In fact, any computation function implemented by a cWRAM instruction is a *finite-state transducer*; see Appendix A. (The limit of two operands per instruction is convenient, not fundamental: instructions with higher operand *arity* only complicate optimality proofs.) All cWRAM instructions execute in *unit time*. However, floating-point instructions are not in cWRAM because, for the same data size, they are typically twice as slow as the corresponding integer instructions in latency-bound computations; i.e, when one instruction depends on the results of the previous one, as in the Horner-rule step below. Thus they cannot lower the concrete space-time bounds of our integer computations. Likewise, non-computation instructions left out of cWRAM are irrelevant for our application.

Like all real processors, the cWRAM has a fixed number of registers with distinguished names and a memory that comprises a finite sequence of words indexed by an integer. Operand addressing in memory is immediate, direct and indirect, and operands comprise words and bit fields.

B. Proving optimality of universal hash functions in cWRAM

The immediate consequence of the constant word length and limit of two single-word operands per instruction is that any instruction-complexity hierarchy based on *variable* circuit fan-in/fan-out and depth collapses. Hence, lower bounds established in WRAM models with *variable* word length and number of input operands [1], [60], [65] and in branching-program models [56] are irrelevant in cWRAM. For example, lower bounds for universal hash functions show the necessity of executing multiplication instructions [1], [56]. Not only is this result unusable in cWRAM, but proving the necessity of any instruction is made harder by the requirement of unit-time execution for all instructions.

In contrast, concrete space-time lower bounds of cryptographic hash functions built using circuits with *constant* fan-in, fan-out, and depth [3], [4] would be relevant to cWRAM computations. However, these bounds would have to hold in adversary execution, which is a significant challenge, as seen in Section II-B. Even if such bounds are eventually found,

these constructions allow only bounded adversaries and hence would not satisfy our goal of unconditional security.

Since we use polynomials to construct k-independent (almost) universal hash functions, we must prove their concrete optimality in cWRAM evaluations. However, all concrete optimality results for polynomial evaluation are known *only* over infinite (e.g., rational) fields [12], and the gap between these bounds and the lower bounds over finite fields (e.g., $\mathbf{Z_p}$) is very large [43]. Furthermore, optimality is obtained using only two operations (i.e., $+, \times$) and cannot hold in computation models with large instruction sets like the cWRAM and real processors. We address these problems by adopting a complexity measure based on *function locality* [60], which enables us to distinguish between classes of unit-time computation instructions, and by providing an evaluation condition that extends the unique optimality of Horner's rule to cWRAM.

C. Unique optimality of Horner's rule in cWRAM

Horner's Rule. Let p be a prime. Polynomial $P_d(x) = a_d \times x^d + a_{d-1} \times x^{d-1} + \cdots + a_1 \times x + a_0 \ (mod\ p)$ is evaluated by Horner's rule in a finite field of order p as $P_d(x) = (\cdots (a_d \times x + a_{d-1}) \times x + \cdots + a_1) \times x + a_0 \ (mod\ p)$.

Horner-rule step and *programs.* A program that evaluates $a_i \times x + a_{i-1} \ (mod\ p)$ as a sequence of four instructions *integer multiplication (·)*, $mod\ p$, *integer addition (add)*, $mod\ p$ in cWRAM, or $mod(add(mod(\cdot(a_i, x), p), a_{i-1}), p)$ in infix notation, is called the Horner-rule *step*. If arithmetic is in $mod\ 2^{w-1}$ where $w-1$ bits represent an unsigned integer value of a w-bit word, the Horner-rule step[7] simplifies to the multiply-add sequence; i.e., $add(\cdot(a_i, x), a_{i-1})$.

A cWRAM loop that executes a Horner-rule step d times to evaluate $P_d(x)$ is a Horner-rule *program*. Note that there may be multiple encodings of a Horner-rule program that evaluate $P_d(x)$ in the same space and time.

Initialization. If $p < 2^{w-1}$, the initialization of a Horner-rule program is a cWRAM structure of $d + 11$ storage words comprising 6 instructions (i.e., 4 for a Horner-rule step and 2 for loop control) and $d + 5$ data words; i.e., d, $a_i (0 \le i \le d)$, x, p, and output z. If arithmetic is $mod\ 2^{w-1}$, the structure has $d + 8$ storage words; i.e., 4 instructions and $d + 4$ words.

One time, honest evaluation. Polynomial $P_d(x)$ is evaluated *one time* if nothing is known about coefficients a_i and input x before the evaluation starts; i.e., a_i and x are variables. The evaluation of $P_d(x)$ is *honest* if its program code is fixed before constants are assigned to variables a_i and x (i.e., before constants are input and initialized in cWRAM memory) and returns correct results for all inputs. In a *dishonest* (e.g., adversarial) evaluation, program code can be changed after x or any a_i become known; e.g., if $x = 0$, $P_d(0) = a_0$ can be output without code execution.

Theorem 1. Let $w > 3$ be an integer, $2 < p < 2^{w-1}$ a prime, and $P_d(\cdot) = \sum_{i=0}^{d} a_i \times x^i \ (mod\ p)$ a polynomial over $\mathbf{Z_p}$. The honest one-time evaluation of $P_d(x)$ by a Horner-rule program is *uniquely space-time optimal* whenever the cWRAM execution time and memory are simultaneously minimized; i.e., no other programs can use fewer than both $d + 11$ storage words and $6d$ time units after initialization.

[7] In many processors, this is implemented by a single *three-operand* multiply-accumulate instruction.

The proof of this theorem and of all others are in *Appendix B*. Briefly, since a Horner-rule program provides the upper bounds, we only need to prove the lower bounds that match them in cWRAM. To prove the lower bounds, we use finite field properties, linear polynomials over $\mathbf{Z_p}$, locality-based cWRAM instruction complexity, and the two-operand per instruction limit. First, we show that a four-instruction Horner-rule step is optimal when the cWRAM evaluation space and time for linear polynomials are simultaneously minimized. Then, we use the facts that the evaluation is one-time and honest to show that a Horner-rule step is uniquely optimal. Finally, we define a polynomial of degree d as a special composition of linear polynomials, and show that its evaluation requires a unique two-instruction loop-control sequence that must iterate d times over the Horner-rule step.

A similar proof holds over $\mathbf{F_q}$ when $q > 2$ is a prime power. To illustrate, we outline it for the important case $q = 2^{w-1}$. Here the Horner-rule program needs only $d + 8$ words and $4d$ time units after initialization.

Theorem 1 answers A. M. Ostrowski's 1954 questions regarding the optimality of Horner's rule [12] in a realistic model of computation. However, both bounds $t = 6d$ and $m = d+11$ depend on d, and thus t cannot scale independently of m. If t needs to be large, d becomes large. Hence not all $d+1$ coefficients of P_d could always be input at the same time; e.g., in one packet. This would enable an adversary's code to pre-process the coefficients that arrive early and circumvent the optimal bounds; e.g., with pre-processing, the lower bound for P_d's evaluation drops from d to $(d+1)/2$ multiplications [72].

IV. Randomized Polynomials and Malware Freedom

In this section we define a family of *randomized polynomials*, prove their space-time optimality in adversary evaluation on cWRAM (Theorem 5), and show that they have stronger collision-freedom properties than k-independent (almost) universal hash functions in cWRAM (Corollary). These properties enable the verifier to establish control-flow integrity on a single device (Theorem 6), and scale bounds for correct transaction order and duration in a multi-device untrusted system. This helps establish malware-free states (Theorem 7).

A. Randomized Polynomials – Definition

Let p be prime and $d > 0$, $k > 1$ integers. A degree-d polynomial over $\mathbf{Z_p}$ with k-independent (e.g., [16]), uniformly distributed coefficients s_i

$$P_{d,k}(\cdot) = s_d \times x^d + \cdots + s_1 \times x + s_0 \ (mod\ p),$$

is called the (d, k)-*randomized polynomial* [8].

If $v_d, \ldots, v_0 \in \mathbf{Z_p}$ are constants independent of s_i and x, and $\oplus$ is the bitwise *exclusive-or* operation, then polynomial

$$H_{d,k}(\cdot) = (v_d \oplus s_d) \times x^d + \cdots + (v_1 \oplus s_1) \times x + (v_0 \oplus s_0) \ (mod\ p)$$

is called the *padded*[9] randomized polynomial.

Each padding constant v_i will be used to represent the least significant $log\ p$ bits of a memory word i or of a special

[8] Our notion of *randomized polynomial* differs from Tarui's [85] as we cannot input variable numbers (i.e., $d + 1$) of random coefficients.

[9] Of course, other padding schemes not based on the $\oplus$ operation exist, which preserve the k-wise independence and uniform distribution of the padded coefficients.

processor-state register; whereas the k of random numbers (which generate the s_i) will fill the least significant $log\ p$ bits of all general-purpose processor registers; e.g., see the device initialization in Section IV-E1 below.

Theorem 2 below shows that $H_{d,k}(\cdot)$ is second pre-image free, has uniform output, and is k-independent. Everywhere below, $\xleftarrow{\$}$ denotes a uniform random sample.

Theorem 2. Let $p > 2$ be a prime and $u \in \mathbf{Z_p}$ a constant.

1. $Pr[x \xleftarrow{\$} \mathbf{Z_p}, \exists y \in \mathbf{Z_p}, y \neq x : H_{d,k}(y) = H_{d,k}(x)]$
$\leq \frac{1}{(p-1)}$
2. $Pr[x \xleftarrow{\$} \mathbf{Z_p} : H_{d,k}(x) = u] = \frac{1}{p}$
3. $H_{d,k}(\cdot)$ is k-independent.

The proofs of parts 1 and 2 follow from two notable facts. First, let x, m be positive integers. If $gcd(x, m) = 1$, then equation $\alpha \cdot x = y\ (mod\ m)$ has a unique solution $mod\ m$. Hence, for all $y \in \mathbf{Z_p}$ and $x \in \mathbf{Z_p^+}$ there exists a unique α such that $y = \alpha \cdot x\ mod\ p$, and thus $H_{d,k}(y) - H_{d,k}(x)$ becomes a univariate polynomial in x. Second, any univariate polynomial over $\mathbf{Z_p}$ whose free coefficient is uniformly distributed and independent of input x has uniform output in $\mathbf{Z_p}$ when evaluated on a uniform, random x. For part 3, we evaluate $H_{d,k}(\cdot)$ at k distinct points and obtain a system of k linear equations with $d+1$ unknowns $v_i \oplus s_i$, k of which are independent. We fix any $d-k+1$ unknowns, evaluate their terms, and obtain a system of k linear equations that has a unique solution. Now the independence result follows by definition [90].

Below we define the k-independent uniform elements s_i for a family of randomized polynomials $\mathbf{H}$ in the traditional way [16], [90]. We use family $\mathbf{H}$ in the rest of this paper.

Family H. Let $p > 2$ be a prime and $r_j, x \xleftarrow{\$} \mathbf{Z_p}$. Let $v = v_d, \ldots, v_0$, $v_i \in \mathbf{Z_p}$, be a string of constants independent of r_j and x. Family $\mathbf{H}$ is indexed by tuples $(d, r_0, \ldots, r_{k-1}, x)$ denoted as (d, k, x) below.

$$\mathbf{H} = \{H_{d,k,x}(\cdot) \mid H_{d,k,x}(v) = \sum_{i=0}^{d}(v_i \oplus s_i) \times x^i\ (mod\ p),$$
$$s_i = \sum_{j=0}^{k-1} r_j \times (i+1)^j\ (mod\ p)\},$$

where $v_i \oplus s_i$ is represented by a $mod\ p$ integer.

Note that $H_{d,k,x}(\cdot) \in \mathbf{H}$ has properties 1 and 2 of $H_{d,k}(\cdot)$ in Theorem 2 in a *one-time* evaluation on $x \xleftarrow{\$} \mathbf{Z_p}$. The proof of its k-independence is similar to that of part 3.

Notation. For the balance of this paper, p is the *largest prime less than* 2^{w-1}, $w > 4$. The choices made for the random uniform selection of *nonce* $H_{d,k,x} \xleftarrow{\$} \mathbf{H}$ are denoted by $S = \{r_j, x \xleftarrow{\$} \mathbf{Z_p}, 0 \leq j \leq k-1\}$.

B. Code optimality in honest evaluation

In this section, we prove the optimal space-time bounds in a *honest* one-time evaluation of $H_{d,k,x}(\cdot)$. The only reason we do this is to set the bounds an adversary must aim to beat.

Let $Horner(H_{d,k,x}(\cdot))$ denote a *Horner-rule program* for the honest one-time evaluation of $H_{d,k,x}(\cdot) \in \mathbf{H}$ on input string v. That is, $Horner(H_{d,k,x}(\cdot))$ is implemented by a nested cWRAM loop using the recursive formula $z_{i-1} = z_i \times x + (v_{i-1} \oplus s_{i-1})$, where $z_d = v_d \oplus s_d$, $z_0 = H_{d,k,x}(v)$, $1 < i \leq d$. (We omit the correctness proof of the Horner loop invariant since it's a simple exercise.) Both the outer loop $\sum_{i=0}^{d}(v_i \oplus s_i) \times x^i\ (mod\ p)$ and the inner loop $s_i = \sum_{j=0}^{k-1} r_j \times (i+1)^j\ (mod\ p)$ are Horner-rule programs.

Upper bounds. We show that the upper bounds are $m = k + 22$ storage words and $t = (6k-4)6d$ execution time units in cWRAM, after variable initialization. By Theorem 1, the inner and the outer loops of $Horner(H_{d,k,x}(\cdot))$ can be implemented by 6 instructions each. For each of coefficient, $v_i \oplus s_i$, 2 instructions are sufficient whenever word indexing in v is sequential; i.e, an *addition* for indexing in v and an *exclusive-or*. The *addition* is sufficient when $d+1 \leq |v|$, where $|v|$ is the number of words comprising memory M and the special processor registers. If $d + 1 > |v|$, indexing in v also requires a $mod\ |v|$ instruction.

Modular indexing in v increases the instruction bound by 1 but does not affect the concrete optimality proofs since fewer instructions cannot simulate memory addressing in cWRAM. Furthermore, indexing to access a special processor register (e.g., asynchronous event status bits) contained in v assumes that the register is mapped in physical memory. When it isn't, accessing it via its index in v would require a couple of extra instructions. Again, these instructions would only insignificantly increase the memory and time bounds, but not affect their optimality. Thus, for simplicity of exposition and without loss of generality, we assume coefficient padding requires only 2 instructions. Hence, 14 instructions comprising 2 nested 6-instruction loops and the 2 instructions for computing a coefficient $v_i \oplus s_i$ are sufficient. Thus, $Horner(H_{d,k,x}(\cdot))$'s time bound is $t = [6(k-1)+2]6d = (6k-4)6d$ time units.

By the definition of family $\mathbf{H}$, the operands of these instructions are evident; i.e., $k + 8$ data words comprising the $H_{d,k,x}(\cdot)$'s index in $\mathbf{H}$, namely $(d, r_0, \ldots, r_{k-1}, x)$, degree $k-1$, index $i+1$, coefficient s_i, modulus p, output $z = v_d \oplus s_d$, and v_i's word index in v. Thus $k + 8$ data words and 14 instruction words, or $k+22$ (general-purpose processor register and memory) words, is $Horner(H_{d,k,x}(\cdot))$'s space bound.

Lower bounds. The upper space-time bounds of $H_{d,k,x}(\cdot)$ are unaffected by the excess memory and register space required by the programs for processor-state (i.e., special processor register) initialization, I/O, and general-purpose register initialization ($Init$) in cWRAM; see Section IV-E1. However, excess space prevents us from using Theorem 1 to prove the lower bounds since the execution space is no longer minimized. To avoid this technical problem, we assume these programs are space-optimal and memory M contains only the additional $k+22$ words. We also take advantage of the fact that an honest program does not surreptitiously modify the settings of the special processor registers after its code is committed. The above assumption is only used to simplify the concrete-optimality proof for the honest evaluation of $H_{d,k,x}(\cdot)$. It is *unnecessary* for the optimality proof of $Horner(H_{d,k,x}(\cdot))$ code in adversarial evaluation; see Section IV-C. There we use the collision-freedom properties of $H_{d,k,x}(\cdot)$ in cWRAM (e.g., Corollary, Section IV-D) and its uniform distribution of output, which we can avoid here thanks to the assumption made.

Theorem 3 (Optimality in Honest Evaluation). Let M comprise space-optimal processor-state initialization, I/O, and

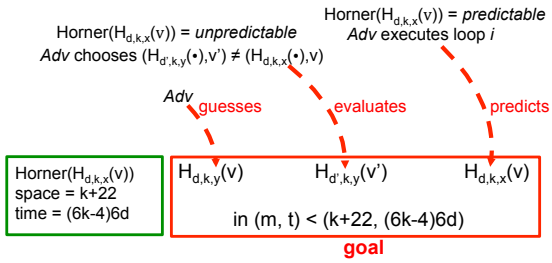

Fig. 4: Adversary goal and strategy space

$Init$ code, and $k+22$ words. The honest one-time evaluation of $H_{d,k,x}(\cdot)$ on v by $Horner(H_{d,k,x}(\cdot))$ is optimal whenever the cWRAM execution time and memory are simultaneously minimized; i.e., no other programs can use both fewer than $k+22$ storage words and $(6k-4)6d$ time units after initialization.

The proof of this theorem follows from Theorem 1, k-independence, and honest one-time evaluation.

C. Code optimality in adversary evaluation

Adversary Goal. By Theorem 3, the adversary's goal is to output $H_{d,k,x}(v)$ using only m words of storage and time t such that at least one of the lower bounds is lowered; i.e., $m < k+22$ and $t = (6k-4)6d$, or $m = k+22$ and $t < (6k-4)6d$, or $m < k+22$ and $t < (6k-4)6d$. We denote this goal by $(m,t) < (k+22, (6k-4)6d)$.

Strategy Space. We partition the adversary's strategy space into mutually exclusive cases *1 - 3* below, which s/he can select at no cost, and bound the probability of success in each case. These cases are summarized in Figure 4.

1. Guess $H_{d,k,x}(v)$. The adversary predicts $H_{d,k,x}(v)$ independent of *nonce* $H_{d,k,x} \xleftarrow{\$} \mathbf{H}$ and v; i.e., the prediction is a constant relative to the random choices made in $H_{d,k,x} \xleftarrow{\$} \mathbf{H}$. Hence, the probability of adversary's success in a one-time evaluation within bounds (m,t) is $\frac{1}{p}$, by Theorem 2-2.

