What's Necessary to Establish Malware Freedom Unconditionally?

Virgil D. Gligor
ECE Department and CyLab
Carnegie Mellon University

***Abstract*—For two decades, attempts to detect malware in an untrusted system have relied on a trustworthy external verifier that challenged the system with the execution of special functions and measured whether the system's response was correct and timely. Recently, it was shown that such a verifier can provably and unconditionally establish *persistent-malware* freedom when the challenge functions are k-independent randomized polynomials. This provided a *sufficient* solution on a concrete Word Random Access Machine (cWRAM) – the closest machine model to real systems; e.g., commodity processors with large instruction sets. In this paper, we show what is *necessary* to establish malware freedom unconditionally on real systems. This is particularly relevant since gaps between what's sufficient on cWRAM and what's necessary on a real system may exist. Specifically, we answer the following open questions: Is a trustworthy *external verifier* and *challenge function* required to establish malware freedom? If yes, must the function's *optimal execution time in a given memory space* be measured instead of other parameters that are unavailable on cWRAM; e.g., current, voltage, frequency, energy, temperature? If so, must it have a *unique* optimal space-time bound for its code? And must it also be *target claw free*, as on cWRAM? If this is the case, why would the traditional claw-free functions, which have been successfully used in cryptography, be inadequate here? Finally, we argue that k-independent randomized polynomials are good choices for challenge functions.**

I. Introduction

Malicious software (malware) can be surreptitiously implanted into a system by an adversary or unwittingly imported by an unsuspecting user. It is *persistent* if it can survive in the firmware of device controllers, network interface cards, baseboard management controllers, for instance, despite repeated power cycles, secure- and trusted-boot operations [1]. In principle, anti-malware tools can detect and remove malware infecting application software, while security monitors, like security and separation kernels, can limit its damage. However, persistent malware is much harder to detect: an adversary can exploit any (e.g., supply chain) vulnerability that enables malware implants *under* the operating system where it can survive undetected for years. There it can bypass all anti-malware tools and security monitors, and communicate with remote controllers by detecting environment conditions (e.g., network identity, connection status) or exploiting OS vulnerabilities. Naïve attempts to remove it by re-flashing firmware do not work because malware can hide in areas that do not get updated or disingenuously respond with a prepackaged message; e.g., "update complete" or "already the latest version" [2]. Other than unpredictable connections to remote controllers, persistent malware has no properties [3] that can be recognized on sets of system (e.g., device) execution traces by an external observer [4].

Background. If persistent malware leaves no tell-tale sign on execution traces, how can an external observer determine either that there is or that there isn't malware in the system, without taking the system apart to analyze device firmware? To establish malware freedom, we describe a simple protocol that is executed at system boot whenever deemed necessary [1, 5].

Suppose that a small and simple verifier is locally connected to the untrusted system via a synchronous private channel. The untrusted system has a processor, which comprises both general purpose and special processor-state registers, and a memory. Processor-state registers and memory words represent the system state defined as an input vector v to a family of computations $\mathbf{C_{m,t}}(v)$ with execution space m and time t.

A simple protocol. The verifier asks the untrusted system to initialize v to chosen content that includes $\mathbf{C_{m,t}}$'s code. Then, as shown in Figure 1, it challenges the system to execute a function $\mathbf{C_{nonce}}$, which is selected by its random *nonce* from family $\mathbf{C_{m,t}}$, on input v. Suppose that $\mathbf{C_{m,t}}$'s code has a unique space-time optimal bound, m-t, and a strong collision-freedom property, which we call *target claw freedom within* m-t; i.e., for any $\mathbf{C_{nonce}}$ and v (any *target*), a function f and input y such that $(f, y) \neq (\mathbf{C_{nonce}}, v)$ and $f(y) = \mathbf{C_{nonce}}(v)$ (a *claw*) cannot be found within lower space-time bounds than m and/or t, except with low probability. If the system responds with the correct result $\mathbf{C_{nonce}}(v)$ in time t given execution space m, then the verifier concludes[1] that the system's state v contains *all and only* its chosen content; hence v is *malware free*. Otherwise, either malware execution or unaccounted content (e.g., malware hiding) is detected, or both.

The verifier's conclusion is intuitive. Any other function (e.g., malware) execution on that system would either exceed the $\mathbf{C_{nonce}}$'s execution time bound t and/or memory bound m on input v or return an incorrect result, or both. And since $\mathbf{C_{m,t}}$ is target claw-free, any malware modification of a verifier-chosen system state's v would yield a different result from $\mathbf{C_{nonce}}(v)$; i.e., the initialization of the processor state registers and/or memory contents is unverifiable. These outcomes have demonstrable high probabilities on the untrusted system. The verifier's conclusion holds whenever the untrusted system cannot surreptitiously communicate with a powerful remote system, obtain the correct result $\mathbf{C_{nonce}}(v)$, and return it to the verifier – all within time t; viz., Sections V and VII.