For any evaluation that depends on *nonce* $H_{d,k,x} \xleftarrow{\$} \mathbf{H}$, the adversary must execute a program which inputs at least one of the random choices y made by $H_{d,k,x} \xleftarrow{\$} \mathbf{H}$, or a function thereof; e.g., $y \in \{x, r_j, 0 \le j < k\}$, $y = s_d$. The choice of program instructions depends on whether $Horner(H_{d,k,x}(v))$ is predictable: either the adversary executes at least one Horner-rule step (i.e., at least one outer loop execution) and then outputs the result prediction or s/he executes an entirely different instruction sequence in (m,t).

2. $Horner(H_{d,k,x}(v))$ is unpredictable. In this case, the adversary does not execute any Horner-rule step. Instead, s/he chooses a sequence of cWRAM instructions which inputs at least a $y \in \mathbf{Z_p}$ that depends on *nonce* $H_{d,k,x} \xleftarrow{\$} \mathbf{H}$, executes the sequence, and outputs its result in $\mathbf{Z_p}$. That is, the chosen sequence evaluates a function $f_H(\cdot) : \mathbf{Z_p} \to \mathbf{Z_p}$ on an input y and outputs $f_H(y)$ in (m,t). Its instructions may read and write multiple values in $\mathbf{Z_p}$; e.g., they may read and modify the values of the general-purpose processor registers, and/or those of v. Since $H_{d,k,x}(v)$ is unknown before $f_H(y)$ is output, the adversary's success depends on whether $f_H(y) = H_{d,k,x}(v)$.

Note that the execution of *any* instruction sequence with input and output in $\mathbf{Z_p}$ represents the evaluation of a unique polynomial $Q_{d'}(\cdot)$ of degree $d' \le p-1$ on some input y over $\mathbf{Z_p}$. This follows from a well-known fact that establishes the one-to-one correspondence between functions and polynomials $Q_{d'}(\cdot)$ in finite fields[10]. Hence, the adversary can *always* find a pair $(Q_{d'}(\cdot), y) \neq (H_{d,k,x}(\cdot), v)$ whose cWRAM evaluation has desired bounds $(m,t) < (k+22, (6k-4)6d)$. To upper bound the probability of adversary's success, we write $Q_{d'}(\cdot)$'s coefficients a_i $(0 \le i \le d')$ as $a_i = s_i \oplus v'_i$ for some $v' = v'_{d'}, \ldots, v'_0$, where $s_i = \sum_{j=0}^{k-1} r_j \times (i+1)^j \ (mod\ p)$ and $r_j \xleftarrow{\$} \mathbf{Z_p}$ are the same values used to generate $H_{d,k,x}(\cdot)$'s coefficients. If for any index $i < d'$ coefficient $a_i = 0$, we set $v'_i = s_i$. Thus, $Q_d(y) \equiv H_{d',k,y}(v')$ and $(H_{d',k,y}(\cdot), v') \neq (H_{d,k,x}(\cdot), v)$.

Let $Adv(H_{d,k,x}(\cdot), v) = H_{d',k,y}(v')$ denote the adversary's choice of polynomial, input v', and evaluation result output in (m,t). We denote event $[S, Adv(H_{d,k,x}(\cdot), v) = H_{d',k,y}(v') : H_{d',k,y}(v') = H_{d,k,x}(v) \mid (m,t)]$ succinctly by $[S : H_{d',k,y}(v') = H_{d,k,x}(v)]$. Lemma 4 bounds the adversary's probability of success, $Pr[S : H_{d',k,y}(v') = H_{d,k,x}(v)]$.

Lemma 4. Let $H_{d,k,x} \xleftarrow{\$} \mathbf{H}$ and v be its input. For any one-time choice of $(H_{d',k,y}(\cdot), v') \neq (H_{d,k,x}(\cdot), v)$, let the adversary output $H_{d',k,y}(v')$ in $(m,t) < (k+22, (6k-4)6d)$. Then $Pr[S : H_{d',k,y}(v') = H_{d,k,x}(v)] \le \frac{3}{p}$.

To prove this lemma, we partition all adversary's one-time choices of $(H_{d',k,y}(\cdot), v')$ into mutually exclusive attack events, given *nonce* $H_{d,k,x}(\cdot) \xleftarrow{\$} \mathbf{H}$ and v. Then we use the definition of family $\mathbf{H}$, Theorem 2, and two notable facts. The first is that $\mathbf{Z_p^+}$ is closed under multiplication. The second is the first fact used in the proof of Theorem 2 above.

3. $Horner(H_{d,k,x}(v))$ is predictable. Alternatively, the adversary decides that, for the given *nonce* $H_{d,k,x}(\cdot) \xleftarrow{\$} \mathbf{H}$ and v, $Horner(H_{d,k,x}(v))$ can be predicted within bounds (m,t). Hence, s/he executes at least one Horner-rule step (i.e., at least one outer loop of $Horner(H_{d,k,x}(v))$ and then predicts $H_{d,k,x}(v)$ without completing $Horner(H_{d,k,x}(v))$. The bounds goal (m,t) is met: $m \le k+22$ and $t < (6k-4)6d$.

Let us denote the event of adversary's success by $[S : Adv(Horner(H_{d,k,x}(\cdot), v) = H_{d,k,x}(v) \mid (m,t)]$, or succinctly by $[S : H_{d,k,x}(v) = predictable]$.

In the proof of Theorem 5 below we show that the bound of $Pr[S : H_{d,k,x}(v) = predictable]$ is not higher than the bound in Lemma 4. This theorem shows that the concrete optimality requirements of Section II-B are satisfied for a $Horner(\cdot)$ program with bounds $m = k+22$, $t = (6k-4)6d$, which is invoked with $nonce = H_{d,k,x}(\cdot) \xleftarrow{\$} \mathbf{H}$ on input v. This is summarized in Figure 5.

Theorem 5 (Optimality in Adversary Evaluation).

In a one-time cWRAM evaluation of $H_{d,k,x}(\cdot) \xleftarrow{\$} \mathbf{H}$ on v an adversary can lower either the space or the time bound of $Horner(H_{d,k,x}(v))$, or both, with probability at most $\frac{3}{p}$.

The proof of this theorem follows from the definition of result (un)predictability in a one-time evaluation, Theorem 2 and the second notable fact used in its proof, Lemma 4, and the definition of family $\mathbf{H}$.

[10] By L. E. Dickson (1897) and C. Hermite (1854); viz., *Lemma 1.1-1.2* [79].

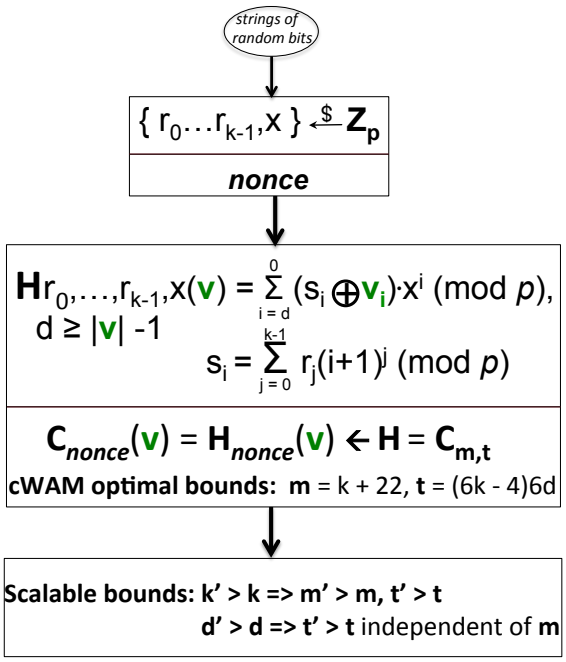

Fig. 5: Optimality of $\mathbf{C_{m,t}}$ with Family H

D. Collision Freedom of H in cWRAM

The corollary below shows not only that $\mathbf{H}$ is a family of k-independent (almost) universal hash functions, but also that an adversary is unable to find a function in $\mathbf{Z_p}$ whose one-time cWRAM evaluation on an input y collides with $H_{d,k,x}(v)$ within bounds $(m,t) < k+22, (6k-4)6d$.

Corollary.
1. $\mathbf{H}$ is a k-independent (almost) universal hash function family.
2. Let $(m,t) < (k+22,(6k-4)6d)$. For a given one-time evaluation of $H_{d,k,x}(\cdot) \xleftarrow{\$} \mathbf{H}$ on input v in cWRAM, $Pr[H_{d,k,x} \xleftarrow{\$} \mathbf{H}, v, \exists f, y \in \mathbf{Z_p} : f(y) = H_{d,k,x}(v) \mid (m,t)] \leq \frac{3}{p}$.

Part 1 follows by a similar proof as in Lemma 4, and the k-independence follows along the same lines as the proof of Theorem 2-3. Part 2 follows directly from Theorem 5.

E. Device Initialization and Atomicity of Verifier's Protocol

1) Device Initialization: Upon system boot, the verifier requests each device's boot loader (e.g., akin to U-boot in Section VII) to initialize the device memory with the verifier's chosen content, as described in steps *(i)* – *(v)* below, and then transfer control to the first instruction of the processor-state initialization program. The boot loaders may not contain all and only the verifier's chosen code, and hence are untrusted.

i) Processor-state initialization. This is a straight-line program that accesses special processor registers to:

- disable all asynchronous events; e.g., interrupts, traps, breakpoints;
- disable/clear caches, disable virtual memory, TLBs[11], and power off/disable stateless devices;

[11]Disabling/clearing caches/TLBs prevents an adversary from loading chosen content *before* the timed protocol starts and circumvent time measurements; viz., Sections VI-B and VII.

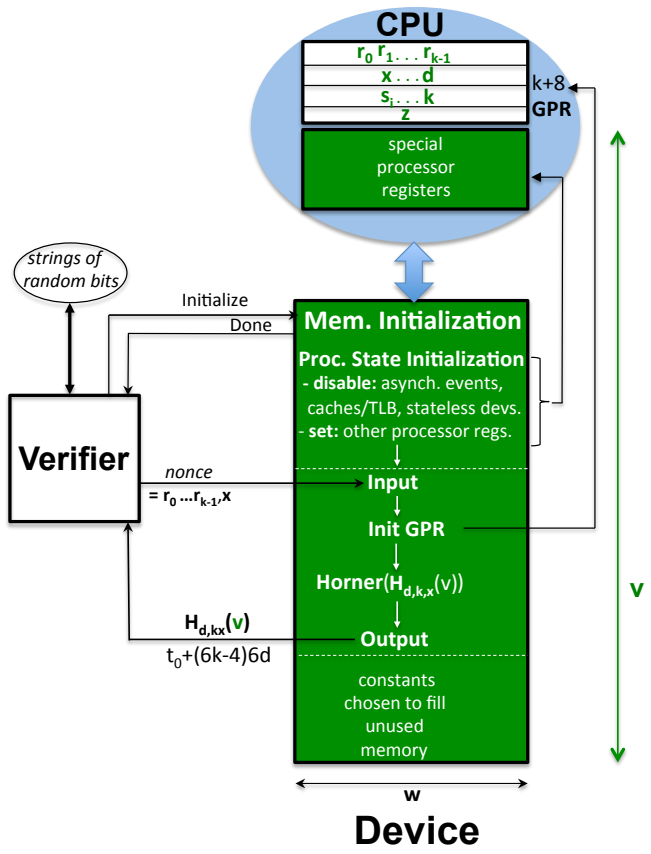

Fig. 6: Device Initialization and Verifier Protocol Execution

- set all remaining state registers to chosen values; e.g., clock frequency, *I/O* registers.

When execution ends, the *Input* program follows in straight line.

ii) Input/Output programs. The *Input* program busy-waits on the verifier's channel device for input. Once *nonce* $H_{d,k,x}$ arrives, the *Init* program follows in straight line. The *Output* program sends result $H_{d,k,x}(v)$ to the verifier after which it returns to busy-waiting in the boot loader for further verifier input.

iii) Init program. This is a straight-line program that loads the k random values of *nonce* $H_{d,k,x}$ into the general-purpose processor registers so that no register is left unused; e.g., if 16 registers are available, $k = 16$. Its execution time, t_0, is constant since k is constant. When execution ends, the $Horner(H_{d,k,x}(\cdot))$ program follows in straight line.

iv) $Horner(H_{d,k,x}(\cdot))$ *program*. This comprises 14 instructions whenever the address space is linear in physical memory. When execution ends, the *Output* program follows in straight line and outputs $H_{d,k,x}(v)$.

v) Unused-memory initialization. After the initialization steps *(i)* – *(iv)* are performed, the rest of the memory M is filled with verifier's choice of constants.

The device initialization and the verifier's protocol with the device are illustrated in Figure 6.

2) Control Flow Integrity: Recall that the *verifier's protocol* begins with the input of *nonce* $H_{d,k,x}$ into a device and

ends when the verifier checks the device's output.

Theorem 6 (Verifiable Control Flow). Let the verifier request a device's untrusted boot loader to initialize its memory, and constant t_0 be the time required by the *Init* program on cWRAM. Let the verifier receive $H_{d,k,x}(v)$ in $t_0 + (6k-4)6d$ time units in response to *nonce* $H_{d,k,x}$. Then

a) there exists a verifier choice of instruction encoding and sequencing for the *processor-state initialization*, *Input*, or *Init* programs such that the omission of any instruction execution modifies at least a word of input v to $Horner(H_{d,k,x}(\cdot))$;

b) a control flow deviation in the verifier-protocol code on the device remains undetected with probability at most $\frac{6}{p}$.

In the proof of a), we use the fact that a verifier can choose instruction encoding and sequencing for the three programs such that the lower $log\ p$ bits of their memory words form a unique sequence of distinct words of the input v to $Horner(H_{d,k,x}(\cdot))$. For part b) we show that, given the the verifier's choice of instruction encoding and sequencing, any control flow deviation from it requires either a modification of input v or a violation of $Horner(H_{d,k,x}(\cdot))$'s bounds, or both. The probability of the former event is bounded by Lemma 4 and of the latter by Theorem 5.

Scalable Bounds. By Theorem 6, the $Horner(H_{d,k,x}(\cdot))$ code bounds for device i must scale from $m = k + 22$ to $m_i = k_i + 22$, $k_i > k$, and from $t = (6k-4)6d$ to $t_i = (6k_i - 4)6d_i$, $d_i = |v^i| - 1 > d$, where $|v^i|$ is the number of memory and special processor-register words. To scale execution time t_i for a constant k_i (and hence m_i), the verifier can increase d_i past constant $|v^i|-1$ to whatever value is required by transaction duration. In this case, indexing in v^i would require an additional $mod\ |v^i|$ instruction execution.

3) Concurrent-transaction order and duration: Let a system comprise c devices with the *smallest word size* of w bits and $p < 2^{w-1}$. Let the verifier request an untrusted boot loader to initialize device i with chosen content v^i as described in Section IV-E1. Then $Init_i$ initializes the k_i general-purpose registers on device i in constant time t_{0_i}. If $t_{0_j} + t_j$ is the *slowest* $Horner(H_{d_j,k_j,x_j}(\cdot))$ execution time on any device's v^j, then the verifier selects values of d_i, k_i for the other device *nonces* H_{d_i,k_i,x_i} such that $t_i = t_{0_i} + (6k_i - 4)6d_i$ equals $t_{0_j} + t_j$, or exceeds it by a very small amount, to satisfy the duration requirement. Then the verifier's chosen concurrent-transaction order can assure that the start times and end times do not allow malicious devices to circumvent lower bounds.

Theorem 7 (Malware-free state). Let a verifier initialize an untrusted c-device system to v^i ($i \in [1, c]$), where c is small; i.e., $10c << p$. Then the verifier challenges the devices concurrently in transaction order, with device i receiving *nonce* H_{d_i,k_i,x_i} whose t_i satisfies the duration requirement. If the verifier receives result $H_{d_i,k_i,x_i}(v^i)$ at time t_i for all i, the probability that malware survives in a system state is at most $\frac{9c}{p}$. If the verifier runs the protocol n times, the malware-survival probability becomes negligible in n; i.e., $\epsilon(n) = [\frac{9c}{p}]^n$.

The proof follows directly from the concurrent transaction order and duration property of the verifier's protocol, Theorem 6, and Lemma 4.

Example. For $w = 32$ and $w = 64$ bits, the largest primes $p < 2^{w-1}$ are $2^{31} - 1$ and $2^{63} - 25$. In practice $c \leq 16$ as we rarely encounter commodity computer systems configured with more than eight CPU cores and eight peripheral-device controllers whose non-volatile memories can be re-flashed with code[12]. For $w = 32$ ($w = 64$), the probability of malware survival for $n = 1$ is less than 2^{-23} (2^{-55}), for $n = 2$ is less than 2^{-46} (2^{-110}), *etc*. Hence, $n \leq 2$ is sufficient, in practice.

V. Unconditional Root of Trust Establishment

Theorem 7 establishes a malware-free, multi-device system state. However, this is insufficient to establish RoT. While the general-purpose registers contain w-bit representations of the k random numbers, the memory and special processor registers of a device comprise w-bit words, rather than the $log\ p$-bit fields $v^i_{d_i}, \ldots, v^i_0$ words, where $p < 2^{w-1}$ is the largest prime. Hence, a sliver of unaccounted for content exists.

To establish RoT, the verifier can load a *word-oriented* (almost) universal hash function in each malware-free device memory and verify the results they return after application to memory and special processor register content. Note that space-time optimality of these hash functions and verifier's protocol atomicity are unnecessary, since malware-freedom is already established. A pairwise verifier - device$_i$ protocol checking device memory and special register content is sufficient. Let $\mathbf{H_w}$ be such a family and V comprise the set of w-bit words of a device's memory and special processor registers.

Fact (*e.g., Exercise 4.4* [88]). Let $q > 2^w$ be a prime, $|V| = q/2^w$, and $a, b, c \xleftarrow{\$} \mathbf{Z_q}$ be the function index of family $\mathbf{H_w}$, where

$$\mathbf{H_w} = \{H_{a,b,c}(\cdot) \mid w_i \in [0, 2^w), H_{a,b,c}(w_{|V|-1} \ldots w_0) = ((a \times (\sum_{i=0}^{|V|-1} w_i \times c^i) + b)\ mod\ q)\ mod\ 2^w\}$$

is a family of almost universal hash functions, with collision probability of $2^{-(w-1)}$. The probability is computed over the choices of $H_{a,b,c}(\cdot) \xleftarrow{\$} \mathbf{H_w}$.

Theorem 8 (RoT Establishment). Let a verifier establish a malware-free state of a c-device system in n protocol runs, as specified in Theorem 7. Then let the verifier load $H_{a_i,b_i,c_i}(\cdot) \xleftarrow{\$} \mathbf{H_w}$ on device i and check each result $H_{a_i,b_i,c_i}(M_i)$ received. If all checks pass, the verifier establishes RoT with probability at least $(1 - \epsilon(n))(1 - c \cdot 2^{w-1})$, where $\epsilon(n) = [\frac{9c}{p}]^n$; e.g., higher than $1 - \frac{10c}{p}$ for $n = 1$.

The proof is immediate by Theorem 7 and the Fact above.

Implementation considerations of the cWRAM model in real processors for suitable choices of prime p are discussed in *Appendix C*.

Secure Initial State. After the verifier establishes RoT, it can load a trustworthy program in the system's primary memory. That program sets the contents of all secondary storage to verifier's choice of content; i.e., content that satisfy whatever security invariants are deemed necessary. This extends the notion of the *secure initial state* to all system objects.

VI. Time-Measurement Security

Past software-based attestation designs fail to assure that a verifier's time measurements cannot be bypassed by an adversary. For example, to account for cache, TLB, and clock

[12]Although GPUs have many cores, GPU malware is *cannot* persist, as it cannot survive GPU power-offs/reboots [76] by *processor-state initialization*.

jitter caused primarily by pseudo-random memory traversals by $C_{m,t}(\cdot)$ computations and large t, typical verifiers' measurements build in some slack time; e.g., 0.2% – 2.6% of t [47], [52], [53], [77]. An adversary can easily exploit the slack time to undetectably corrupt $C_{m,t}(\cdot)$'s memory [47], [52]. In this section we show how to counter these threats.

A. Verifier Channel

The verifier's local channel must satisfy two common-sense requirements. First, the channel connection to any device must not pass through a peripheral device controller that requires RoT establishment. Otherwise, malware on the controller could pre-process some of the computation steps for the verifier's protocol with that device and help it to circumvent the time measurements. Second, the channel's delay and its variation must be small enough so that the verifier time measurements can reliably detect all surreptitious untrusted-system communication with external devices and prevent both *memory-copy* [52] and *remote-proxy* [53] attacks.

We envision a verifier device to be attached to the main system bus via a DMA interface, similar in spirit to that of Intel's Manageability Engine or AMD's Platform Security Processor, but without flaws that would enable an attacker to plant malware in it [63]. These processors can operate independently of all other system components; e.g., even when all other components are powered down [83]. The external verifier could also run on a co-processor connected to the main system bus, similar in spirit to Ki-Mon ARM [51]. In both cases, the verifier would have direct access to all components of the system state. An advantage of such verifiers is that their communication latency and variation of the local channel are imperceptible in contrast with the adversary's network channel.

B. Eliminating Cache and TLB jitter

To perform deterministic time measurement, it is necessary to eliminate cache/TLB jitter and interprocessor interference, and avoid clock jitter in long-latency computations.

Preventing Cache, Virtual Memory, and TLB use. In contrast with traditional software-based attestation checksums (e.g., [47], [52], [77], [78]), the execution-time measurements of $Horner(H_{d,k,x}(v))$ is deterministic. Most modern processors, such as the TI DM3730 ARM Cortex-A8 [6], include cache and virtual-memory disabling instructions. Hence, *processor-state initialization* can disable caches, virtual memory, and the TLB *verifiably* (by Theorem 6). In addition, the Horner-rule step is inherently sequential and hence unaffected by pipelining or SIMD execution. The only instructions whose execution could be overlapped with Horner-rule steps are the two loop control instructions, and the corresponding timing is easily accounted for in the verifier's timing check.

Preventing Cache pre-fetching. In systems where caches cannot be disabled, the *inherent* sequentiality of $Horner(H_{d,k,x}(v))$ code and the known locality of the instruction and operand references helps assure that its execution-time measurements are deterministic. However, the adversary's untrusted boot loader could perform undetected cache pre-fetches before the verifier's protocol starts, by selectively referencing memory areas, and obtain better timing measurements than the verifier's; viz., Section VII. To prevent pre-fetching attacks the *processor-state initialization* can *clear caches verifiably* (by Theorem 6), so that *Init* and $Horner(H_{d,k,x}(v))$ code can commence execution with clean caches. Hence, cache jitter can be prevented.

Alternately, the verifier's processor-state initialization could warm up caches [77] by verifiable pre-fetching. Nevertheless, verifiable cache clearing is often required; e.g., in ARM processors instruction and data caches are not hardware synchronized, and hence they have to be cleared to avoid malware attacks [52]. Furthermore, cache anomalies may occur for some computations where a cache miss may result in a shorter execution time than a cache hit because of pipeline scheduling effects [24]. This makes cache clearing a safer alternative.

C. Handling clock jitter and inter-processor interference

When $Horner(H_{d,k,x}(v))$ executes in large memories it can have large latencies; e.g., several minutes. These may experience small time-measurement variations in some systems due to uncorrected random clock jitter at high frequencies [84], and multi-processor interference in memory accesses. These timing anomalies are typically addressed in embedded real-time systems [24]. For such systems, we use a random sequential protocol. This protocol leverages smaller memory segments and the verifiable choice of clock-frequency setting such that random clock jitter becomes *unmeasurable* by an adversary. It also ensures that different processors access *different* memory segments to eliminate interprocessor interference. The protocol also provides an alternate type of bounds scaling. For example, in controller devices can complete verification concurrently with the first memory segments without having to scale their space-time optimal computations to meet the large time bound of a primary memory verification.

Random Sequential Evaluation. Let $F = \{f_1, f_2, \ldots, f_n\}$ be a family of n functions and $K_i \xleftarrow{\$} [1, n]$, $i = 1, \ldots, N$, be identifiers of their random invocations. $f_{K_1}, f_{K_2}, \ldots, f_{K_N}$ are evaluated on inputs $x_1, x_2, \ldots, x_N$, and $\perp$ denotes the event that an *invalid* result is returned by a function evaluation. The protocol for the random sequential evaluation of F, namely $(f_{K_1}(x_1), f_{K_2}(x_2), \ldots, f_{K_N}(x_N))$, is as follows:
1) $N = n \cdot log\ n$;
2) if $f_{K_i}(x_i) \neq \perp$, then $f_{K_{i+1}}(x_{i+1})$, $1 \leq i < N$; and
3) $Pr[K_i \xleftarrow{\$} [1, n] : \forall j > i, f_{K_j}(x_j) = y_j \mid f_{K_i}(x_i) = y_i, \cdots, f_{K_1}(x_1) = y_1] = Pr[K_i \xleftarrow{\$} [1, n] : f_{K_j}(x_j) = y_j]$.
The evaluation terminates *correctly* if $f_{K_i}(x_i) \neq \perp$ for all i, and *incorrectly*, otherwise.

Condition 1) implies that the evaluation invokes *all* randomly selected functions with high probability *at least once* [25], [77]. Condition 2) defines the sequential evaluation rule. Condition 3) implies that the j-th function evaluation is independent from the previous $i < j$ evaluations.

Verifier Initialization. Let the verifier request the boot loader to initialize M to n memory segments each comprising processor-state initialization, I/O, $Init$, and $Horner(H_{d,k,x}(\cdot))$ programs. Then verifier's boot loader transfers control to the first instructions of the *processor-initialization* program.

Verifier Protocol. Let F be family $\mathbf{H}$, f_{K_i} be $Horner(H_{d_i,k_i,x_i}(\cdot))$, where $K_i \xleftarrow{\$} [1, n]$, and $H_{d_i,k_i,x_i}(\cdot) \xleftarrow{\$}$

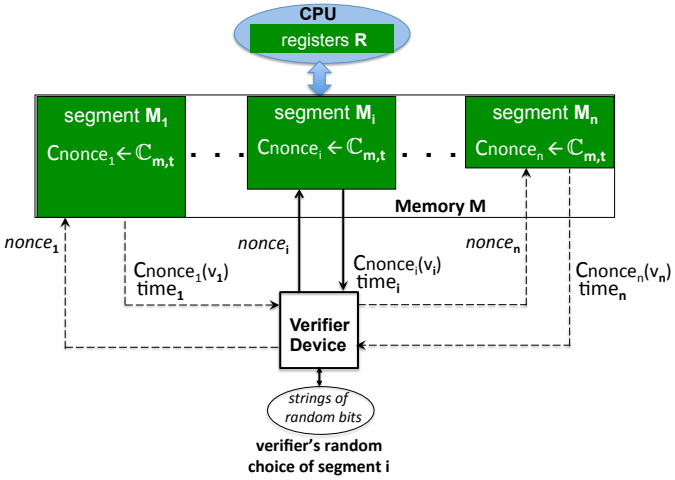

Fig. 7: Random Sequential Evaluation Protocol

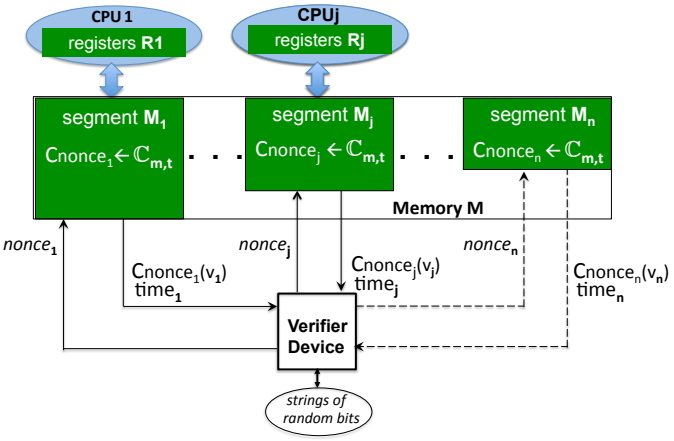

Fig. 8: Processors Accessing Different Memory Segments

H; i.e., the random selection of a memory segment[13]. If the random sequential evaluation protocol terminates incorrectly or the termination is untimely, or both, the verifier *rejects*. Otherwise, the verifier *accepts*. This is the verifier's protocol for *the n-segment memory model*. The protocol is illustrated in Figure 7.

Specifically, the verifier writes the values denoting the choice of $H_{d_i,k_i,x_i}(\cdot) \xleftarrow{\$} \mathbf{H}$ separately to each of the n memory segments. Furthermore, the verifier's *Output* code is modified so that it returns to the *Input* busy-waiting code after outputting an evaluation result, which transfers to the first instruction of the *Input* code of the next randomly chosen segment. The address of the next segment's *Input* code is provided by the verifier along with the next *nonce* $H_{d_i,k_i,x_i}(\cdot) \xleftarrow{\$} \mathbf{H}$.

Note that the size of the segments shown in Figure 7 can vary. Memory M can be initialized with segments that are small enough such that their evaluation time becomes smaller than the round-trip time necessary for a *remote proxy* attack, where the remote proxy is powerful enough to evaluate any randomized polynomial in zero time [53]. Hence, if malware attempts to enlist the help of an adversary of unbounded power to bypass the optimal space-time bounds of all the small-memory segments, the verifier protocol fails.

In a multiprocessor system where j processors share RAM memory M, the $Init$ programs would start the concurrent execution of all j processors in *different* memory segments along with those of the device controllers; see Figure 8. The assignment of segments to processors can be done by selecting j segments at random without replacement from the $n \geq j$ segments. Thus no two processors or more can interfere in accessing the same segment at the same time.

Theorem 9 (Malware-free Segmented Memory). Let a verifier initialize memory M of a (e.g., multiprocessor) device to n segments and perform the verifier's protocol for the segmented memory. If the verifier accepts the result, the device state is malware-free, except with probability at most $\frac{9n}{p}$.

The proof of this theorem follows from the definition of the verifier's initialization of memory M including the modified I/O instruction sequences, by the verifier's protocol for the segmented memory model, and by Theorem 6 and Lemma 4.

VII. Performance

In this section, we present preliminary performance measurements for the Horner-rule evaluation of randomized polynomials. The only goal here is to illustrate implementation practicality on a commodity hardware platform. For this reason, we compare these measurements to those of Pioneer – the best-known attestation checksum [77] – on the same hardware configuration [52]. Presenting a study of randomized-polynomial performance is beyond the scope of this paper.

Our measurements also illustrate the importance to provably clearing (or disabling, when possible) caches for deterministic time measurements. We noticed no timing anomalies due to uncorrected clock jitter in our single-processor configuration for a fairly large memory. This suggests that the random sequential evaluation for large memories (Section VI) may be useful primarily to prevent inter-processor interference.

Hardware. Our measurements were done on a Gumstix Overo FireSTORM-P Computer-On-Module (COM), which is the ARM-based development platform for embedded hardware used by Li *et al.* [52]. This gives us an opportunity to compare the performance of Horner's rule for randomized polynomials with that of the Pioneer checksum. This platform features a 1GHz Texas Instruments DM3730 ARM Cortex-A8 32-bit processor and 512MB of DDR SDRAM [86]. The processor has a 32KB L1 instruction cache and a 32KB L1 data cache, both with 16 bytes per cache line. In addition, it also features a 256KB L2 unified cache [6].

Recall that the parameter $|M|$ must reflect the total amount of primary storage in the device. Besides the 512MB of SDRAM, our particular Gumstix also features 64KB of SRAM and also a large address space for device control registers with $5,548$ registers. Summing these up as bits, we set $|M|$ to $4,295,669,120$.

Software. Our measurements are implemented inside a popular secondary boot loader known as U-Boot, which in a

[13] A *non-random* sequential selection would enable malware to take control after a correct and timely result is returned by a memory segment evaluation, modify the memory of an already evaluated segment or prefetch instructions, and then overwrite itself with correct evaluation code before the next input arrives from the verifier.

typical application would be responsible for loading the Linux kernel on the COM. For our purpose, however, we extend U-Boot with measurement capabilities; i.e., U-Boot 2015.04-rc4 is cross-compiled with Linaro *gcc* 4.7.3.

We implemented Horner's rule for several polynomials in $\mathbf{Z_p}$, where $p = (2^{32} - 5)$ is the largest prime that can fit inside a 32-bit register. Since the DM3730 ARM Cortex-A8 CPU does *not* support the `udiv` (unsigned integer division) instruction, *gcc* uses the `__aeabi_uidivmod` function to emulate division, which is slower than the hardware `udiv` instruction followed by the `mls` (integer multiply-and-subtract) instruction to compute the modulus. Nevertheless, an adversary cannot change the emulation since the code image is committed by the second pre-image freedom of randomized polynomials.

The first Horner-rule measurement is for ordinary polynomials; i.e., with constant, rather than k-wise independent, coefficients. This establishes the baseline, which helps calibrate the expected performance loss for increasing the values of k. The performance of Horner rule for a single polynomial of degree 128M covering the entire SDRAM is $11,739$ms.

For the measurements of Horner-rule evaluation of randomized polynomials, the k random numbers are stored contiguously in memory. For values of k that match one cache line, namely $k = 4$, evaluating a polynomial of degree $d = 128$M (same as the baseline) takes $67,769$ms due to extra memory accesses and added cache contention. However, most modern processors have more than $k = 4$ and fewer than $k = 64$ registers. Hence, larger values of k would have to be used to ensure that the adversary *cannot* be left with spare processor registers after loading the k random numbers.

Randomized Polynomials vs. Pioneer Checksum. The timing for $k = 64$ and $d = 10$M is $54,578$ms. For the baseline $d = 128$M the running time is close to 700 seconds, which is about 6% faster than the fastest Pioneer checksum (745.0 seconds), 8.7% faster than the average (765.4 seconds), and 9% faster than the slowest (768.1 seconds) reported by Li *al.* [52] on the same hardware configuration. While these measurements illustrate practical usefulness, additional measurements are necessary for a complete performance study; e.g., additional hardware platforms and configurations.

Why Disable or Clear Caches? Instruction and data caches on the DM3730 ARM Cortex-A8 can be disabled and enabled individually, using single instructions. We used this feature to illustrate the inferior cache utilization compared to an adversary's cache pre-fetching strategy. With only the instruction (data) cache turned off, we observed a 5.15x (23.76x) slowdown in Horner-rule evaluation. With both caches turned off, the slowdown increases to 53.13x. This shows that the adversary can gain a real advantage by cache pre-fetching.

VIII. Related Work

A. Past Attestation Protocols

Past attestation protocols, whether software-based [7], [40], [47], [77], [78], [80], cryptographic-based [8], [22], [27], [38], [46], [64], or hybrid [53], [94], 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 verification can be remote and can be repeated after boot, rather than local and limited to pre-boot time.

Given their different goals, it is unsurprising that past attestation protocols fail to satisfy some RoT establishment requirements defined in Section II even for bounded adversaries and secret-key protection in trusted hardware modules. For example, these protocols need not be concerned with the content of *system registers* (e.g., general processor and I/O registers), since they cannot contain executable code. Furthermore, they need not satisfy the concurrent-transaction order and duration requirements (see Section II-C) of the verifier's protocol since they need not establish any system state properties, such as secure initial state in multi-device systems. Finally, none of these protocols aims to satisfy security properties *provably and unconditionally*. Beyond these common differences, past protocols exhibit some specific differences.

Software-based attestation. Some applications in which software-based attestation can be beneficially used do not require control-flow integrity [69], and naturally this requirement is not always satisfied [15], [52]. A more subtle challenge arises *if* one uses traditional checksum designs with a *fixed* time bound in a multi-device system since scalable time bounds are important. As shown in Section II-C, these checksums cannot scale time bounds by repeated checksum invocation with different *nonces* and retain optimality. Software-based attestation models [7], [40] also face this challenge.

Despite their differences from RoT establishment, software-based attestation designs met their goals [77], [78], and offered deep insights on how to detect malware on peripheral controllers [53], embedded devices [15], [52], mobile phones [40], and special processors; e.g., TPMs [47].

Cryptographic attestation. Cryptographic protocols for remote attestation typically require a trusted hardware module in each device, which can be as simple as a ROM module [46], to protect a secret key for computing digital signatures or MACs. If used in RoT establishment, the signature or MAC computations must verifiably establish control-flow integrity. Otherwise, similar control-flow vulnerabilities as software-based attestation would arise. Furthermore, the trusted hardware module must protect *both* the secret key *and* the signature/MAC generation code.

More importantly, cryptographic attestation relocates the root of trust to the third parties who install the cryptographic keys in each device controller and those who distribute them to verifiers. However, 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 verifier can be compromised, which enables known attacks; e.g., see the Cuckoo attack [66]. Thus, these protocols can offer only conditional security.