Adversary. Our adversary can exercise *all* attacks that implant persistent malware into the untrusted system, and then remotely control it. Malware can read/write the external

[1] The verifier obtains the correct result $\mathbf{C_{nonce}}(v)$ in execution time t and memory m by using a trusted computer, or equivalently a trusted simulator of the trusted computer, having the same configuration as the untrusted system.

verifier's local I/O channel, and modify system's initialization, software and firmware, but not its hardware. It can extract any software secret stored on the untrusted system, modify program code adaptively based on inputs, and execute any function on its chosen input. And it can communicate with its computationally *unbounded* remote controller. e.g., while trying to obtain and output result $\mathbf{C_{nonce}}(v)$ in less time than t. However, it lacks access to external verifier's source of true random numbers used for *nonces*. Although the unbounded remote controller can break *all* complexity-based cryptography, it cannot predict true random numbers or modify the system's hardware.

A sufficient solution on cWRAM. Prior work [1, 6] provided a sufficient solution to establish malware-free states on an untrusted concrete Word Random Access Machine (cWRAM) – the closest machine model to real systems; e.g., commodity processors with large instruction sets. The cWRAM is defined in Appendix A of [1] and briefly described below. The solution is *unconditional* in a general sense; i.e., without using secrets, trusted hardware modules/tokens, or bounds on the adversary's computing power. It follows the simple protocol outlined above: the verifier challenges the untrusted cWRAM to initialize itself, execute a special function, namely a k-independent randomized polynomial, and then measures whether the cWRAM's response is correct and timely, and detects malware execution or unaccounted content if not. k-independent randomized polynomials (reviewed in Section VII) have a single concrete m-t optimal bound on cWRAM and are target claw free within this bound. (For traditional claw-free functions [7, 8], see Section V.)

cWRAM. The cWRAM model is a concrete variant of Miltersen's *practical RAM* model [9] in the sense that it has a fixed word length and at most two operands per instruction. It has a small number of registers and a large number of memory words. The cWRAM extends the practical RAM with higher-complexity instructions, such as $multiplication$ and mod, and *I/O* instructions. It also has special processor-state registers (e.g., for interrupt and device status), and its instruction execution model accounts for interrupts. The cWRAM includes *all* known register-to-register, register-to-memory, and branching instructions of real instruction sets, as well as *all* integer, logic, and shift/rotate computation instructions. All operand addressing modes are supported, and all instructions execute in *unit time*. We adopted *function locality* – a complexity measure introduced for the practical RAM model by Miltersen [9] – to distinguish among cWRAM's computation instructions.

Concrete bounds and target claw freedom. The concrete m-t optimality[2] of k-independent randomized polynomials requires that we prove the concrete optimality of polynomial evaluation on cWRAM. However, such bounds have been known only for Horner's rule and only over infinite fields [10], and the gap between these bounds and the lower bounds over finite fields (e.g., $\mathbf{Z_p}$) is very large [11, 12]. Also, these bounds are limited to field operations (e.g., $+, \times$) and cannot hold in any system with large instruction sets like the cWRAM model and commodity processors. To prove the unique optimality Horner's rule on cWRAM we added a simultaneous space-time minimization condition and used *function locality* when needed to distinguish among the computation instructions. Then

[2] A program implementing a function is *concretely space-time optimal* on an instruction set architecture if *both* its space upper and lower bounds match *and* its time upper and lower bounds match, *non-asymptotically*.

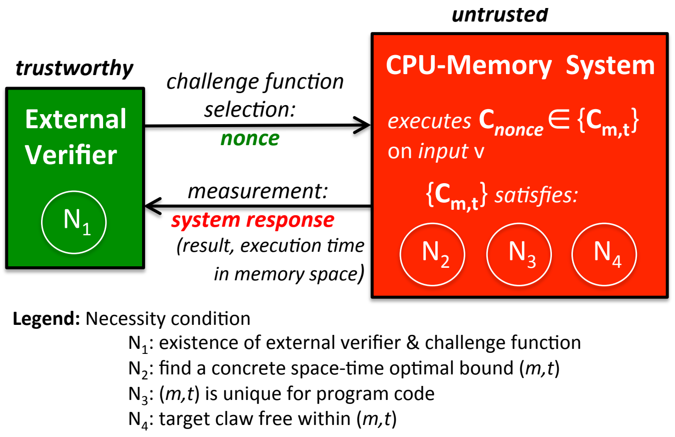

untrusted
trustworthy
External Verifier
N_1
challenge function selection:
nonce
measurement:
system response
(result, execution time in memory space)
CPU-Memory System
executes $C_{nonce} \in \{C_{m,t}\}$ on input v
$\{C_{m,t}\}$ satisfies:
N_2
N_3
N_4
Legend: Necessity condition
N_1: existence of external verifier & challenge function
N_2: find a concrete space-time optimal bound (m,t)
N_3: (m,t) is unique for program code
N_4: target claw free within (m,t)

Fig. 1. **Necessary conditions for malware freedom on untrusted systems**

we proved the m-t optimality of k-independent randomized polynomials in adversary evaluations on cWRAM, and showed that they have the collision property of target claw freedom within m-t; see Section IV-E(2) [1].

Contributions. In this paper, we show what is *necessary* to establish malware freedom unconditionally on a real system. Necessity conditions establish what is required for all solutions; e.g., $N_1 - N_4$ in Figure 1. They tell whether an existing solution is close to what's required, and if not, they identify gaps that suggest analysis, design, and implementation improvement. This is particularly relevant here since what's provably sufficient on cWRAM may not be required in real systems, thereby raising the question of the cWRAM solution's practicality. Specifically, necessity seeks answers to the following open questions.