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

B. Polynomial Evaluation

If the only operations allowed for polynomial evaluation are the addition and multiplication, Horner rule's bound of $2d$ operations for degree-d polynomials was shown to be uniquely optimal in one-time evaluations [12], [72]. However, this bound does not hold in finite fields, where the minimum number of modular additions and multiplications is $\Omega(\sqrt{(d+1)})$ [43]. Furthermore, these bounds do not hold in any WRAM models or any real computer where *many* more operations are implemented by the ISA.

For WRAM models with *variable* word widths, polynomial-evaluation lower bounds are typically obtained in the *cell probe* model. Here the polynomial is assumed to be already initialized in memory. The evaluation consists of the reading (probing) a number of cells in memory, and after of all read operations are finished, it must output the result. The cell probed by each read operation may be any function of the previously probed cells and read operations, and thus all computations on the already read data take no time.

Using the cell-probe model, Gál and Miltersen [28] showed that the size r of any additional data structure needed for the evaluation of a degree-d polynomial beyond the information theoretical minimum of $d+1$ words must satisfy $r \cdot t = \Omega(d)$, where t is the number of probes, $d \leq p/(1+\epsilon)$, p is a prime, and $\epsilon > 0$. For linear space data structures (i.e., w-bit words and memory size $|M| = O(d \cdot log\ p/w)$), Larsen's lower bound of $\Omega(log\ p)$ is the highest [50], but it is not close to the lowest known upper bound [44]. Neither bound holds in cWRAM or in a real computer.

C. Memory-Hard Functions

Recent interest in countering password cracking and crypto-currency mining by off-line cryptographic computations on GPU and ASIC cores has led to the design of memory-hard functions (MHFs). If n is the number of hash function output blocks that fit into a large memory buffer, an optimal MHF implements the fundamental space-time trade-off $m \cdot t \in \Omega(n^2)$ in sequential evaluation; whereas in parallel evaluation, the amortized space cost over time has the lower bound of $\Omega(\frac{n^2}{log\ n})$ [10], [11]. A goal of MHF design is to make the asymptotic constants be as high as possible to amplify the space-time trade-off.

Leaving aside the fact that the MHF adversary is always bounded, the *properties* of optimal MHFs differ from those of the optimal space-time computations in RoT establishment for at least three basic reasons. First, no matter how one adjusts the number of rounds an MHF traverses the available memory to increase its asymptotic constant, the evaluation time t cannot scale independently of m, as required by the verifier's protocol; see Section II-C. Second, while a MHF's memory space comprises n blocks, it does not capture the number and content of processor registers, since they do not add any MHF adversary advantage. Third, the MHF time measurement is approximate: it is unaffected by jitter caused by cache/TLB use and multiprocessor interference in shared memory access, in contrast with time-measurement in establishing malware-freedom and RoT. Of course, MHF have other desirable properties, such as the independence of memory access patterns [9], [10], [11] , which are irrelevant for randomized polynomials.

Further, the *use* of MHFs in password and mining attacks is very different from randomized polynomials in RoT establishment. First, unlike a MHF [11], a randomized polynomial cannot be evaluated in parallel; i.e., the verifier ensures that different device controllers (Section II-C), and different processors or cores (Section VI-C), of the same system evaluate different randomized polynomials. Second, the evaluation of a randomized polynomial on a real processor, such as an Intel x86 processor, yields two or more space-time optimal execution points that exclude each other; see Appendix C. That is, unlike with MHFs, space-time trade-offs are ruled out in randomized polynomial evaluations. Third, the MHF bounds are asymptotic, even when optimal, and hence unusable by verifiers; see Section II-B.

IX. Conclusions

RoT establishment is a necessary primitive for a variety of basic system security problems, including starting a system in a secure initial state [30], [31] and performing trusted recovery [62]. These problems have not been demonstrably resolved since their identification decades ago. They only became harder in the age of persistent malware attacks. RoT establishment is also necessary for *verifiable boot* – a stronger notion than secure and trusted boot [29]. It is also a basic requirement for the installation of secrets into a computing device for later use in secure cryptographic protocols and applications [32].

In this paper we showed that, with a proper theory foundation, RoT establishment can be both provable *and* unconditional. We know of no other software security problem that has had such a solution, to date. Finally, the security of time measurements on untrusted systems has been a long-standing unsolved engineering problem [47], [52], [53], [77]. Here, we also showed that this problem can be readily solved given the provable atomicity of the verifier's protocol.

Acknowledgment

Comments received from Pompiliu Donescu, Bryan Parno, Adrian Perrig, Vyas Sekar, Gene Tsudik, Osman Yağan, and Jun Zhao helped improve the clarity of this paper substantially.

References

[1] Andersson, A., Miltersen, P. B., Riis, S., and Thorup, M. Static dictionaries on AC^0 RAMs: query time $\Theta(\sqrt{log\ n/log\ log\ n})$ is necessary and sufficient. In *in Proc 37th FOCS* (1996), pp. 441–450.

[2] Anthony, S. Massive, undetectable security flaw in USB: It's time to get your PS/2 keyboard out of the cupboard. *Extreme Tech*, July 31 (2014).

[3] Applebaum, B., Haramaty, N., Ishai, Y., Kushilevitz, E., and Vaikuntanathan, V. Low-complexity cryptographic hash functions. In *Proc. 8th ITCS, Berkeley, CA* (2017), pp. 7:1–7:31.

[4] Applebaum, B., and Moses, Y. Locally computable UOWHF with linear shrinkage. *J. Cryptology 30*, 3 (2017), 672–698.

[5] Applebaum, J., Horchert, J., and Stocker, C. In *Catalog Reveals NSA Has Back Doors for Numerous Devices* (2013), vol. Dec. 29, Springer Online.

[6] ARM. Cortex-A8 Technical Reference Manual. Rev.:r3p2, May 2010.

[7] Armknecht, F., Sadeghi, A.-R., Schulz, S., and Wachsmann, C. A security framework for the analysis and design of software attestation. In *Proc. of ACM CCS, Berlin, Germany* (2013), pp. 1–12.

[8] Asokan, N., Brasser, F., Ibrahim, A., Sadeghi, A.-R., Shunter, M., Tsudik, G., and Wachsmann, C. SEDA: scalable embedded device attestation. In *Proc. of 2015 ACM CCS, Denver, Colorado* (2015), ACM.

[9] BLOCKI, J., HARSHA, B., KANG, S., LEE, S., XING, L., AND ZHOU, S. Data-independent memory hard functions: New attacks and stronger constructions. Cryptology ePrint Archive, Report 2018/944, 2018. https://eprint.iacr.org/2018/944.

[10] BLOCKI, J., REN, L., AND ZHOU, S. Bandwidth-hard functions: Reductions and lower bounds. In *Proc. of the ACM CCS 2018, Toronto, ON, Canada, October 15-19, 2018* (2018), pp. 1820–1836.

[11] BONEH, D., CORRIGAN-GIBBS, H., AND SCHECHTER, S. Balloon hashing: A memory-hard function providing provable protection against sequential attacks. In *Advances in Cryptology – ASIACRYPT 2016* (2016), Springer Berlin Heidelberg, pp. 220–248.

[12] BORODIN, A. Horner's rule is uniquely optimal. In *Proc. of the International Symposium on the Theory of Machines and Computations* (1971), Z. Kohavi and A. Paz, Eds., Elsevier Inc, pp. 47–57.

[13] CAMPBELL, H. God does play dice with the universe (and the dice are fair). *Science 2.0*, July (2012).

[14] CARTER, L., AND WEGMAN, M. Universal classes of hash functions. *Journal of Computer and System Sciences 18*, 2 (1979), 143 – 154.

[15] CASTELLUCCIA, C., FRANCILLON, A., PERITO, D., AND SORIENTE, C. On the difficulty of software-based attestation of embedded devices. In *Proc. of the 16th ACM CCS* (2009), pp. 400–409.

[16] CHRISTIANI, T., AND PAGH, R. Generating k-independent variables in constant time. In *2014 IEEE 55th IEEE FoCS* (Oct 2014), pp. 196–205.

[17] CHRISTIANI, T., PAGH, R., AND THORUP, M. From independence to expansion and back again. In *Proc. of the ACM Symp. on Theory of Computing* (2015), STOC '15, pp. 813–820.

[18] COSTAN, V., AND DEVADAS, S. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[19] CUI, A., COSTELLO, M., AND STOLFO, S. When firmware modifications attack: A case study of embedded exploitation. In *Proc. of the 2013 NDSS* (2013), ISOC.

[20] DELUGRE, G. Closer to metal: Reverse engineering the broadcom NetExtreme's firmware. In *Sogeti ESEC Lab* (2010).

[21] DUFLOT, L., PEREZ, Y.-A., AND MORIN, B. What if you can't trust your network card? In *Proc.of the 14th RAID* (2011), Springer.

[22] ELDEFRAWY, K., PERITO, D., AND TSUDIK, G. SMART: Secure and minimal architecture for (establishing a dynamic) root of trust. In *NDSS, San Diego, CA* (2012).

[23] ELIA, M., ROSENTHAL, J., AND SCHIPANI, D. Polynomial evaluation over finite fields: new algorithms and complexity bounds. *Applicable Algebra in Engineering, Communication and Computing 23*, 3 (Nov 2012), 129–141.

[24] ENGBLOM, J., ERMEDAHL, A., AND SJODIN, M. Worst-case execution-time analysis for embedded real-time systems. In *International Journal on Software Tools for Technology Transfer* (2003), vol. 4, pp. 437–455.

[25] ERDOS, P., AND RENYI, A. On a classical problem of probability theory. In *Proc. of Magyar Tudomanyos Akad. Matematikai Kutato Intezetenek Kozlemenyei* (1961), pp. 215–220.

[26] FOG, A. Instruction tables. Technical University of Denmark, Sept. 15, 2018.

[27] FRANCILLON, A., NGUYEN, Q., RASMUSSEN, K. B., AND TSUDIK, G. A minimalist approach to remote attestation. In *Proc. of the Conference on Design, Automation & Test in Europe, Dresden, Germany* (2014), pp. 244:1–244:6.

[28] GÁL, A., AND MILTERSEN, P. B. The cell probe complexity of succinct data structures. *Theoretical Computer Science 379*, 1 (2007), 405–417.

[29] GLIGOR, V. Dancing with the adversary – A tale of wimps and giants. In *Proc. of the 22nd Security Protocols Workshop, Cambridge, UK* (2014), vol. 8809 of *LNCS*, Springer, pp. 116–129.

[30] GLIGOR, V., AND GAVRILA, S. Application-oriented security policies and their composition. In *Proc. of the Security Protocols Workshop, Cambridge, UK* (1998), vol. 1550 of *LNCS*, Springer, pp. 67–74.

[31] GLIGOR, V., GAVRILA, S., AND FERRAIOLO, D. On the formal definition of separation of duty policies and their composition. In *Proc. of the IEEE Symp. on Security and Privacy* (1998), pp. 172–183.

[32] GLIGOR, V. D., AND WOO, M. Requirements for Root of Trust Establishment (Transcript of Discussion). In *Proceedings of the 26nd Security Protocols Workshop, Cambridge, UK* (03 2018), vol. 11286 of *Lecture Notes in Computer Science*, Springer, pp. 203–208.

[33] GRANLUND, T. Instruction latencies and throughput for AMD and Intel x86 processors, April 24, 2017.

[34] GRANLUND, T., AND MONTGOMERY, P. L. Division by invariant integers using multiplication. *ACM SIGPLAN Notices 29*, 6 (1994), 61–72.

[35] H. IBARRA, O., MORAN, S., AND E. ROSIER, L. On the control power of integer division. *Theoretical Computer Science 24* (06 1983), 3552.

[36] HAGERUP, T. Searching and sorting in the Word RAM. In *Proc. of the 15th Symp. on Theoretical Aspects of Computer Science (STACS 98)* (1998), Springer.

[37] HERRERO-COLLANTES, M., AND GARCIA-ESCARTIN, J. C. Quantum random number generators. *Reviews of Modern Physics 89*, 1 (2017).

[38] IBRAHIM, A., SADEGHI, A.-R., TSUDIK, G., AND ZEITOUNI, S. DARPA: Device Attestation Resilient to Physical Attacks. In *Proc. of 9th ACM Conf. on Sec. & Priv. in Wireless and Mobile Networks* (2016), WiSec '16, pp. 171–182.

[39] INTEL CORP. Intel 64 and I-32 architecture optimization – Reference Manual. p. 340.

[40] JAKOBSSON, M., AND JOHANSSON, K.-A. Retroactive detection of malware with applications to mobile platforms. In *Proc. of 5th USENIX HoTSec Workshop* (2010).

[41] JOHANSSON, T., KABATIANSKII, G., AND SMEETS, B. On the relation between A-Codes and Codes Correcting Independent Errors. In *Advances in Cryptology — EUROCRYPT '93* (1994), T. Helleseth, Ed., Springer Berlin Heidelberg.

[42] KASPERSKY LAB. The Duqu 2.0 - Technical Details (ver. 2.1), 2015.

[43] KAYAL, N., AND SAPTHARISHI, R. A selection of lower bounds for arithmetic circuits. In *Perspectives in Computational Complexity – Progress in Computer Science and Applied Logic* (2014), M. Agrawal and V. Arvind, Eds., International Publishing Switzerland, pp. 77–115.

[44] KEDLAYA, K. S., AND UMANS, C. Fast polynomial factorization and modular composition. *SIAM J. on Comput. 40*, 6 (2011), 1767–1802.

[45] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. sel4: Formal verification of an os kernel. In *Proc. of the 22nd ACM SOSP* (2009), pp. 207–220.

[46] KOEBERL, P., SCHULZ, S., SADEGHI, A.-R., AND VARADHARAJAN, V. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems* (2014), EuroSys '14, pp. 10:1–10:14.

[47] KOVAH, X., KALLENBERG, C., WEATHERS, C., HERZOG, A., ALBIN, M., AND BUTTERWORTH, J. New results for timing-based attestation. In *Proc. of the 2012 IEEE Symposium on Security and Privacy* (2012), pp. 239–253.

[48] KROVETZ, T. Message authentication on 64-Bit architectures. In *Selected Areas in Cryptography* (2007), E. Biham and A. M. Youssef, Eds., Springer Berlin Heidelberg.

[49] KROVETZ, T., AND ROGAWAY, P. Fast universal hashing with small keys and no preprocessing: The polyr construction. In *Information Security and Cryptology (ICISC)* (2001), Springer Berlin Heidelberg, pp. 73–89.

[50] LARSEN, K. G. Models and techniques for proving data structure lower bounds. PhD Dissertation, University of Aahrus, Denmark, 2013.

[51] LEE, H., MOON, H., HEO, I., JANG, D., JANG, J., KIM, K., PAEK, Y., AND KANG, B. Ki-mon ARM: A hardware-assisted event-triggered monitoring platform for mutable kernel object. *IEEE Transactions on Dependable and Secure Computing PP*, 99 (2017), 1–1.

[52] LI, Y., CHENG, Y., GLIGOR, V., AND PERRIG, A. Establishing software-only root of trust on embedded systems: facts and fiction. In *Proc. of the Security Protocols Workshop* (2015), vol. 9379 of *LNCS*, Springer, pp. 50–68.

[53] LI, Y., MCCUNE, J. M., AND PERRIG, A. VIPER: Verifying the Integrity of PERipherals' firmware. In *Proc. of the 18th ACM CCS, Chicago, IL* (2011), pp. 3–16.

[54] LONE-SANG, F., NICOMETTE, V., AND DESWARTE, Y. I/O attacks in Intel-PC architectures and countermeasures. In *Symp. for the Security of Information and Communication Technologies SSTIC* (2011).

[55] LONE-SANG, F., NICOMETTE, V., AND DESWARTE, Y. A tool to analyze potential I/O attacks against PCs. In *IEEE Security and Privacy* (2014), pp. 60–66.

[56] MANSOUR, Y., NISAN, N., AND TIWARI, P. The computational complexity of universal hashing. In *Proc. of ACM STOC* (1990).

[57] MANSOUR, Y., SCHIEBER, B., AND TIWARI, P. A lower bound for integer greatest common divisor computations. *J. ACM 38*, 2 (1991).

[58] MASSALIN, H. Superoptimizer – a look at the smallest program. In *In Proc. of the Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1987), ACM Press.

[59] MEARIAN, L. There's no way of knowing if the NSA's spyware is on your hard drive. *Computerworld 2* (2015).

[60] MILTERSEN, P. B. Lower bounds for static dictionaries on RAMs with bit operations but no multiplication. In *Proc. of the Int. Coll. on Automata, Languages and Programming (ICALP)* (1996), Springer, pp. 442–453.

[61] NATIONAL COMPUTER SECURITY CENTER. Trusted computer system evaluation criteria (The Orange Book), 1985. DoD 5200.28-STD.

[62] NATIONAL COMPUTER SECURITY CENTER. A guideline for understanding trusted recovery in trusted systems (Yellow Book), 1991. NCSC-TG-022, Library No. 5-236,061 Version 1.

[63] NEWMAN, L. H. Intel chip flaws leave millions of devices exposed. WIRED, 2017.

[64] NOORMAN, J., AGTEN, P., DANIELS, W., STRACKX, R., VAN HERREWEGE, A., HUYGENS, C., PRENEEL, B., VERBAUWHEDE, I., AND PIESSENS, F. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Proc. of 22nd USENIX Security Symposium (USENIX Security 13)* (2013).

[65] PAGH, A., AND PAGH, R. Uniform hashing in constant time and optimal space. *SIAM J. Comput. 38*, 1 (Jan 2008), 85–96.

[66] PARNO, B. Bootstrapping trust in a trusted platform. In *Proceedings of the 3rd conference on Hot topics in security* (2008), USENIX Association, pp. 1–6.

[67] PARNO, B., MCCUNE, J. M., AND PERRIG, A. *Bootstrapping Trust in Modern Computers*, vol. 10 of *Springer Briefs in Computer Science*. Springer, 2011.

[68] PĂTRAŞCU, M. WebDiarios de Motocicleta, December 2008.

[69] PERRIG, A., AND VAN DOORN, L. Refutation of "On the Difficulty of Software-Based Attestation of Embedded Devices", 2010.

[70] PIRONIO, S. *et al.*. Random numbers certified by Bell's theorem. *Nature* (Apr 2010).

[71] RAIU, C. Commentary in Equation: The Death Star of the Malware Galaxy. In *Kaspersky Lab* (Feb 2015).

[72] REINGOLD, E. M., AND STOCKS, A. I. Simple proofs of lower bounds for polynomial evaluation. In *Proceedings of the IBM Research Symposium on the Complexity of Computer Computations* (1972), Springer, pp. 21–30.

[73] RIVEST, R. Permutation polynomials modulo 2^w. In *Finite Fields and Their Applications* (04 2001), vol. 7, pp. 287–292.

[74] SCHKUFZA, E., SHARMA, R., AND AIKEN, A. Stochastic superoptimization. In *Proc. of the Conf. on Architectural Support for Programming Languages and Operating Systems* (2013), ASPLOS, pp. 305–316.

[75] SCHKUFZA, E., SHARMA, R., AND AIKEN, A. Stochastic program optimization. *Commun. ACM 59*, 2 (February 2016 2016).

[76] SCHMUGAR, C. GPU malware: separating fact from fiction. In *McAfee Labs Threats Report* (2015), no. August, pp. 25–28.

[77] SESHADRI, A., LUK, M., SHI, E., PERRIG, A., VAN DOORN, L., AND KHOSLA, P. Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems. In *Proc. of the 20th ACM SOSP* (2005), pp. 1–16.

[78] SESHADRI, A., PERRIG, A., VAN DOORN, L., AND KHOSLA, P. SWATT: Software-based attestation for embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy* (2004), IEEE, pp. 272–282.

[79] SHALLUE, C. J. Permutation polynomials of finite fields. *ArXiv e-prints* (Nov 2012).

[80] SPINELLIS, D. Reflection as a mechanism for software integrity verification. *ACM Transactions on Information System Security 3*, 1 (Feb. 2000), 51–62.

[81] SRINIVASAN, V., SHARMA, T., AND REPS, T. Speeding up machine-code synthesis. In *Proc. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications* (2016), OOPSLA, pp. 165–180.

[82] STEIN, W. *Elementary Number Theory: Primes, Congruences, and Secrets*, 2017.

[83] STEWIN, P. Detecting peripheral-based attacks on the host memory. *T-Lab Series in Telecommunication Services*, Springer-Verlag, 2014.

[84] TAM, S. Modern clock distribution systems. In *Clocking in Modern VLSI Systems*, Integrated Circuits and Systems. Springer, 2009, ch. 2, pp. 6–95.

[85] TARUI, J. Randomized polynomials, threshold circuits, and the polynomial hierarchy. In *STACS 91: 8th Ann. Symp. on Theoretical Aspects of Computer Science, Hamburg, Germany* (1991), Springer, pp. 238–250.

[86] TEXAS INSTRUMENTS. AM/DM37X multimedia device technical reference manual. Version R, Sep 2012.

[87] THE TRUSTED COMPUTING GROUP. TPM Main specification version 1.2 (revision 116), 2011.

[88] THORUP, M. High speed hashing for integers and strings. *CoRR abs/1504.06804* (Sept 2015).

[89] VASUDEVAN, A., CHAKI, S., JIA, L., MCCUNE, J., NEWSOME, J., AND DATTA, A. Design, implementation and verification of an eXtensible and Modular Hypervisor Framework. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (2013), IEEE, pp. 430–444.

[90] WEGMAN, M., AND CARTER, L. New hash functions and their use in authentication and set equality. *J. of Computer and Systems Sciences*, 22 (1981), 265–279.

[91] YAO, A. C.-C. Should tables be sorted? *J. ACM 28*, 3 (1981), 615–628.

[92] YOUNG, W., BOEBERT, E., AND KAIN, D. Proving a computer system secure. *Scientific Honeyweller 6*, 2 (1985), 18–27.

[93] ZADDACH, J., KURMUS, A., BALZAROTTI, D., BLASS, E., FRANCILLON, A., GOODSPPED, T., GUPTA, M., AND KOLTSIDAS, I. Implementation and implications of a stealth hard-drive backdoor. In *Proc. of the 29th ACSAC* (2013), ACM.

[94] ZHAO, J., GLIGOR, V. D., PERRIG, A., AND NEWSOME, J. *ReDABLS: Revisiting Device Attestation with Bounded Leakage of Secrets*, vol. 8263 of *Lecture Notes in Computer Science*. Springer, 2013.

[95] ZHOU, Z., YU, M., AND GLIGOR, V. D. Dancing with giants: Wimpy kernels for on-demand isolated I/O. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (2014), IEEE, pp. 308–323.

[96] ZIEVE, M. On a theorem of Carlitz. *J. of Group Theory 17* (10 2014).

X. Appendix A – The Concrete Word RAM Model

Storage. cWRAM storage includes a fixed sequence M of w-bit memory words indexed by an integer, such that constant $w > log|M|$. The allocation of each instruction in a memory word follows typical convention: the *op code* in the low-order bytes and the operands in the higher-order bytes. Furthermore, cWRAM storage also includes k w-bit general-purpose processor registers, $R_0, R_1, \ldots, R_{k-1}$. A memory area is reserved for the *memory mapped* I/O registers of different devices and the *interrupt vector* table, which specifies the memory location of the interrupt handlers. The I/O registers include data registers, device-status registers, and device-control registers.

Special Registers. In addition to the program counter (PC), the processor state includes internal registers that contain the asynchronous-event status bits specifies whether these events can be posted or are disabled; e.g., by the events clear or enable instructions. It also includes a set of flags and processor configuration settings (e.g., clock frequency) and specifies whether virtual memory/TLBs and caches are enabled. Instructions to

enable and disable cashes/virtual memory are also included. In systems that do not automatically disable cache use when virtual memory is disabled, an internal register containing cache configuration status is provided.

Addressing. Each instruction operand is located either in a separate memory word or in the immediate-addressing fields of instructions. Immediate addressing is applicable only when operands fit into some fraction of a word, which depends on the size of the instruction set and addressing mode fields. Indirect, PC-relative, and bit addressing are also supported.

Instruction Set. The cWRAM instruction set includes all the types of *practical RAM* instructions [60] with up to two operands.

- *Register initialization. Load immediate:* $R_i := \alpha$, or *relative:* $R_i := M[PC + \alpha]$, where α is a constant, and *direct Read:* $R_i := M[R_j]$;

- *Register transfer. Move:* $R_i := R_j$; *Write:* $M[R_i] := R_j$;

All known register initialization and transfer instructions can be represented in cWRAM. They can access memory-mapped I/O registers in I/O transfers.

- *Unconditional branches: go to g*. Branch target g designates either a positive/negative offset from the current *program counter*, PC, and the branch-target address is $PC + g$, or a register R_k, which contains the branch-target address.

- *Conditional branches:* for each predicate *pred*: $F_{2^w} \times F_{2^w} \rightarrow \{0, 1\}$, where *pred* $\in \{\leq, \geq, =, \neq\}$, there is an instruction $pred(R_i, R_j)g$, which means *if pred*$(R_i, R_j) = 1(true)$, *go to* $PC + g$.

If one of the input registers, say R_j, contains a *bit mask*, there is an instruction $pred(R_i, mask)g$, which means *if* $(R_i \wedge mask) = 0$, *go to* $PC + g$. If $R_j = 0$, there is an instruction $pred(R_i)g$, which means *if pred*$(R_i, 0) = 1$, *go to PC+g*.

Note that the predicate set, *pred*, can be extended with other two-operand predicates so that all known conditional-branch instructions can be represented in cWRAM.

- *Halt*: there is an instruction that stops program execution and outputs either the result, when program accepts the input, or an error when the program does not.

- *Computation Instructions.* We adapt Miltersen's notion of the *function locality* [60] for computation functions and use it to classify the set of cWRAM *computation instructions* based on their locality.

Function Locality. Let $\mathbf{I} = \{i_j > i_{j-1} > ... > i_1\} \subseteq \{0, 1, ..., w-1\}$ be a bit-index set, $x \in \{0, 1\}^w$ a bit string of length w, and write $x[\mathbf{I}] = x[i_j]x[i_{j-1}]...x[i_1]$ for the bits of x selected by I. Let $I = \{i, i+1, , j-1, j\}$ be an interval of consecutive bit indices. Then, for constants $\alpha, \beta \geq 1$, function $f : F_{2^w} \times F_{2^w} \rightarrow F_{2^w}$ is (α, β)-*local* if for any interval I of cardinality $\#I$ there are intervals I_1 and I_2, such that:

- $\#I_1, \#I_2 \leq \alpha \cdot \#I$; and

- if the values represented by the bits of x selected by I_1 and those of y selected by I_2 are fixed, then the bits of $f(x, y)$ selected by I take on at most β different values; i.e., for any constants c_1, c_2,

$$\#\{f(x, y)[I] \mid (x)[I_1] = c_1 \wedge (y)[I_2] = c_2\} \leq \beta.$$

Basic set: For any $f : F_{2^w} \times F_{2^w} \rightarrow F_{2^w}$, where $f \in \{\vee, \wedge; \oplus,$ *logic* $shift_{r/l}(R_i, \alpha), rotate_{r/l}(R_i, \alpha), +, -\}$, and $f : F_{2^w} \rightarrow F_{2^w}$, where $f \in \{bitwise\ \neg\}$, there is an instruction $R_h = f(R_i, R_j)$, $R_h = f(R_i, \alpha)$, and $R_h = f(R_k)$, respectively. Integers are represented in two's complement binary notation and hence are in the range $[-2^{w-1} \ldots -1, 0, 1 \ldots 2^{w-1} - 1]$.

The instructions of the basic set implement $(1, \beta)$-local functions where $\beta \leq 2$; e.g., all logic instructions are $(1, 1)$-local, and the addition/subtraction are $(1, 2)$-local [60].

Extended set: This set includes all instructions implementing $(1, \beta)$-local functions with $\beta \leq w - 1$. For example, *variable* $shift_{r/l}(R_i, R_j)$ and $rotate_{r/l}(R_i, R_j)$, where $content(R_j) \in [0, w-1)$, are $(1, w-1)$-local.

Multiplication set: This set includes all instructions implementing $(1, \beta)$-local functions with $\beta \leq 2^{w-1}$. For example, $R_i\ mod\ R_j$[14], where $content(R_j) = p$, $2 < p < 2^{w-1}$, is $(1, p)$-local. Integer *multiplication* is $(1, 2^{w-1})$-local[15].

All integer, logic, and shift/rotate computation instructions of real ISAs with up to two operands fall into the three sets defined above. In fact, any computation function implemented by a *finite-state transducer* is $(1, \beta)$-local for a constant β. Note that in all other WRAM models w is a variable, and hence the instructions of the extended and multiplication sets become non-local since β is no longer constant.

However, as in all WRAM models [36], floating-point instructions are not included in cWRAM. These instructions are irrelevant to the concrete space-time bounds of optimal integer computations where instruction sequences are latency-bound; i.e., where an instruction depends on the results of another. Here integer operations are always faster for the same data size; e.g., typically twice as fast in most commodity ISAs. Similarly, non-computation instructions left out of cWRAM are irrelevant to these types of optimal integer computations.

Relationship with other WRAM computation instructions. The *basic set* is in all WRAM models; viz., the practical/restricted model [36], [60]. The *extended set* augments the basic set since its instructions do not violate the unit-cost execution requirement of WRAM; e.g., the (non-local) *variable shift* is in the practical RAM [60]. The *multiplication set* was excluded from all early WRAM models since its (non-local) instructions cannot meet the unit-cost requirement. A notable exception is the Circuit RAM, which allows variable instruction-execution cost [1]. In a concession to the small constant execution-time difference between multiplications and additions in real instruction sets, all recent unit-cost WRAM models include the multiplication set [16], [50], [65].

Function Simulation. Let functions $f, g : F_{2^w} \times F_{2^w} \rightarrow F_{2^w}$ be $(1, \beta_f)$- and $(1, \beta_g)$-local, respectively. Function f simulates function g if for all $x, y \in [0, 2^w)$, $f(x, y) = g(x, y)$, which implies that $\beta_f = \beta_g$. If $\beta_f \neq \beta_g$, the simulation of g by f causes a *simulation error*. This implies, for instance, that neither the *addition* nor the *multiplication* instructions can be simulated by any single other instruction without error.

Execution Model. Once a program's instructions are stored in memory and the processor registers are initialized, the *program counter* register, PC, is set to the index of the

[14]When a $R_i\ mod\ R_j$ instruction is unavailable, an optimal implementation by other cWRAM instructions exists; see Appendix C.

[15]This is for signed integer multiplication with single-word output. Unsigned integer multiplications with a two-word register output, R_{lo} and R_hi, is illustrated in Appendix C.

memory word denoting the next instruction (i.e., program line number) to be executed. The PC is incremented at the completion of each instruction, except when (1) a conditional-branch predicate evaluates to 1, (2) an unconditional branch instruction is executed, (3) an interrupt triggers, and (4) the *Halt* instruction is executed. In cases (1) and (2), the PC is either offset by g or set to R_k, whereas in case (3) the PC is set to the first instruction of an interrupt handler. Appendix C shows that conditional instruction execution is also useful; i.e., an instruction is executed only if a condition flag, such as the carry flag, is set in the special processor registers.

A program in which the execution of all branching instructions precede their targets is called a *loop-free program*. A program with no branch instructions is *straight-line*. Let $I_1, \ldots, I_n$ be a straight-line program. A program

repeat $I_1, \ldots, I_n$ *until* $pred(R_i, R_j)g = 0$

is called the *loop program*[16]. Alternatively, the conditional-branch instruction can be $pred(R_i)g$.

A loop program can implement synchronous I/O by *busy waiting*; e.g., if register R_i selects the *busy/done* status bit of a device-status register and $g = -1$, then one-operand instruction $pred(R_i)g$ represents a *busy waiting* loop program.

Running Time. Unit-time instruction execution implies the running time of a computation is the number of instructions executed until *Halt* or *error* output.

XI. Appendix B – Proofs

In the proofs of some theorems we use the well-known **Facts 1–4**, and let $p > 2$ be a prime.

Theorem 1.

Fact 1. If $q > 2$ is a prime power, every permutation of $\mathbf{F_q}$ is the composition of permutations induced by linear polynomials over $\mathbf{F_q}$ and by x^{q-2} [96].

If q is a power of two, a linear polynomial induces a permutation in $\mathbf{F_q}$ only if the leading coefficient is odd [73].

Fact 2 Let x, m be two positive integers such that $gcd(x, m) = 1$, and c a constant integer.

a) If U is a uniformly distributed variable in $[0, m)$, then $(x \times U)\ mod\ m$ and $(x \times U + c)\ mod\ m$ are also uniformly distributed in $[0, m)$ [88]; and

b) For any $y \in [0, m)$ there exists a unique $\alpha \in [0, m)$ such that $y = \alpha \times x\ mod\ m$ [82].

Important cases are (i) $x < m$ and $m > 2$ is a prime, and (ii) x is odd and m is a power of two.

Lemma 1.1. In any implementation of $f_i(x) = a_i \times x + a_{i-1}\ (mod\ p)$ that simultaneously minimizes space and time in cWRAM, the program schema

$mod(op_2(mod(op_1(operand_1, operand_2), p), operand_3), p)$,

$op_1, op_2 \not\equiv mod\ p$, is optimal and so is a Horner-rule step.

Proof. In the proof we use the following two claims.

Claim 1: At least three instructions are necessary to implement $f_i(x)$, the last of which must be $mod\ p$.

Proof. $f_i(x) = a_i \times x + a_{i-1}\ (mod\ p)$ is a permutation polynomial of $\mathbf{Z_p}$ (by **Fact 1**) whose implementation requires at least three instructions; i.e., it has four input variables, a_i, a_{i-1}, x, and p, and all instructions have at most two operands. Then $a_i \times x\ mod\ p$ also permutes x in $\mathbf{Z_p}$, and thus the size of its value set is p. Hence, its implementation requires a two-instruction sequence, where at least one instruction must have $\beta \geq p$. Furthermore, one of the two instructions must be $mod\ p$, since both cannot differ from $mod\ p$ and both cannot be $mod\ p$; i.e., in the former case, x would not be permuted in $\mathbf{Z_p}$ for all a_i, and in the latter both a_i and x could not be assigned the single input left unused. Finally, if $a_{i-1} \neq 0$, the third instruction must read and use it, and if $mod\ p$ is not the last instruction, $f_i(x)$ could not permute x for all a_i, a_{i-1}. □.

Claim 2: If space and time are simultaneously minimized, four instructions are necessary and sufficient to implement $f_i(x)$, and the second and the last are $mod\ p$.

Proof. If three instructions the last of which is a $mod\ p$ were sufficient, then the other two must be

$op_2((op_1(operand_1, operand_2), operand_3)$,

since $mod\ p$ has a single operand in addition to p. Then

$op_2(op_1(operand_1, operand_2), operand_3)$

must map the operand assigned to x to the p distinct integers of an interval $[i, i + p\ (mod\ 2^{w-1})), i = 0, \ldots, 2^{w-1} - p - 1$. Otherwise, the three instructions could not permute x in $\mathbf{Z_p}$, since $mod\ p$ can implement a one-to-one correspondence only between $[i, i + p\ (mod\ 2^{w-1}))$ and $[0, p)$. This means that at least one of $\{op_1, op_2\}$ must be $mod\ p$, and hence one of its operands must be assigned to p. Since the single remaining operand of $op_2((op_1(operand_1, operand_2), operand_3)$ cannot be assigned both a_i and a_{i-1}, four instructions are necessary, two of which must be $mod\ p$.

Now note that the second $mod\ p$ can be neither the first nor the third instruction executed. Otherwise, three instructions would be sufficient: a $mod\ p$ as the first instruction would be redundant since all inputs are in $\mathbf{Z_p}$, whereas a $mod\ p$ as the third instruction would be redundant since the fourth is already $mod\ p$. Hence, four instructions are sufficient when $op_1, op_2 \not\equiv mod\ p$; e.g., a Horner-rule step

$mod(add(mod(\cdot(a_i, x), p), a_{i-1}), p)$,

where add denotes the addition and $\cdot$ the multiplication instructions of cWRAM. □

Lemma 1.2. In any one-time honest evaluation of $f_i(x) = a_i \times x + a_{i-1}\ (mod\ p)$ that simultaneously minimizes space and time in cWRAM, the only possible assignment of $f_i(x)$'s variables to the operands of the optimal program-schema is:

$(operand_1, operand_2) \leftarrow (a_i, x)$, $operand_3 \leftarrow a_{i-1}$.

Proof. By **Fact 1**, every linear polynomial induces a unique permutation of x in $\mathbf{Z_p}$. Hence, all possible assignments of $f_i(x)$'s variables to the operands of instructions op_1 and op_2 of an honest evaluation must yield a linear polynomial and induce the same permutation as $f_i(x)$ in $\mathbf{Z_p}$. We show that two of the three possible operand assignments are excluded for any op_1, op_2 in a one-time honest evaluation.