First, are a *trustworthy external verifier* and *challenge function* required to prove that an untrusted system is (un)compromised by persistent malware? Can't the untrusted system run a protocol with other systems to establish malware freedom or detect malware execution/hiding, without an external verifier or challenge function? The necessity of external verifiers and challenge functions is shown in Section II.

Second, must the external verifier measure a challenge function's *optimal execution time* in a given *memory space* instead of other responses of a real system, which are unavailable on cWRAM? Couldn't it measure current, voltage, frequency, energy, and temperature, or emanations, such as electromagnetic or acoustic emanations [13, 14]? The necessity of optimal space-time measurements is demonstrated in Section III.

Third, must the challenge function have a *unique* optimal space-time bound for its program code? Or is this an artifact of the cWRAM's unit-time instruction execution? Clearly, a function's program code on a real system can have two such bounds: one using more time and less space and and the other more space and less time. The necessity of unique optimal code bounds is shown in Section IV and illustrated in Appendix A.

Fourth, must the challenge function be drawn from a *target claw-free* family like the k-independent randomized polynomials? If so, why would traditional claw-free functions, which have been successfully used in cryptography [7, 8], be inadequate here? The need for target claw free functions within optimal space-time bounds is argued in Section V. The use of these functions in *verifiable instruction execution* and untrusted-system initialization is discussed in Section VI.

This paper also shows why k-independent randomized polynomials are good choices of the new target claw-free functions on real systems (Section VII). Appendix B, Section X-D, shows how to measure their clock-cycle accurate performance on a typical commodity instruction set; i.e., Intel IA-32, IA-64.

II. Necessity of External Verifiers

Assume that a trustworthy external verifier and challenge function are unnecessary to establish persistent-malware freedom on an untrusted system. Since the system is untrusted on account of possible malware presence, it cannot prove it's malware free to any other system connected to it. Even if it's malware-free, an untrusted system cannot prove it because persistent malware has no known properties that can be denied. Hence, a protocol cannot exist between two such systems that establishes persistent-malware freedom of the untrusted one. This holds for n systems that run a protocol with an untrusted system to establish its persistent-malware freedom. (Since none of these systems are trustworthy external verifiers, they can be clustered into a single system that runs the protocol with the untrusted system.)

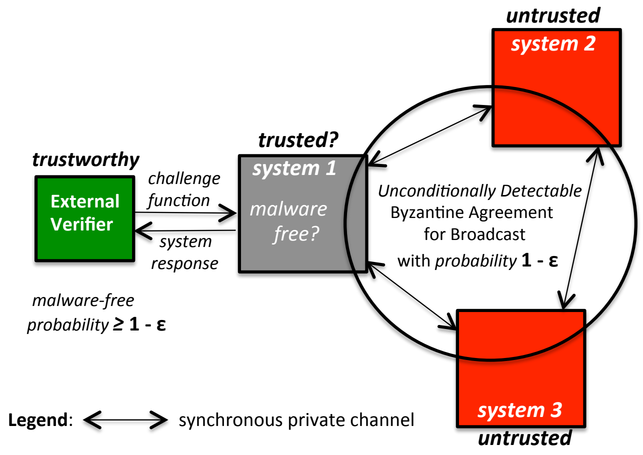
The necessity of trustworthy external verifiers and challenge functions generalizes to n untrusted systems ($n > 2$) that run a protocol to detect a *global* security property probabilistically and unconditionally in a computational sense. For example, the global property might be that all system states contain all and only content that excludes persistent malware. A protocol that detects the global property either terminates correctly if all n systems satisfy that property or aborts if at most $n-1$ systems do not. This is possible *only if* at least one of the n systems is trustworthy, and hence uncompromised by persistent malware, unconditionally and with at least the same probability as the detected security property. However, whether this trust condition holds requires independent verification. This reduces the two-system impossibility argument to this case, showing the need for external verifiers and challenge functions in global security-property detection.

For example, a distributed system can synchronously interconnect $n > 2$ component systems by pairwise channels that are unreadable by persistent malware; i.e., private channels. The distributed system can implement Byzantine agreements for *probabilistic detectable broadcast* [15–17] and *consensus* against a rational adversary [18] *unconditionally*, but only if at least one of its component systems is trustworthy[3]. A three-system example is illustrated in Figure 2. If *all* n systems are strategically manipulated by persistent malware, even *traditional consensus* becomes impossible when some systems crash [19]. However, that system's trustworthiness, and hence its persistent-malware freedom, has always been assumed but not independently verified for these protocols.