1) $(operand_1, operand_2) \leftarrow (a_i, a_{i-1})$ and $operand_3 \leftarrow x$. For all op_1 in $mod(op_2(mod(op_1(a_i, a_{i-1}), p), x), p)$, op_2 must be a $\cdot\ (multiplication)$ instruction; otherwise, the assignment cannot yield a linear polynomial in $\mathbf{Z_p}$, since no

[16] Ibarra *et al.* [35] show that for any non-loop-free program an equivalent *while* $pred = 1$ *do* $I_1, \ldots, I_n$ *end* exists and its length is proportional with the original program. This obviously holds for *repeat-until* programs.

single cWRAM instruction can simulate multiplication in a honest one-time evaluation that minimizes space and time. Hence, there exist a_i and a_{i-1} that produce the permutation error $\cdot(mod(op_1(a_i, a_{i-1}), p), x) \neq a_i \times x \bmod p + a_{i-1}$, which the final $mod\ p$ cannot remove to yield $f_i(x)$; i.e., $mod\ p$ is the only instruction executed after op_2 in a one-time evaluation, and it merely maps the error from $[0, 2^{w-1})$ to $\mathbf{Z_p}$.

2) $(operand_1, operand_2) \leftarrow (a_{i-1}, x)$ and $operand_3 \leftarrow a_i$.

For all op_2 in $mod(op_2(mod(op_1(a_{i-1}, x), p), a_i), p)$, op_1 must be a $\cdot$ $(multiplication)$ for the same reason as in part 1). However, if $a_{i-1} = 0$, $mod(op_2(mod(\cdot(a_{i-1}, x), p), a_i), p)$ cannot induce a permutation in $\mathbf{Z_p}$ for any op_2 and a_i, since it maps all values of x into $mod(op_2(0, a_i), p)$. □.

Lemma 1.3. A Horner-rule step is uniquely optimal in a one-time honest evaluation of $f_i(x) = a_i \times x + a_{i-1}\ (mod\ p)$ that simultaneously minimizes space and time in cWRAM.

Proof. **Lemma 1.2** allows instruction assignments $op_1 \neq \cdot(multiplication)$ and $op_2 \neq add$ $(addition)$ to program schema $mod(op_2(mod(op_1(a_i, x), p), a_{i-1}), p)$. All such assignments can be partitioned into two possible simulation cases:

1) either $op_1 = \cdot, op_2 \neq add$ or $op_1 \neq \cdot, op_2 = add$; and

2) $op_1 \neq \cdot$ and $op_2 \neq add$.

We show that neither case is possible in an one-time honest evaluation.

1) Since $mod(\cdot(a_i, x), p) = y$ is uniform in $\mathbf{Z_p}$ (by **Fact 2a**), a simulation error $op_2(y, a_{i-1}) \neq y + a_{i-1}$ is generated for all $op_2 \neq add$. Similarly, for all $op_1 \neq \cdot$ there exists an error $mod(op_1(a_i, x), p) \neq y$. In the first case, $mod\ p$ is the only instruction executed after op_2 in a honest one-time evaluation, and $mod\ p$ merely remaps error from $[0, 2^{w-1})$ to $[0, p)$. For the second case, recall that the add has at most two values in any bit interval of its output, and thus $\beta_{add} = 2$. Hence, $mod(add(mod(op_1(a_i, x), p), a_{i-1}), p)$ cannot permute x, since $\beta_{add} < p$ and in any honest one-time execution $mod\ p$ is the only instruction executed after add.

2) For all $op_1 \neq \cdot$, there exists an a_i and x that cause the simulation error $op_1(a_i, x) \neq \cdot(a_i, x)$ in $[0, 2^{w-1})$ and $y' = mod(op_1(a_i, x), p) \neq mod(\cdot(a_i, x), p)$ in $\mathbf{Z_p}$. However, for any y' and x there exists a unique $a_i' \in \mathbf{Z_p^+}$, such that $y' = mod(\cdot(a_i', x), p)$ (by **Fact 2b**), and $a_i' \neq a_i$. Furthermore, by evaluation honesty, $mod(op_2(y', a_{i-1}), p)$ is a linear polynomial in $\mathbf{Z_p}$ for all y' and a_{i-1}. This means that both $mod(op_2(mod(\cdot(a_i', x), p), a_{i-1}), p)$ and $mod(add(mod(\cdot(a_i, x), p), a_{i-1}), p)$ must perform the same linear permutation of x, for all a_i', a_i, x and a_{i-1}. Since that permutation is unique (by **Fact 1**), $op_2 = add$ and, for $a_{i-1} = 0$, $a_i' = a_i$, which implies $op_1 = \cdot$. □

Lemma 1.4. There exists a permutation polynomial $P_d(\cdot)$, and input x over $\mathbf{Z_p}$ whose honest one-time evaluation in minimal cWRAM space and time requires a six-instruction Horner-rule program. That is, a Horner-rule program is uniquely optimal: no other program exists that can evaluate $P_d(x)$ is fewer than $d + 11$ words and $6d$ time units.

Proof.

Loop Existence. Let $a_d, \ldots, a_0 \in \mathbf{Z_p}$ and $a_d \neq 0$. Define polynomials

$$f_{d-1}(z_d, x) = z_d \times x + a_{d-1}\ (mod\ p)$$
$$\ldots$$
$$f_1(z_2, x) = z_2 \times x + a_1\ (mod\ p)$$
$$f_0(z_1, x) = z_1 \times x + a_0\ (mod\ p)$$

where $z_i = f_i(z_{i+1}, x)$, $0 \leq i < d$, and $z_d = a_d \neq 0$. Now choose $a_i \in \mathbf{Z_p}$ and set $X = \{x : x \in \mathbf{Z_p^+}\}$ for which all $z_i \neq 0, i > 0$. Then all polynomials $f_i(z_{i+1}, x)$ become linear permutation polynomials for all $x \in X$, by **Fact 1**. Furthermore, by composing the linear permutation polynomials using rule $z_i = f_i(z_{i+1}, x)$ we obtain degree d polynomial

$$P_d(x) = f_0(f_1(\cdots f_{d-1}(a_d, \cdot)\ \cdots, x), x)$$
$$= (\cdots (a_d \cdot x + a_{d-1}) \cdot x + \cdots + a_1) \cdot x + a_0\ (mod\ p),$$

which also permutes $x \in X$ for some choices of a_i. By construction, the honest one-time evaluation of $P_d(x)$ on input $x \in X$ in minimum space and time implies that each linear permutation polynomial $f_i(x)$ is evaluated honestly one-time in in minimum space and time. Then, by **Lemma 1.3**, the evaluation of each $f_i(x)$ requires a Horner-rule step. Hence, a loop is required for a sufficiently small memory when $d > 1$.

Loop Control. Without loss of generality, let the *conditional branch* follow the Horner-rule step; see the definition of a loop in Appendix A. Assume by contradiction that an additional variable (and instruction) is unnecessary to implement the branch predicate in a honest evaluation. Then the branch outcome can depend only on either (1) the content of the Horner-rule step's output register z_i or (2) the set of coefficients already read, since x and p are loop constants. In case (1), the branch outcome is independent of the memory size, which determines the number of loops necessary regardless of the evaluation result in z_i. Hence the evaluation cannot be honest. In case (2), polynomials $P_d(x)$ and $P_{d+1}(x) = P_d(x) \times x\ (mod\ p)$ must have identical branch outcomes, which also contradicts the honest-evaluation assumption.

Let d be the additional variable. Since none of the instructions of the loop body has a free input to read it, an additional instruction is necessary. Storage and execution time minimization implies that this instruction must also update d on every loop execution; i.e., an extra variable is unavailable to count up to d. Hence, the additional instruction must be $d := d - 1$, and the conditional branch instruction must be $pred(d)g$[17]. The resulting six-instruction cWRAM program provides the lower time bound and $d + 11$ words the lower space bound for evaluating $P_d(x)$. □

A similar proof applies for polynomials over $\mathbf{F_q}$, where q is a prime power. Here the important case is $q = 2^{w-1}$, and hence a_i must be odd (by **Facts 1** and **2**). Then we restate **Lemmas 1.1–1.3** to show that a multiply-and-add Horner-rule step is uniquely optimal for these linear permutation polynomials. Also, the proof of **Lemma 1.4** requires that, for all chosen inputs x, z_i has an odd value for all i. □

Theorem 2.

Fact 3 (e.g., *Construction 1, Theorem 2* [41], p. 6).

Let $P_d(\cdot) = \sum_{i=0}^{d} a_i \times x^i\ (mod\ p)$ be a polynomial of degree

[17] In the absence of $pred(d)g$ the simulation of sequence $d := d - 1, pred(d)g$ would require at least three integer instructions; viz., Fig. 1 [35].

$d > 0$ over $\mathbf{Z_p}$, and $u \in \mathbf{Z_p}$ a constant. Then,

$$Pr[x, a_0 \xleftarrow{\$} \mathbf{Z_p} : P_d(x) = u] = \tfrac{1}{p}.$$

If $x \neq 0$, the probability equals $\frac{1}{(p-1)}$.

Part 1. We use the total probability formula for disjoint events $x \neq 0$ and $x = 0$.

$Pr[x \xleftarrow{\$} \mathbf{Z_p}, \exists y \in \mathbf{Z_p},\ y \neq x : H_{d,k}(y) = H_{d,k}(x)]$
$= Pr[x \xleftarrow{\$} \mathbf{Z_p} : x \neq 0]$
$\times\ Pr[x \xleftarrow{\$} \mathbf{Z_p^+}, \exists y \in \mathbf{Z_p},\ y \neq x : H_{d,k}(y) = H_{d,k}(x)]$
$+\ Pr[x \xleftarrow{\$} \mathbf{Z_p} : x = 0]$
$\times\ Pr[\exists y \in \mathbf{Z}_p,\ y \neq 0 : H_{d,k}(y) = H_{d,k}(0)]$
$= \frac{p-1}{p} \times Pr[x \xleftarrow{\$} \mathbf{Z}_p^+, \exists y \in \mathbf{Z}_p,\ y \neq x : H_{d,k}(y) = H_{d,k}(x)]$
$+\ \frac{1}{p} \times Pr[\exists y \in \mathbf{Z}_p,\ y \neq 0 : H_{d,k}(y) = H_{d,k}(0)]. \qquad (*)$

To complete the proof, we bound the above probabilities.

Claim.
$Pr[x \xleftarrow{\$} \mathbf{Z}_p^+, \exists y \in \mathbf{Z}_p,\ y \neq x : H_{d,k}(y) = H_{d,k}(x)] \leq \frac{1}{(p-1)}$

Proof. Since $x \in \mathbf{Z}_p^+$, we have $y = \alpha \times x \ mod\ p$ (by **Fact 2b**), where $\alpha - 1 \neq 0$, since $y \neq x$. Then,
$H_{d,k}(y) - H_{d,k}(x)$
$= (v_d \oplus s_d)(\alpha^d - 1)x^d + \ldots + (v_1 \oplus s_1)(\alpha - 1)x = P_{d-1}(x) \times x$,
where $P_{d-1}(x) = (v_d \oplus s_d)(\alpha^d - 1)x^{d-1} + \ldots + (v_1 \oplus s_1)(\alpha - 1)$.

Hence,
$Pr[x \in \mathbf{Z}_p^+, \exists y \in \mathbf{Z_p}, y \neq x : H_{d,k}(y) = H_{d,k}(x)]$
$= Pr[x \in \mathbf{Z}_p^+ : P_{d-1}(x) \times x = 0]$
$= Pr[x \in \mathbf{Z}_p^+ : P_{d-1}(x) = 0]$.

Now recall that, by the definition of $H_{d,k}(\cdot)$, all s_i are uniform and independent of x and all v_i. Hence, $v_1 \oplus s_1$ is uniform and independent of x. Since $\alpha - 1 \neq 0$, polynomial $P_{d-1}(x)$'s free term, $(v_1 \oplus s_1)(\alpha - 1)$, is uniform in $\mathbf{Z_p}$ (by **Fact 2a**) and independent of x. Thus, when $d - 1 > 0$,

$Pr[x \in \mathbf{Z}_p^+ : P_{d-1}(x) = 0] = \frac{1}{(p-1)}$, by **Fact 3**.

When $d - 1 = 0$,

$Pr[x \in \mathbf{Z}_p^+ : (v_1 \oplus s_1)(\alpha - 1) = 0] = \frac{1}{p} < \frac{1}{(p-1)}$,
since $(v_1 \oplus s_1)(\alpha - 1)$ is independent of x and uniform. Hence,

$Pr[x \in \mathbf{Z}_p^+ : P_{d-1}(x) = 0] \leq \frac{1}{(p-1)}$. □

To bound $Pr[\exists y \in \mathbf{Z}_p,\ y \neq 0 : H_{d,k}(y) = H_{d,k}(0)]$, we note that $H_{d,k}(y) - H_{d,k}(0)$
$= (v_d \oplus s_d)(\alpha^d - 1)y^{d-1} + \ldots + (v_1 \oplus s_1)(\alpha - 1) = P_{d-1}(y)$.
Then we proceed as in the proof of the *Claim* above and obtain

$$Pr[\exists y \in \mathbf{Z}_p,\ y \neq 0 : H_{d,k}(y) = H_{d,k}(0)] \leq \tfrac{1}{(p-1)}.$$

Combining the above bounds in eq. (*) completes the proof.

Part 2 follows from **Fact 3** and the definition of $H_{d,k}(\cdot)$.

Part 3. Let $u_i \in \mathbf{Z_p}$ be k constants and $x_i \in \mathbf{Z_p}$ be k distinct values. By polynomial interpolation on the point set $\{(x_i, u_i)\}$, we obtain a system of k linear equations $H_{d,k}(x_i) = u_i$ with $d+1$ unknowns $v_i \oplus x_i$, k of which are independent. Fix any of the $d-k+1$ unknowns; e.g., fix $v_d \oplus x_d, \ldots, v_k \oplus x_k$. Evaluate the sums $(v_d \oplus x_d)x^d + \cdots + (v_k \oplus x_k)x^k$ and move the results to the right-hand side of the k equations. The new linear system, $H_{d,k}(x_i) = u_i - [(v_d \oplus x_d)x^d + \cdots + (v_k \oplus x_k)x^k] = u'_i$, has k equations with k unknowns and, by using its Vandermonde matrix, we obtain its the k unique solutions $v_i \oplus s_i = a_i$ for the distinct x_i. Thus,
$Pr[x_i \xleftarrow{\$} \mathbf{Z_p} : H_{d,k}(x_0) = u'_0, \ldots, H_{d,k}(x_{k-1}) = u'_{k-1}]$
$= Pr[x_i \xleftarrow{\$} \mathbf{Z_p} : v_0 \oplus s_0 = a_0, \ldots, v_{k-1} \oplus s_{k-1} = a_{k-1}]$.
Since $v_i \oplus s_i$ are k-independent and uniform,
$Pr[x_i \xleftarrow{\$} \mathbf{Z_p} : v_0 \oplus s_0 = a_0, \ldots, v_{k-1} \oplus s_{k-1} = a_{k-1}]$
$= Pr[x_i \xleftarrow{\$} \mathbf{Z_p} : v_0 \oplus_0 = a_0]$
$\times\ Pr[x_i \xleftarrow{\$} \mathbf{Z_p} : v_1 \oplus s_1 = a_1]$
$\ldots$
$\times\ Pr[x_i \xleftarrow{\$} \mathbf{Z_p} : v_{k-1} \oplus s_{k-1} = a_{k-1}] = 1/p^k$. □

Theorem 3.

By hypothesis, the honest evaluation code for $H_{d,k,x}(v)$ is committed to *at most* $k + 22$ words and $(6k - 4)6d$ time units after initialization. We show that these are also the lower bounds for any evaluation that simultaneously minimizes space and time. We distinguish two cases for the lower bounds of $H_{d,k,x}(v)$ depending on d.

1) $d \leq k - 1$. Since each word v_i is independent of s_i and stored in word w_i of M in cWRAM, each coefficient $v_i \oplus s_i$ must be computed. Note that $v_i \oplus s_i$ are k-independent since s_i are k-independent [65], [90]. Hence, the computation of any one coefficient cannot decrease the computation time and space of any others. Thus each $s_i = \sum_{j=0}^{k-1} r_j \times (i+1)^j \ (mod\ p)$ must be computed one time, and its honest computation requires a Horner-rule program of 6 instructions and at least $k + 3$ data words (i.e., k random values, the degree $k - 1$, input j, and modulus p), by **Theorem 1**, for a total of $k + 9$ words and $6(k - 1)$ time units.

Once a s_i is computed, a coefficient $v_i \oplus s_i$ requires at least 2 instructions (i.e., one for computing the index of v_i and the $exclusive\text{-}or$), and 2 data words; i.e., one for the index and one for s_i. Hence, each padded coefficient $v_i \oplus s_i$ requires at last $k + 13$ words and $6(k - 1) + 2 = 6k - 4$ time units. However, if d coefficients $v_i \oplus s_i$ are computed and stored after initialization, the one-time honest evaluation $H_{d,k,x}(v)$ by a Horner-rule program is optimal in an additional 6 instructions and 3 data words (i.e., degree d, input x, output z) for a total of $k + 22$ words by **Theorem 1**.

Now note that the Horner-rule program that computes s_i, the program that computes each coefficient $v_i \oplus s_i$, and the Horner-rule program that implements the outer loop for the evaluation of $H_{d,k,x}(v)$ have no instruction in common. Furthermore, neither has any instructions in common with the space-optimal processor-state initialization, ***I/O*** and register initialization (*Init*) programs in cWRAM. This implies that none of these programs can reuse instructions from the others or use extra space. Hence their composition in a $Horner(H_{k-1,k,x}(v))$ program, which is stored alongside processor-state initialization, *I/O* and *Init* code, in $k+22$ words and which executes in $(6k-4)6d$ time units after initialization is optimal.

2) $d > k - 1$. By honesty, the same $Horner(H_{k-1,k,x}(v))$ program must be used in this case, with the degree initialized to d. Assume by contradiction that not all Horner-rule steps between k and d need to be executed. Then at least one $i >$

$k-1$ coefficient $v_i \oplus s_i$ is not computed, and some other some value, a_i, is used at Horner-rule step i, $z_i = z_{i+1} \times x + a_i$, where $v_i \oplus s_i \neq a_i = v'_i \oplus s_i$. Hence, $H_{d,k,x}(v') \neq H_{d,k,x}(v)$ is evaluated, contradicting the honesty assumption. □

Lemma 4.