III. Necessity of Optimal Space-Time Bounds

In the simple protocol presented in the Introduction, the untrusted system's response is compared against the baseline measurement of optimal space and time obtained from the execution of $\mathbf{C_{nonce}}$'s program on a trusted system or trusted system simulator. However, $\mathbf{C_{nonce}}$'s program execution on

[3] A motivating example for *detectable broadcast* is global application-level malware detection running anti-malware tools on individual systems [17], Section 6.2. For a *detectable multi-party computation* at least $n/2$ components must be trustworthy [16].



trustworthy
External Verifier
challenge function
system response
malware-free probability ≥ 1 - ε
trusted?
system 1
malware free?
Unconditionally Detectable Byzantine Agreement for Broadcast with probability 1 - ε
untrusted
system 2
system 3
untrusted
Legend: ⟷ synchronous private channel

Fig. 2. **A Verified Unconditionally Detectable Byzantine Agreement (n=3)**

a real system could also be captured by measuring other parameters such as current, voltage, frequency, temperature, and clock cycles per instruction that are unavailable on cWRAM. Now the verifier would need the *baseline measurement* of the minimum amount of resources used by $\mathbf{C_{nonce}}$'s program to prevent malware execution or hiding in system state v, and the *correct result*, $\mathbf{C_{nonce}}(v)$. Note that these new baseline measurements would always be performed on a *trusted system*, or equivalently a *trusted simulator*, having the same configuration as the untrusted system.

As in the simple protocol of the Introduction, a correct untrusted system's response implies that malware could not bypass the baseline measurement limits by modifying $\mathbf{C_{nonce}}$'s program or input *and* return the correct result to the verifier. Hence, upon receiving a correct result, the verifier would measure any difference between the actual resources used by $\mathbf{C_{nonce}}$'s program on the untrusted system and the baseline amount of resources needed to establish malware freedom or detect malware presence. Differences arise because malware must either execute instructions or hide in memory, thereby exceeding the baseline resources. An incorrect result would also signal malware presence.

In the balance of this section, we show that the baseline measurements of resource use by $\mathbf{C_{nonce}}$'s program imply energy minimization. We also show that energy minimization implies optimal space-time bounds for a specific choice of $\mathbf{C_{nonce}}$'s programs and system initialization needed to improve measurement accuracy. However, optimal space-time bounds need not minimize the program's energy consumption. Hence, an external verifier only needs to measure the optimal space-time bounds of $\mathbf{C_{nonce}}$'s program execution.

A. *Baseline measurements imply energy minimization*

Instruction granularity. Detecting differences between the actual resource use by malware and baseline measurements requires that measurements are performed at the instruction execution granularity. Injecting a single jump instruction can modify the control flow of $\mathbf{C_{nonce}}$'s program execution and allow return-oriented programming attacks [20]. Even if such attacks are prevented [21], bypassing a single instruction execution during $\mathbf{C_{nonce}}$'s program initialization would enable malware survival. For example, if a *disable_interrupts* instruction is not executed when $\mathbf{C_{nonce}}$ is initialized, a future-posted interrupt could be programmed to trigger and reboot a malware-contaminated kernel after the verifier's measurement ends [5].

Similarly, if the execution of a *clear_I-cache* instruction cannot be detected in systems that do not synchronize instruction and data caches (e.g., ARM Cortex-A8), malware could hide in the I-cache during the $\mathbf{C_{nonce}}$'s non-contaminated execution.

Internal sensors. Baseline measurements are made by (possibly intrusive) internal sensors, which can be reliably read by the external verifier. External sensors, which can detect fluctuations of electric potential, electromagnetic fields, and acoustic vibrations [13, 14], are unnecessary since all emissions are generated by electronic components whose state is captured by the internal sensors.

Since baseline measurements are always done on a trusted (malware-free) system, the verifier need not be concerned with sensor manipulation by malware. However, some sensor readings cannot be used in baseline measurements either because they are inaccurate or because they are constant during program execution. Temperature readings are inaccurate because fan throttling and ambient-temperature bias them [22] and because sensors are often located away from the CPU. Fortunately, temperature effects can be removed from other sensor readings to increase their accuracy; e.g., power measurements [23]. Voltage and frequency, which are linearly related [24], are often constant. When they aren't, they can be kept constant value (e.g., by disabling dynamic voltage and frequency scaling) since their fluctuation only decreases the accuracy of other sensor readings.

Energy minimization. Baseline measurements require the entire execution of $\mathbf{C_{nonce}}$ programs on their inputs, v. Thus, the readings of the remaining sensors, namely the power level (e.g., current at constant voltage) and running time (e.g., CPU clock cycles at constant frequency), yield $\mathbf{C_{nonce}}$'s energy consumption. Measurement accuracy is assured by verifier's choice of $\mathbf{C_{nonce}}$ programs and system initialization, as shown in the next section. Since the baseline measurements must show the minimum use of resources by a $\mathbf{C_{nonce}}$ program and input that prevent malware execution or hiding, they require minimum energy use.

B. Minimum energy implies optimal space-time bounds

The energy $E_{sys,i}$ used during the execution of instruction i by an *application program* initialized in system memory is measured by the power, $P_{sys,i}$, used during the execution time, Δt_i, of that instruction; i.e., $E_{sys,i} = P_{sys,i} \times \Delta t_i$, where $P_{sys,i}$ is constant during Δt_i. We expand $P_{sys,i}$ and Δt_i as in De Vogeleer *et al.*'s model [24]:

$$P_{sys,i} = (P_{cpu,i} + P_{drop,i} + P_{back}),$$

where

- $P_{cpu,i}$ is the power used by instruction i on a single-core CPU;

- $P_{drop,i}$ is the power of auxiliary devices during the execution of instruction i. It includes GPUs, radio interfaces, camera circuits, *etc*.