Fact 4 (e.g., *Fact 2.1* [88]). If p is a prime and $\alpha, \beta \in \mathbf{Z}_\mathbf{p}^+$, then $\alpha \times \beta \in \mathbf{Z}_\mathbf{p}^+$.

To bound $Pr[S : H_{d',k,y}(v') = H_{d,k,x}(v)]$, we use the total probability formula for disjoint events $x \neq 0$ and $x = 0$. If $x \neq 0$, for any $y \in \mathbf{Z_p}$, there exists a unique α such that $y = \alpha \times x \ mod\ p$, by **Fact 2b**. Since $S = \{r_j, x \xleftarrow{\$} \mathbf{Z_p}, 0 \leq j \leq k-1\}$, we denote $S^+ = \{r_j \xleftarrow{\$} \mathbf{Z_p}, x \xleftarrow{\$} \mathbf{Z}_p^+, 0 \leq j \leq k-1\}$.

$Pr[S : H_{d',k,y}(v') = H_{d,k,x}(v)]$
$= Pr[S : x \neq 0] \times Pr[S^+ : H_{d',k,\alpha\times x}(v') = H_{d,k,x}(v)]$
$+ Pr[S : x = 0] \times Pr[S : H_{d',k,y}(v') = H_{d,k,0}(v)]$
$= \frac{p-1}{p} \times Pr[S^+ : H_{d',k,\alpha\times x}(v') = H_{d,k,x}(v)]$
$+ \frac{1}{p} \times Pr[S : H_{d',k,y}(v') = H_{d,k,0}(v)]$
$\leq \frac{p-1}{p} \times Pr[S^+ : H_{d',k,\alpha\times x}(v') = H_{d,k,x}(v)] + \frac{1}{p}$.

The following claim completes the proof.

Claim. $Pr[S^+ : H_{d',k,\alpha\times x}(v') = H_{d,k,x}(v)] \leq \frac{2}{(p-1)}$.

Proof. Given polynomial $H_{d,k,x}(\cdot)$ and input v, we partition all adversary's choices $(H_{d',k,\alpha\times x}(\cdot), v') \neq (H_{d,k,x}(\cdot), v)$ into two mutually exclusive cases:

1) the index-ordered coefficients of one polynomial are included in the other's; i.e., for all i, $0 \leq i \leq \min(d', d)$, $(s_i \oplus v'_i) \times \alpha^i = s_i \oplus v_i$; and

2) there exists at least one (common) coefficient index where the two polynomials have different coefficients; i.e., there exists an index i, $0 \leq i \leq \min(d', d)$, such that $(s_i \oplus v'_i) \times \alpha^i \neq s_i \oplus v_i$.

Case 1). Since $(H_{d',k,\alpha\times x}(\cdot), v') \neq (H_{d,k,x}(\cdot), v)$, it must be that $d' \neq d$. Let $j > \min(d', d)$ be the smallest coefficient index of the polynomial difference $H_{d',k,\alpha\times x}(v') - H_{d,k,x}(v)$, such that either *a)* $s_j \oplus v_j \neq 0$, if $d > d'$, or *b)* $(s_j \oplus v'_j) \times \alpha^j \neq 0$, if $d' > d$.

a) We write $H_{d,k,x}(v) - H_{d',k,\alpha\times x}(v') = H_{d-j,k}(x) \times x^j$, where $H_{d-j,k}(x) = (s_d \oplus v_d) \times x^{d-j} + \cdots + (s_j \oplus v_j)$. Thus,
$Pr[S^+ : H_{d',k,\alpha\times x}(v') - H_{d,k,x}(v) = 0]$
$= Pr[S^+ : H_{d-j,k}(x) \times x^j = 0]$
$= Pr[S^+ : H_{d-j,k}(x) = 0] \leq \frac{1}{(p-1)}$,
since $x \neq 0$. The probability bound is $\frac{1}{(p-1)}$ when $d > j$, by **Theorem 2-2**, and $\frac{1}{p}$ when $d = j$, since $s_j \oplus v_j$ is uniform by the definition of **H**.

b) Similarly, $Pr[S^+ : H_{d',k,\alpha\times x}(v') - H_{d,k,x}(v) = 0]$
$= Pr[S^+ : H_{d'-j,k}(\alpha \times x) \times (\alpha \times x)^j = 0]$
$\leq Pr[S^+ : \alpha = 0]$
$+ Pr[S^+ : H_{d'-j,k}(\alpha \times x) \times (\alpha \times x)^j = 0 \mid \alpha \neq 0]$
$\leq Pr[S^+ : \alpha = 0] \ + \ \frac{1}{(p-1)}$,
by **Fact 4**, since $\alpha \times x \neq 0$, and by a similar reason as in *a)*.

To find the bound for $Pr[S^+ : \alpha = 0]$, note that if $\alpha = 0$, $H_{d',k,\alpha\times x}(v') = s_0 \oplus v'_0 = s_0 \oplus v_0$, by definition, in case 1). Then $H_{d,k,x}(v) - H_{d',k,\alpha\times x}(v') = H_{d,k}(x) - (s_0 \oplus v_0)$, since $H_{d,k,x}(v) = H_{d,k}(x)$ in a one-time evaluation. Thus
$Pr[S^+ : \alpha = 0] \leq Pr[S^+ : H_{d,k}(x) - (s_0 \oplus v_0) = 0]$
$= Pr[S^+ : H_{d-1,k}(x) \times x = 0] = Pr[S^+ : H_{d-1,k}(x) = 0]$,
where $H_{d-1,k}(x) = (s_d \oplus v_d) \times x^{d-1} + \cdots + (s_1 \oplus v_1)$ and $x \neq 0$. By similar reasoning as above,
$Pr[S^+ : H_{d-1,k}(x) = 0] \leq \frac{1}{(p-1)}$. Thus, for sub-case *b)* $Pr[S^+ : H_{d',k,\alpha\times x}(v') - H_{d,k,x}(v) = 0] \leq \frac{2}{(p-1)}$.

Case 2). Let i be the minimum coefficient index where the two polynomials have different coefficients; i.e., $(s_i \oplus v'_i) \times \alpha^i \neq s_i \oplus v_i$. Then we can write $H_{d',k,\alpha\times x}(v') - H_{d,k,x}(v) = P(x) \times x^{i+1} + ((s_i \oplus v'_i) \times \alpha^i - (s_i \oplus v_i)) \times x^i$.

a) If the choice of $H_{d',k,\alpha\times x}(v')$ implies $P(x) = 0$,
$Pr[S^+ : H_{d',k,\alpha\times x}(v') - H_{d,k,x}(v) = 0 \mid P(x) = 0] = 0$,
by **Fact 4**, since both $(s_i \oplus v'_i) \times \alpha^i \neq (s_i \oplus v_i)$ and $x \neq 0$.

b) If the choice of $H_{d',k,\alpha\times x}(v')$ implies $P(x) \neq 0$,
$Pr[S^+ : H_{d',k,\alpha\times x}(v') - H_{d,k,x}(v) = 0 \mid P(x) \neq 0]$
$= Pr[S^+ : P(x) \times x^{i+1} + (s_i \oplus v'_i) \times \alpha^i - (s_i \oplus v_i)) \times x^i = 0]$
$= Pr[S^+ : x = ((s_i \oplus v_i) - (s_i \oplus v'_i) \times \alpha^i) \times P(x)^{-1}]$
$= \frac{1}{(p-1)}$,
since both $(s_i \oplus v'_i) \times \alpha^i - (s_i \oplus v_i)$ and $P(x)$ are independent of $x \xleftarrow{\$} \mathbf{Z}_\mathbf{p}^+$. This is because $(s_i \oplus v_i)$ is independent of x for all i, by the definition of **H**, and hence $P(x)$'s free coefficient, $(s_{i+1} \oplus v'_{i+1}) \times \alpha^{i+1} - (s_{i+1} \oplus v_{i+1})$ is independent of x.

The claim follows since the evaluation is one-time and the adversary picks the the highest bound of the mutually exclusive cases *1-a), 1-b)*, and *2)*, which is $\frac{2}{(p-1)}$. □

Theorem 5.

We only need to prove that $Pr[S : H_{d,k,x}(v) = predictable] \leq \frac{3}{p}$. After executing Horner-rule step i ($0 < i \leq d$), the adversary has two mutually exclusive choices to predict result z_0:
1) output z_0 after executing a (possibly empty) instruction sub-sequence of the next Horner-rule step, $i-1$; or
2) output z_0 after executing a non-empty instruction sequence that is not a sub-sequence of the next Horner-rule step, $i-1$[18].

We bound the probability of adversary success in both cases.

Case 1). If the prediction is made when executing instructions in $Horner(H_{d,k,x}(v))$'s inner loop, the adversary would have no higher probability of success than if the prediction is made when after executing outer-loop instructions. For example, in the former case, adversary would have to predict successfully *both* the step's coefficient *and* z_0 rather than only z_0 in the latter. Hence, we only need to consider z_0 predictions made in the outer loop. In this case, s/he cannot derive any prediction benefit from interrupting execution *between* an integer (i.e., $mod\ 2^{w-1}$) instruction and the $mod\ p$ reduction of its output, nor from interrupting execution between the two instructions that compute the next coefficient, $v_{i-1} \oplus s_{i-1}$.

The adversary can output the prediction z_0 at any of the following three mutually exclusive points in outer-loop code.

- when loop $i-1$ begins execution: $z_i = u_i$;
- after computing coefficient $v_{i-1} \oplus s_{i-1} = w_{i-1}$;

[18]Note the case where the adversary executes an arbitrary sequence of instructions followed by the instructions of a Horner-step falls into the event of **Lemma 2**. This is because the arbitrary sequence must modify at least a processor register or a memory word, v_i. Thus, the subsequent execution of a Horner-rule instructions no longer implement any Horner-rule step.

- after computing $u_i \times x \ mod \ p = y_{i-1}$.

By definition, there are no other useful prediction points in the outer loop when $i - 1 > 0$. That is, the point after the last instruction of a Horner-rule step, namely "after computing $z_{i-1} = (y_{i-1} + w_{i-1}) \ mod \ p$," is the same as "when loop $i-2$ begins execution: $z_{i-1} = u_{i-1}$." The value $z_{i-1} = u_{i-1}$ is known only in the next loop, $i-2$. Let $u_i, y_{i-1}, w_{i-1} \in \mathbf{Z_p}$ be constants. Then

- $Pr_{i>1}[S : H_{d,k,x}(v) = z_0 \mid z_i = u_i]$ $= Pr_{i>1}[S : u_i \times x^i + (v_{i-1} \oplus s_{i-1}) \times x^{i-1} + \cdots + (v_0 \oplus s_0) = z_0] = \frac{1}{p}$, by **Fact 3**, since S implies $x, (s_0 \oplus v_0) \stackrel{\$}{\leftarrow} \mathbf{Z_p}$. Similarly,
- $Pr_{i>1}[S : H_{d,k,x}(v) = z_0 \mid v_{i-1} \oplus s_{i-1} = w_{i-1}]$ $= Pr_{i>1}[S : (u_i \times x \ mod \ p + w_{i-1}) \ mod \ p \times x^{i-1} + \cdots + (v_0 \oplus s_0) = z_0] = \frac{1}{p}$; and
- $Pr_{i>1}[S : H_{d,k,x}(v) = z_0 \mid u_i \times x \ mod \ p = y_{i-1}]$ $= Pr_{i>1}[S : (y_{i-1} + w_{i-1}) \ mod \ p \times x^{i-1} + \cdots + (v_0 \oplus s_0) = z_0] = \frac{1}{p}$;

Since the above prediction points are mutually exclusive and one-time per evaluation, the maximum probability of success at step $i > 1$ is $\frac{1}{p}$. To output prediction z_0 with higher probability of success the adversary would have to wait until the last Horner-rule step, namely when $i = 1$, and output z_0 after $z_0 = y_0 + w_0 \ (mod \ p)$ is executed. However, s/he would have to know it is the last step ($i = 1$), which implies that either a) s/he must execute the last two instructions or b) she must modify the loop control instructions of the $Horner(H_{d,k,x}(\cdot))$ code. In case a) s/he would have to use $k + 22$ words and execute the same number of (unit-time) instructions as $Horner(H_{d,k,x}(\cdot))$, contradicting the hypothesis that $H_{d,k,x}(\cdot)$ is predictable in $(m, t) < (k + 22, (6k - 4)6d)$. In case b) the adversary's probabilty of success is bounded by **Lemma 4** and hence is no higher than $\frac{3}{p}$.

Case 2). For $i > 1$, the output register of $Horner(H_{d,k,x}(\cdot))$ code $z_i = (s_d \oplus v_d) \times x^{d-i} + \cdots + (s_i \oplus v_i) = H_{d-i,k}(x)$ in any one-time evaluation. By **Theorem 2-2**, z_i is uniformly distributed, and by the definionon of **H**, $s_i \oplus v_i$ is also uniformly distributed. Thus, neither conveys any information about $z_0 = H_{d,k,x}(v))$ to the adversary. Hence, the sequence of instructions executed after step i yields a polynomial evaluation $H_{d',k,y}(v'))$ that equals $H_{d,k,x}(v))$ with probability no higher than $\frac{3}{p}$, by **Lemma 4**. If $i = 1$, the probability of outputting a correct prediction in (m, t) is no higher than $\frac{3}{p}$, by the same argument as in *Case 1*. □

Corollary.

1. If v' and v are two distinct strings of $log \ p$ bit words, $Pr[S, v' \neq v : H_{d',k,x}(v') = H_{d,k,x}(v)] \leq \frac{4}{p}$.

Proof. Using the total probability formula and upper bounding as in the proof of Lemma 4, we have
$Pr[S, v' \neq v : H_{d',k,x}(v') = H_{d,k,x}(v)]$
$\leq \frac{p-1}{p} \times Pr[S^+, v' \neq v : H_{d',k,x}(v') = H_{d,k,x}(v)] + \frac{1}{p}$.
Since $\alpha = 1$ the probability bound in each of the three possible adversary choices in the proof of Lemma 4's *Claim* (i.e., Cases *1-a, 1-b* and *2*) is at most $\frac{1}{p-1}$. Here, the adversary can exercise all three choices since the evaluation is not one-time. Hence, $Pr[S^+, v' \neq v : H_{d',k,x}(v') = H_{d,k,x}(v)] \leq \frac{3}{p-1}$. □

The k-independence proof is similar to that of Theorem 2-3.

2. The proof of this part follows directly from Theorem 5. □

Theorem 6

Part a) Any omission of an instruction execution in the *processor-state initialization*, *Input*, or *Init* programs can be caused by (1) absence of the instruction from a given memory location of the initialized instruction sequence (e.g., incorrect instruction or no instruction at that location) or (2) incorrect instruction-execution order of a correctly initialized instruction sequence in memory, or both. Below we say that a sequence of distinct memory words is unique if no two words are equal and no other sequence of the same number of words can be found in the same memory locations.

Claim 1. There exists a verifier choice of instruction encoding and sequencing for the three programs above such that the lower $log \ p$ bits of their memory words form a unique sequence of distinct words of the input v to $Horner(H_{d,k,x}(\cdot))$. Hence, absence of any instruction from memory at a given memory location in the sequence modifies at least one word of v.

Proof. The verifier can use the following instruction encoding and sequencing for the three program above. For the straight-line *processor-state initialization* program, all disable (e.g., asynchronous events, caches, virtual memory, TLBs) operations and all power-off operations for stateless device are executed by instructions with unique *opcodes* or single operands, or both. At least one of the operands of all two-operand instructions is unique; e.g., setting clock frequencies, *I/O* register bits. Hence, their (*opcode, unique operand*) pairs are unique for all verifier's programs. All lower $log \ p$ bits of the memory words in this program form a unique instruction sequence of distinct words of v.

The busy-waiting *Input* program, which follows the *processor-state initialization* in a straight line, comprises the status-reading instruction for the uniquely named *I/O* device of the verifier's channel followed by a conditional backward-branching instruction (Appendix A), which is unique in all three program sequences. Hence, the lower $log \ p$ bits in the two memory words are unique and distinct at their memory locations.

The *Init* program comprises a straight-line sequence of *Load* instructions that follow those of the *Input* program in straight line. This sequence is unique in the lower $log \ p$ bits of its memory words because its encoding partitions both the general-purpose registers and the k random numbers such that no two numbers are loaded into the same register; i.e., (*Load, operand*$_i$) $\neq$ (*Load, operand*$_j$), $i \neq j$.

None of the three programs above shares any instruction with the other two. □

Claim 2. There exists a verifier choice of instruction encoding and sequencing for the three programs above such that an incorrect execution order of a correctly initialized instruction sequence modifies at least one word of input v to $Horner(H_{d,k,x}(\cdot))$.

Proof. Let the straight-line *processor-state initialization* program comprise a sequence of prefix instructions followed by a sequence of suffix instructions. Let the verifier choose the prefix to be the instructions that modify the lower $log \ p$ bits

of special processor-state registers. Then if the execution of any prefix instructions is omitted by the boot loader's transfer of control, or by a return from a surreptitiously triggered interrupt whose handler causes a forward branch, some word of v representing the lower $log\ p$ bits of a special processor-state register differs from the verifier's choice, and hence is modified.

Recall that, except the conditional backward-branching instruction of the busy-waiting the *Input* program, all instructions of the three programs form a straight line. Now assume that no instruction-execution omission occurs in the prefix instructions of the *processor-state initialization* program. Hence the omission can only occur in the instruction sequences following the prefix. However, this requires either that a forward-branching instruction exists in these instruction sequences or that a return from an interrupt causes a forward branch, or both. The former case contradicts the claim's hypothesis since no forward branch instruction exists in the verifier's programs, whereas the latter contradicts the assumption the the prefix instructions, which disable all asynchronous events, have been executed. □

Part b) A control flow deviation from the verifier's protocol on the device can occur either 1) outside or 2) inside the optimal $Horner(H_{d,k,x}(\cdot))$'s code execution, or both. We show that in both cases either the input v to $Horner(H_{d,k,x}(\cdot))$ is modified, or the optimality bounds of $Horner(H_{d,k,x}(\cdot))$ program are violated, or both. The former occurs with probability at most $\frac{3}{p}$, by *Part a)* and **Lemma 4**, whereas the later with probability at most $\frac{3}{p}$, by **Theorem 5**.