- P_{back} is the background system power, which is independent of the CPU operation and includes power components such as stand-by and refresh primary memory power, AC/DC conversion power, chipset and fan power. Also,

$$\Delta t_i = cc_i \cdot (\tfrac{1}{f-f_k} + \beta)$$

where

- cc_i is the clock cycle count for instruction i;

- f is the clock frequency of the CPU core;

- f_k is the average number of clock cycles per time unit lost to OS services (e.g., page faults, interrupt handling), pipeline stalls due to branch miss-prediction during program execution, and other "time thieves."

- β is the average amount of stall time per clock cycle caused by primary-memory references made by the CPU core while executing a program.

Using $P_{sys,i}$ and Δt_i,
$E_{sys,i} = (P_{cpu,i} + P_{drop,i} + P_{back}) \cdot cc_i \cdot (\frac{1}{f-f_k} + \beta)$.

The accuracy of $E_{sys,i}$ in baseline measurements is substantially improved if $P_{drop,i} = 0$, $\beta \to 0$ [24], and $f_k = 0$ by *low-level system-program* measurements; e.g., eliminating OS services, disabling CPU features, and simplifying program behavior.

Measurement accuracy. To perform accurate baseline energy measurements, the verifier must initialize the malware-free system and select a special $\mathbf{C_{nonce}}$ program in a way that minimizes energy variation.

System initialization. At initialization, the verifier performs following three actions. First, it runs the $\mathbf{C_{nonce}}$ program at boot time, before the OS and applications are loaded. Since single-core energy measurements are simpler and more accurate than multi-core ones [25], the boot operation is on a single core; i.e., all other cores are explicitly deactivated. Second, the verifier sets clock frequency f, and implicitly the voltage, to a constant value (whose choice is discussed in Section VI), and powers down auxiliary devices thereby making $P_{drop,i} = 0$. Third, it disables architecture features that would otherwise increase power-measurement variation; e.g., it disables virtual memory, TLBs, caches, and interrupts. These actions and the program choices discussed below make $f_k = 0$. The verifier performs the power measurements at a constant temperature (e.g., 37°C [24]), and then removes any residual temperature fluctuation from these measurements using established techniques [23].

Choice of programs. The verifier also selects a specific $\mathbf{C_{nonce}}$ program that further reduces the variation of baseline energy measurements. For example, to make $\beta \to 0$ in $E_{sys,i}$, the verifier chooses a $\mathbf{C_{nonce}}$ program whose instructions are predominantly CPU-register bound and each memory word is accessed only once. Thus, β becomes a small positive constant ϵ. Also, to eliminate branch-prediction uncertainty, the chosen $\mathbf{C_{nonce}}$ program maintains predictable loops (e.g., constant number of branch-backs) and removes all other program branches. Furthermore, the verifier chooses a $\mathbf{C_{nonce}}$ program whose instruction sequences are latency bound; i.e., the result of one instruction depends on that of previous instructions. This removes the performance advantage of superscalar execution, which would otherwise add significant energy variation [26]. It also removes the usefulness of pipelining since the execution of latency-bound instructions cannot be overlapped. Finally, the verifier selects an integer-based $\mathbf{C_{nonce}}$ program, as this further simplifies energy measurements. For example, $P_{cpu,i}$ and cc_i increase/decrease at the same time for integer instructions since their power/energy use relative to other integer instructions follows the same ordering relationship as the instruction latencies [27, 28].

As a consequence of the verifier's system initialization actions and $\mathbf{C_{nonce}}$'s program selection the energy used for instruction i becomes:

$$E_{sys,i} = (P_{cpu,i} + P_{back}) \cdot cc_i \cdot (\tfrac{1}{f} + \epsilon).$$

Here f and ϵ are constants, and P_{back} is independent of both $P_{cpu,i}$ and cc_i, since it's independent of the CPU operation. However, P_{back} depends on the primary memory size as it powers the entire memory.

Energy minimization implies time and space minimization. Recall that the baseline measurements of $\mathbf{C_{nonce}}$'s execution on input v show the *minimum amount of resources* needed to prevent malware from running or hiding on a system. This implies that, for an n-instruction $\mathbf{C_{nonce}}$ program, $E_{sys} = \sum_{i=1}^{n} E_{sys,i}$ must be minimized. However, E_{sys} is minimized *only if* the program execution time $\sum_{i=1}^{n} cc_i \cdot (\frac{1}{f} + \epsilon)$ is minimized[4], since $P_{cpu,i}$ and cc_i decrease together, and $\frac{1}{f} + \epsilon$ is constant.

E_{sys}'s minimization also implies that the memory space of $\mathbf{C_{nonce}}$'s program code and input is minimized. Let the memory space of the program code and input fill the entire primary memory, and assume by contradiction that E_{sys} is minimized but the memory space is not. If E_{sys} is minimized, P_{back} is minimized, which means that the primary memory size is minimized, by P_{back}'s definition. However, if $\mathbf{C_{nonce}}$'s memory space is not minimized, it could be compressed at initialization without increasing the minimum E_{sys} at run time; e.g., without requiring access to secondary storage or to an external system, or by decompressing code or input data. This means that the size of the primary memory is not filled and hence not minimized, which contradicts the assumption made.

Optimal space-time bounds are necessary. In short, baseline energy measurements imply that *both* the execution time *and* memory space are minimized for the verifier-chosen $\mathbf{C_{nonce}}$ program and input given the low-level system initialization. If the minimized memory space and execution time are *not* the *lower* space-time bounds of the $\mathbf{C_{nonce}}$ program, then after a space-time optimized program completes execution malware could still execute instructions or hide in memory before before it must return the correct result to the verifier. This would violate the definition of baseline energy measurements, and hence the *lower* space-time bounds must be reached. Thus, an external verifier need only find $\mathbf{C_{nonce}}$ program's *lower* space-time bounds on the given input when running a malware-free system. Since $\mathbf{C_{nonce}}$'s *upper* space-time bounds are given by its program execution and input on that system, the *lower* bounds the verifier obtains are *optimal*.

Space-time optimality need not minimize energy. The fact that space-time optimal program need not optimize its energy consumption is of independent interest as it refines earlier experimental observations [26, 29]. Two examples illustrate. First, De Vogeleer *et al.* [24] derive a general energy/frequency convexity rule for single-core processors, and show that the *minimum* energy consumption of a program and input requires *specific frequency values*; viz., Appendix B, Section X-B. In contrast, baseline energy measurements for a program and input that prevent malware execution or hiding need only show *minimum* energy consumption at a frequency value whose choice may differ from the optimal. Second, a program can have different space-time optimal bounds that depend only on instructions' latencies (cc_i), and hence on $P_{cpu,i}$, and are independent of frequency settings – as illustrated in the next section. This means that different frequencies can produce different optimal time bounds but only one yields minimum energy [24].

[4] This implication has already been noted by several other researchers for the past two decades [26, 29]. This is also pointed out by De Vogeleer *et al.* [24], where cc is meant to represent the computation's *code size*.

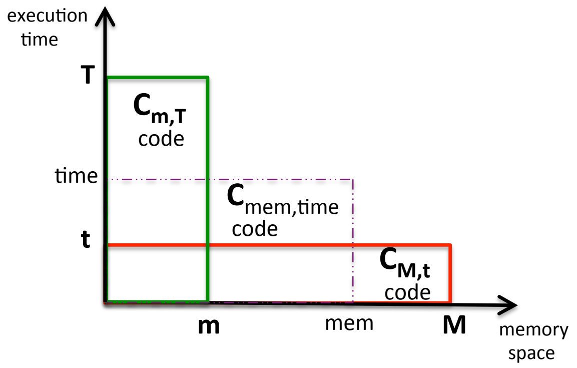


Fig. 3. **Multiple optimal space-time bounds of program family $\mathbf{C_{m,t}}$**

IV. Necessity of Enforcing a Unique Optimal Bound

Multiple optimal bounds. The space-time optimality of $\mathbf{C_{nonce}}$'s code in cWRAM means that the family of functions from which it is drawn uniformly at random by the *nonce*, $\mathbf{C_{m,t}} = \{\mathbf{C_{nonce_1}} \ldots \mathbf{C_{nonce_n}}\}$, is space-time ($m$-$t$) optimal. However, on a real-system instruction set there may exist two, or more, program families corresponding to cWRAM's $\mathbf{C_{m,t}}$ with different optimal space-time bounds, each using a different program code while producing the same evaluation result, $\mathbf{C_{nonce}}(v)$, for same input v. For example, both program codes of families $\mathbf{C_{m,T}}$ and $\mathbf{C_{M,t}}$, where $m < M$ and $T > t$, can be optimal and return the same results for same inputs on a real-system instruction set. There may also exist another space-time optimal program family, $\mathbf{C_{mem,time}}$, where $m < mem < M$ and $T > time > t$ (Figure 3) which also uses different instructions, as illustrated below.

Let the optimal program codes of families $\mathbf{C_{m,T}}$ and $\mathbf{C_{M,t}}$ be the unsigned-integer implementations of the Horner-rule step computation in $\mathbf{Z_p}$, where $\mathbf{p}$ is the largest prime that fits in a memory word. In Appendix A, Section IX, we show that a space-time optimal program of $\mathbf{C_{m,T}}$ can be implemented with the mod (*aka., integer division with remainder*) instruction, which uses less memory (i.e., fewer instructions) and more time than a space-time optimal program of $\mathbf{C_{M,t}}$, which is implemented with only *integer multiplications* and uses more memory and much less time. This is the case in all real-system instruction sets that implement the mod instruction since its latency is far higher than *integer multiplication* instructions. Another optimal program family, $\mathbf{C_{mem,time}}$, could be implemented using ordinary *integer division* – which is always faster than mod – and *integer multiplication*. Different space-time optimal bounds corresponding to three different instruction codes of the same program family are shown in Figure 3. This is impossible in cWRAM since all instructions have unit-time latencies.

Single optimal bound. In the following example we show why a single optimal space-time bound is necessary. That is, malware could surreptitiously substitute the code of $\mathbf{C_{M,t}}$ for the code of $\mathbf{C_{m,T}}$, during untrusted-system initialization requested by the verifier and survive in system memory.