Case 1) The control flow deviation can occur either *before* or *after* the $Horner(H_{d,k,x}(\cdot))$'s code execution, or both. If the control flow deviation occurs *after* $Horner(H_{d,k,x}(\cdot))$ executes, a future-posted asynchronous event must trigger and the a surreptitiously initialized interrupt handler must execute some verifier unaccounted instructions. This means that some event-enabled bit has remained set at the time of the event, which implies that the execution of an event-disable instruction has been omitted in the verifier's processor-state initialization. By a similar argument as in *Part a)*, the input v to $Horner(H_{d,k,x}(\cdot))$ is modified.

If the flow of control deviates *before* $Horner(H_{d,k,x}(\cdot))$ executes, the deviation can happen only in the *Init* program, by the definition of the verifier's protocol. This implies that either (i) the sequence of instructions that exists in memory differs from the verifier's choice, or (ii) the verifier's choice of instructions are executed in an incorrect order and/or some surreptitiously triggered interrupt occurs. Then either input v to $Horner(H_{d,k,x}(\cdot))$ is modified (by *Part a)*) or the surreptitiously installed interrupt handler must execute at least two instructions violating $Horner(H_{d,k,x}(\cdot))$'s time bound.

Case 2) Any control flow deviation implies that either (i) a different instruction sequence from $Horner(H_{d,k,x}(\cdot))$'s is initialized in memory or (ii) same instruction sequence is initialized but it is executed in a different order. In case (i), if a different number of instructions are initialized in memory and executed, then either the result's timeliness assumption is contradicted or a violation of $Horner(H_{d,k,x}(\cdot))$ bounds occurs. Alternately, the correct number of instructions can be initialized in memory but in a different sequence; e.g., recall that multiple program encodings of $Horner(H_{d,k,x}(\cdot))$ can yield the same optimal bounds. However, as shown in *Claim 3* below, the lower $log\ p$ bits of the verifier's $Horner(H_{d,k,x}(\cdot))$ program form a unique sequence of distinct words of the input v. Hence, a different encoding of a Horner-rule program from the verifier's implies that some word of input v is modified.

For case (ii), assume the verifier's sequence of $Horner(H_{d,k,x}(\cdot))$ instructions exists in memory. However, its instructions are either executed in an different order from the verifier's or it is interrupted by a surreptitiously triggered interrupt before a modified input v can be used in a coefficient computation. In the the former case, at least an additional forward branch instruction must be executed between *Init* and before $Horner(H_{d,k,x}(\cdot))$'s loop control, which contradicts the assumption made. In the later, the surreptitiously installed interrupt handler must execute at least two instructions violating $Horner(H_{d,k,x}(\cdot))$'s time bound.

Claim 3. There exists a verifier's choice of $Horner(H_{d,k,x}(\cdot))$ instructions such that the lower $log\ p$ bits of every instruction form a unique sequence of distinct words of v; i.e., for any two instructions $i \neq j$, (*opcode*$_i$*, operand*$_i$) $\neq$ (*opcode*$_j$*, operand*$_j$).

Proof. Let *opcode*$_i$ denote the operation code and *operand*$_i$ the operand adjacent to it in the encoding of instruction i. Then

1) *opcode*$_i \neq$ opcode$_j$, $i \neq j$, by the unique optimality of Horner-rule program instructions (**Lemmas 1.2–1.4**).

2) *operand*$_i \neq$ operand$_j$, $i \neq j$. This is the case because in both the inner and outer loops of $Horner(H_{d,k,x}(\cdot))$ code where the two loops have pairs of instructions with the same *opcode*, the instructions have different *operand*$_i$. That is,

- the two loops assign different input variables to the same *opcodes* of a Honer-rule step and loop-control instructions;
- the only common variable of the two loops is s_i, and the verifier can assign it to the left operand (e.g., of the instruction that produces it) in the inner loop, and the right operand (of the instruction that uses it) in the outer loop. □

Theorem 7

Theorem 6 implies that if the verifier's protocol returns a correct and timely results, no malware instructions exist in the verifier's initialized programs (i.e., *processor-state initialization*, *Input*, *Init*, and $Horner(H_{d,k,x}(\cdot))$) except with probability at most $\frac{6}{p}$. Nevertheless, the device memory may still contain a malware instructions in the device boot loader; i.e., the device boot loader is not the verifier's. Then at least one *opcode* in some sequence of instructions or the number of boot loader instructions in the device's memory, or both, must differ from those chosen by the verifier. Hence, if any malware exists anywhere in the boot loader, the lower $log\ p$ bits some word v_i of input v must differ from the verifier's choice. Since no w-bit memory word i can be modified *after* the verifier's protocol started, an incorrect v_i must be used in computing a coefficient, $s_i \oplus v_i$. However, if the result $H_{d,k,x}(v)$ arrives in time $t_0 + (6k-4)6d$, the probability of any incorrect v is used is at most $\frac{3}{p}$ by **Lemma 4**.

Furthermore, by the verifier's choice of concurrent transaction order and duration, the results received at times $t_{0_i} + t_i$ imply no device can help another device circumvent its optimal

bounds. Finally, external proxy and memory copy attacks are ruled out since communication with external devices is always detectable by the verifier because of the added delays. □

Theorem 8.

By **Theorem 7**, collision bound of the **Fact**, and the definition of the RoT establishment, the probability of verification success is $(1-\epsilon(n))(1-c\cdot 2^{w-1})$. Since $p < 2^{w-1}$, if $n = 1$, $[1-\frac{9c}{p}]\times[1-\frac{c}{2^{w-1}}] > [1-\frac{9c}{p}]\times[1-\frac{c}{p}] > 1-\frac{10c}{p}+\frac{9c^2}{p^2}$ $> 1-\frac{10c}{p}$. □

Theorem 9.

The proof follows directly from the following claim.

Claim If the verifier accepts, the integrity of the control flow is maintained a) within each segment and b) between segments during the verifier's protocol for the n-segment memory model, except with probability at most $\frac{9n}{p}$.

Proof. a) By definition, if the verifier accepts the result, then all independent verifications of all n segments must have passed. This implies that control flow integrity is maintained in each memory segment independently of any other segment, except with probability no higher than $\frac{6}{p}$, by **Theorem 6-b**.

b) The control flow between memory segments is defined by the random sequential selection of each segment; viz., Section VI-C. By the definition of the protocol, the *Input* instruction sequence of a segment is modified such that it unconditionally transfers control to the next random segment selection. However, by a similar proof by **Theorem 6-a**, if the verifier accepts the result of a segment evaluation, any instruction-execution omission in the modified *Input* program cause a modification of the input$v$ to $Horner(H_{d,k,x}(\cdot))$, which is detected except with probability at most $\frac{3}{p}$, by **Lemma 4**.

Finally, if the verifier accepts, the boot loader's last instruction must have transfered control to one of the segments, and by the *Claim*, the flow of control remains within the instructions of each memory segment. □

XII. Appendix C – From the Concrete Word RAM Model to Real Processor Implementations

A. Implementing the Horner-rule Steps

When implemented on commodity processor architectures, the space-time optimality of the cWRAM program $Horner(H_{d,k,x}(\cdot))$ on input v depends primarily on the performance of the Horner-rule steps. The optimal implementation of both the loop control and coefficient $s_i \oplus v_i$ computation is easily achieved on these processors. The Horner-rule steps are defined on *unsigned integers* as $z = (s_{i+1} \oplus v_{i+1}) \times x + (s_i \oplus v_i)\ (mod\ p)$, $i = d-1,\ldots,0$, for the outer loop, and $y = r_{k-j} \times (i+1) + r_{k-j-1}\ (mod\ p)$, $j = 1,\ldots,k-1$, for the inner loop. Hence, the implementation of these Horner-rule steps in different commodity processor architectures illustrates the practical relevance of the results presented herein.

Division-based Implementations. As in cWRAM, the $mod\ p$ implementation of the Horner-rule steps avoids all register carries. In practice, many real processors include the mod (aka., integer *division-with-remainder*) instruction; e.g., Intel x86, AMD, MIPS, IBM PowerPC, SPARC V8 (with special output register), RISC V (with division fused with the remainder), among others. Lower end processors include only the ordinary integer *division-without-remainder*; e.g., ARM Cortex A15 and above and the M3-M4 and R4-R7 series. In these processors, the mod instruction is typically implemented by two instructions: an integer division followed by a (three-operand) multiply-and-subtract. On processors limited to two-operand instructions, mod requires three instructions as the multiply-and-subtract needs two instructions. As expected, the use of mod instructions lowers the memory bounds of the Horner-rule step; viz., the proof of Theorem 1.

Unlike the cWRAM model where the mod instruction has unit cost like all others, in real processors it is more expensive than other instructions such as multiplication or addition [26], [33] in terms of both execution time and energy use. In fact, low-end processors lack even the ordinary integer division-without-remainder – not just mod – due to its higher execution time; e.g., ARM Cortex A5, A8, A9[19]. However, when computing the Horner-rule step all division instructions, not just the mod instruction, can be avoided at the cost of higher memory bounds.

Division-less Implementation. A Horner-rule step is implemented by a unsigned integer multiplication and two addition instructions [14]. Register carries are either handled by single conditional additions or avoided by judicious choice of x [49]. In a full polynomial evaluation by the Horner rule the reductions $mod\ p$ are postponed until the final Horner-rule value is output.

Let p denote the largest prime that fits into a w-bit word. The first Horner-rule step z can be expressed as $z = a_{i+1}\cdot x + a_i\ (mod\ p)$, where $a_{i+1} = s_{i+1} \oplus v_{i+1}$ for $i = d-1,\ldots,0$. Let the product $a_{i+i}\cdot x$ be implemented by an unsigned-integer multiplication instruction with double word output in registers R_{hi} and R_{lo}, and $p = 2^w - b$, where p is the highest prime that fits in a w-bit word. Then $z = a_{i+1}\cdot x + a_i\ (mod\ p) = R_{hi}\cdot 2^w + R_{lo} + a_i\ (mod\ p) = b\cdot R_{hi} + R_{lo} + a_i\ (mod\ p)$, since $2^w = b\ (mod\ p)$. Next, the register carries caused by additions are handled by conditional additions of the unaccounted for 2^w to z; i.e., $z + 2^w = z + b\ (mod\ p)$. [Equivalently, reduce z $(mod\ p)$: $z - p = z - (2^w - b) = z + b\ (mod\ p)$.] In contrast, the register carry in the integer multiplication $b\cdot R_{hi}$ is avoided by picking $x \le \lfloor \frac{2^w}{b} \rfloor$ at the cost of a negligibly higher collision probability. The register carries of the second Horner-rule step, y, above is implemented in a similar way as for z.

In the evaluation of a randomized-polynomial, the final reduction $z\ (mod\ p)$ comprises the test $z > p$ and the conditional subtraction by $z - p$, since register carries are already handled[20]. The conditional testis implemented by a single three-operand instruction or by two instructions when only two-operand instructions are supported. Similarly, the final $y\ (mod\ p)$ reduction is performed at the end of each s_i evaluation.

A division-less implementation of the Horner-rule step with only eight instructions (without counting the final $mod\ p$ reduction) has been available for an Intel x86-32 class processor

[19] If needed, the ordinary integer division by constant p can be simulated by a multiplication and a shift instruction.

[20] When $w = 64$ and $p = 2^{61} - 1 < 2^w$, the reduction of $z\ (mod\ p)$ when $p < z < 2^{64}$ is preformed as $z = a\cdot 2^{61} + b\ (mod\ p)$, where $0 \le a, b < 2^{61}$. Hence, $z = (z\ \mathbf{div}\ 2^{61}) + (z\ \mathbf{mod}\ 2^{61})$ [48]. The integer division operation, **div**, requires a right shift instruction, and **mod** requires a bitwise *and* instruction with the mask $2^{61} - 1$, which requires the third instruction.

[49], where $w = 32$ and $p = 2^{32} - 5$. A MIPS processor requires two additional *move* instructions since its R_{hi} and R_{lo} registers are not directly addressable. These programs far exceed the four-instruction cWRAM implementation, which nevertheless increases the measured time bound in practice.

Note that the time bound of division-less implementations intimately depends on the type of arithmetic for a given word size. A CPU performing w-bit arithmetic on $2w$-bit words needs *many more* instructions to implement the Horner-rule step than a CPU performing w-bit arithmetic; e.g., an efficient forty-instruction implementation exists for a 32-bit CPU operating on 64-bit words ($p = 2^{64} - 59$), and another one for 64-bit CPU arithmetic for 128-bit words ($p = 2^{127} - 1$) [48], [49].

Optimal Space-Time Choice. Eliminating both the mod and ordinary integer-division instructions in real processor implementations yields lower time bounds and higher space bounds for evaluations of a Horner-rule step. In fact there exist multiple space-time optimal bounds even on a single processor ISA. However, every *distinct* space-time optimal implementation has a different program encoding for the Horner-rule program and hence a *different* input v to the $Horner(H_{d,k,x}(v))$ program. Hence, by Part 2 of the Corollary of Section IV-D, none of the different optimal implementations yields a higher chance of adversary success in establishing malware-free states.

Nevertheless, optimal space-time implementations that minimize the time bound are often preferable in devices with large primary memories where randomized-polynomial evaluations may take up to a few minutes for very large k; viz., Section VII. For example, to minimize the time bound of a division-less implementation of the optimal Horner-rule step for a specific processor model and ISA instance, one can use a stochastic superoptimization technique designed for short, loop-free, fixed-point instructions [58], [74]. When given this target implementation and the minimum time as the optimization criterion, a superoptimizer produces the time-optimized minimum-space program for that processor and model; e.g., the STOKE tool use for the Intel x86-64 ISA, which is generally considered to be the most complex instance of a CISC architecture [75]. Program synthesis tools may also be applicable [81].

B. Choosing k General Purpose Registers

The space-time optimality of the $Horner(H_{d,k,x}(\cdot))$ program on input v on commodity processor architectures also depends on the number of general purpose registers available to hold the k random numbers for computing $s_i = \sum_{j=0}^{k-1} r_j \times (i+1)^j \ (mod\ p)$. The $k + 8$ data words required for randomized polynomial evaluation (Section IV-B) are input and initialized in general purpose registers by *Init* such that they occupy all registers that affect the optimal space-time bounds of the randomized polynomials. This requires the analysis of a processor's ISA so that an appropriate value of $k > 1$ is selected.

For example, typical ARM processors have sixteen general purpose registers per mode, including the PC register. Additional registers exist for dedicated use in privileged mode and floating point instructions, and none of these can be used to bypass optimal bounds for randomized polynomial evaluation. Of the fifteen general purpose registers available for divisions-less evaluation, two are used for the output of integer multiplication, only one of which is accounted for among the $k + 8$ data words for randomized polynomial evaluation in cWRAM. Hence, at most fourteen registers hold the $k + 8$ data words, and thus k is at most six.

In MIPS I processors, there are thirty-two general purpose registers of which one is the source of constant zero and the other is the extra register unaccounted for by the output of integer multiplication in cWRAM. Since at most $k + 8$ resisters are required for randomized polynomial evaluations, k is at most twenty-two. The other thirty-two register file are dedicated to floating point instructions and are unusable for randomized polynomial evaluation since they slow down latency-bound integer computations considerably.

In other processors, such as the Intel x86-32, fewer than eight general purpose registers are available. In this case, the k random values are allocated to these registers and the rest of $k + 8$ data values required by randomized polynomial evaluation in cWRAM are allocated to the memory area. This is done because coefficient evaluations based on the k random numbers and the Horner-rule step are essential to optimal evaluations. Hence, the value of k is less than eight; e.g., four. Of course, a higher value for k is expected for the Intel x86-64 architectures where more registers are available. However, similar considerations apply.

C. Mapping Instruction-Word Strings into $\mathbf{Z_p}$ Integer Strings

Recall that p was chosen to be the largest prime that fits into a word of w bits. This implies that some of the values of the word-aligned instruction strings could not fit into $\mathbf{Z_p}$ integers, unlike the verifier-chosen constants which fill unused memory. However, the shorter (*opcode, single-operand*) pairs both fit into these integers and align with the least significant bits of a word in cWRAM. Furthermore, the encoding of the sequence of (*opcode, single-operand*) pairs of the verifier-chosen instructions is unique; viz., the proof of Theorem 6-a. Hence, the mapping from the unique sequence of verifier-chosen word (e.g., instruction) strings to a unique sequence of $\mathbf{Z_p}$ integer strings is preserved without any additional memory initialization action. Although this mapping requires us to *separate* the establishment of malware free states (Theorem 7) before RoT establishment (Theorem 8), the separation can be beneficial in practical applications. For example, consider on-demand I/O channel isolation [95]. A software-implemented verifier of a malware-free application needs to establish only that the controller of its newly allocated device, which is taken from a malware-infested operating system is, in fact, malware-free. Requiring RoT establishment is for the entire multi-device system is impractical here.

Aligned Encodings in $\mathbf{Z_p}$. In some processor architectures, such as the Intel x86 processors and their successors, the alignment of the (*opcode, single-operand*) pairs with the least significant bits of a word – as assumed in cWRAM – is maintained. However, in other processors, such as MIPS and ARM, it is not. The (conditional) *opcode*, *addressing mode*, and *operands* specifications are aligned with the most significant bits, so that part of the unique *opcode* encodings may be chopped off. For example, if $w = 64$ and $p = 2^{64} - 59$, then some of the least significant six bits may be chopped off and the sequence of (*opcode, single-operand*) pairs may no longer be unique. In such cases, the uniqueness of this sequence is

easily restored at the cost of a single additional instruction execution during the evaluation of coefficients $s_i \oplus v_i$, without affecting the optimality of the $Horner(H_{d,k,x}(\cdot)$ program. That is, the processor register which contains v_i is either reversed (e.g., by executing a RBIT instruction in ARM processors), or appropriately rotated/shifted (e.g., as in MIPS processors). Of course, the endianness of the instruction and data sequences *in memory* remains unaffected in any case.

Alignment-free encodings in $\mathbf{Z_p}$. Alignment-free encodings of instruction words into $\mathbf{Z_p}$ integers are supported if we perform an additional unique word-string to integer-string mapping during Horner rule program execution. Thus, the second pass with an ordinary universal hash function (i.e., $\mathbf{H_w}$ in Section V) is avoided. Performing such an additional mappings is always possible at the cost of a few additional instruction execution and processor registers (and hence space-time bounds). For example, it is always possible to bring out-of-range numbers from $\{$p,$\ldots$, $2^w - 1\}$ into the $\{0,\ldots, p-1\}$ range by several alternate methods; e.g., see Figure 4 [49], or probabilistic methods.