Example. Let the memory of an untrusted system have S bytes, which allows the initialization of $\mathbf{C_{m,T}}$ and input string of u bytes but not that of $\mathbf{C_{M,t}}$ and u; i.e., $S = m+u < M+u$. Let the external verifier request the initialization of $\mathbf{C_{m,T}}$ code in the untrusted system's memory. Instead, the persistent malware initializes $\mathbf{C_{M,t}}$'s code and $S-M$ bytes of input u that fit in S. Then, while executing $\mathbf{C_{M,t}}$, malware transfers the rest of $M-m$ bytes of input u, which is a small number of bytes, from secondary storage into system memory in time $\delta \cdot t$, $(0 < \delta < 1)$. $\mathbf{C_{M,t}}$ processes its entire input u and responds to the verifier in the remaining time $T-(1+\delta)\cdot t > 0$. Thus, malware survives undetected in untrusted-system memory since the verifier expects the response in time T as it assumes the initialization of the $\mathbf{C_{m,T}}$ code. Note that condition $T/t > (1+\delta)$ is easily met on real-system instruction sets. For instance, $T/t > 3$ in an early x86-32 implementation of the Horner-rule step using the *mod* instruction, which takes 12.4 clock cycles per byte, whereas the implementation using only the *integer multiplications* takes only 3.69 clock cycles per byte [30]. This ratio is fairly typical for modern processors where the difference between the *mod* and *integer multiplication* latencies is very large [31, 32].

Second pre-image freedom. At a first glance, the above example seems to suggest that the verifier can choose *any* program code of $\mathbf{C_{nonce}} \in \mathbf{C_{M,t}}$ and input u, which fill the system's memory S, as the challenge function. However, this is not the case, as these programs cannot prevent malware from finding a pre-image $u' \neq u$, such that $\mathbf{C_{nonce}}(u') = \mathbf{C_{nonce}}(u)$ and $\mathbf{C_{nonce}}(u')$'s program executes in time $t' < t$. This would enable malware to execute instructions in time $t - t'$ undetected. To prevent this and ensure that $\mathbf{C_{M,t}}$'s code retains the single optimal bound requires that $\mathbf{C_{M,t}}$ must also be *second pre-image free within* (M,t), except with low probability.

Code identity. A stronger property than second pre-image freedom, namely *code-identity within optimal bounds* (m,t), is required whenever malware can optimize program encoding adaptively based on received input values[5], as assumed by the adversary model. That is, a unique result $\mathbf{C_{nonce}}(v)$ within the (m,t) bounds is necessary for each distinct program encoding, except with low probability [1]. This is easily achieved by including the $\mathbf{C_{m,t}}$'s program code into the input string v.

V. Necessity of Target Claw Freedom within Optimal Bounds

Obtaining code identity within a unique optimal bound (m,t) for program family $\mathbf{C_{m,t}} = \{\mathbf{C_{nonce_1}}, \dots, \mathbf{C_{nonce_n}}\}$ is not strong enough a condition when facing adaptive malware. An adversary's malware can find and execute an arbitrary function $f \notin \mathbf{C_{m,t}}$ and input y such that $f(y) = \mathbf{C_{nonce_i}}(v)$ with lower bounds $(m',t') \angle (m,t)$ [1]. (Here $(m',t') \angle (m,t)$ denotes $t' < t, m' = m$ or $t' = t, m' < m$ or $t' < t, m' < m$.) Also, there may exist another function $f = \mathbf{C_{nonce_j}} \in \mathbf{C_{m,t}}$ and input $y = v'$ such that $(\mathbf{C_{nonce_j}}, v') \neq (\mathbf{C_{nonce_i}}, v)$ and $\mathbf{C_{nonce_j}}(v') = \mathbf{C_{nonce_i}}(v)$ with bounds (m',t'). We say that the adversary's choice of (f,y) forms a *claw* for a *target* $\mathbf{C_{nonce}}(v)$ within the unique optimal bound (m,t). However, if no adversary can find such a claw except with low probability,

[5] For example, small inputs could be encoded into immediate address fields of instructions instead of separate memory words to save execution space.

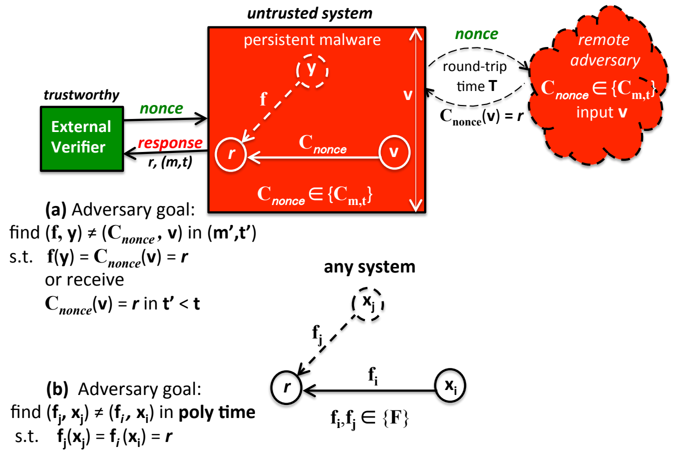


Fig. 4. **Target claw freedom in optimal (m,t) bounds (a) and target claw resistance in polynomial time (b)**

we say that family $\mathbf{C_{m,t}}$ is *target claw free within unique space-time optimal bound* (m,t). Note that the functions $\mathbf{C_{nonce}} \in \mathbf{C_{m,t}}$ are executed only once and *no speedup* of execution on input v is ever possible.

To ensure that family $\mathbf{C_{m,t}}$ is target claw free *unconditionally*, an external verifier forces a computationally unbounded adversary to perform two-part attacks in which the adversary provably fails. First, the adversary's persistent malware must find a claw $(f,y) = C_{nonce}(v)$ and provably fails to do so since their power is limited by a unique optimal bound (m,t) on the given system's instruction-set architecture. Second, since the remote adversary's programs have unbounded power, they can compute correct result $C_{nonce}(v)$ in zero time and return to the persistent malware. To do this, they would have to communicate *undetectably* with the persistent malware programs within round-trip time T and provably fail to do so again.

Early attempts to detect surreptitious wireless communication between the persistent malware and remote adversary programs suggested use of radio-frequency analyzers [5]. Prevention of surreptitious communication is possible in environments where pervasive wireless communication cannot be denied. For example, the external verifier can disable all system communication with any remote system (e.g., running adversary programs) in a verifiable manner (see the *verifiable instructions* in Section VI) until after the result of the one-time evaluated target function $\mathbf{C_{nonce}}(v)$ is checked. In addition, it verifiably limits the optimal execution time, t, of $\mathbf{C_{nonce}}(v)$ to a smaller value than the round-trip time, T, to any remote adversary on the fastest channel. This is illustrated in Figure 4(a). In practice, this may require the random selection, sequential execution, and verification of multiple functions $\mathbf{C_{nonce_1}}(v_1) \dots \mathbf{C_{nonce_n}}(v_n)$, each represented in a memory segment v_s where $v = \sum_{s=1}^{n} v_s$ [1]. Persistent malware attempts to communicate with the remote adversary programs are thus rendered useless.

Differences from traditional claw-free functions. Claw-free permutations and functions have been traditionally used in cryptography for a variety of provably secure constructs; e.g., protection against existential signature forgeries [7], collision resistant hash functions, commitment protocols [8]. Informally, if $F = \{f_1, \dots, f_n\}$ is a finite family of functions or permutations, F is said to be a *claw-free* family if any probabilistic polynomial time (PPT) adversary cannot find a tuple (i, j, x, y) such that $f_i(x) = f_j(y)$. If the adversary is challenged with a *target* tuple (i, x), and hence with $f_i(x)$, then F is said to be a *target claw-free* family, and $f_j(y)$ is the claw for target $f_i(x)$. This is illustrated in Figure 4(b).

The new family of target claw-free functions $\mathbf{C_{m,t}}$ is different from these in three ways. First, function f of claw (f, y) need not necessarily be a function $\mathbf{C_{nonce}}$ of family $\mathbf{C_{m,t}}$ nor input y be a family input v. Instead, f can be an arbitrary function that need only have some input y such that $f(y) = \mathbf{C_{nonce}}(v)$. Second, the computing power of the adversary who attempts to find claw (f, y) is not restricted that of a PPT adversary. Instead, the adversary power is unbounded but verifiably divided between that of persistent malware programs and remote adversary programs, and provably countered by the external verifier; i.e., by detecting or preventing communication between these two types of adversary programs within useful bounds. Third, family $\mathbf{C_{m,t}}$ need not rely on hardness conjectures nor protect secrets; e.g., using trusted hardware modules/tokens. Instead, it relies on its provable optimality of the unique space-time bound on a given system's instruction set. Nevertheless, traditional claw-free functions have PPT adversary bounds in *any* system, and eliminate the need for external verifiers and concrete space-time optimality results on specific systems' instruction sets. These differences are succinctly illustrated in Figure 4.

VI. Verifiable instruction execution

Recall that an external verifier is guaranteed correct system initialization and execution of optimal program code since baseline measurements are made on a trusted (i.e., malware-free) system or simulator, by definition. System initialization executes instructions that strips down processor features and sets processor configuration registers to ensure that measurements are deterministic and accurate. For example, it disables interrupts/asynchronous events, caches, TLBs, extra CPU cores [26, 33], auxiliary devices (e.g., GPUs, radio interfaces, camera circuits), and communication with remote systems. It also disables hyper-threading, dynamic voltage-frequency scaling, and turbo-boost modes [34], and sets clock frequency.

In contrast, verifier's measurements of the untrusted system must either detect that initialization is carried out and optimal program code is executed completely and correctly or signal malware presence, with high probability. For example, these measurements must ensure that the optimal program code is not modified by malware to surreptitiously increase CPU frequency *during* code execution and undetectably bypass the optimal time bound obtained in baseline measurements at a lower frequency. The frequency-setting instructions executed on the untrusted system become verifiable whenever the verifier performs the baseline measurements at the highest admissible frequency value. This ensures that an untrusted system malware cannot surreptitiously over-clock the optimal program execution and bypass its time bounds[6] in verifier's measurements [5].

Stripping down and setting processor features during untrusted-system initialization explicitly set processor-state registers; i.e., they save the enabled/disabled status and values in processor state words of input v. Since $\mathbf{C_{nonce}}$ code is target claw free within the unique optimal bound (m, t), once the code starts running an enabled/disabled feature's status and value can no longer be undetectably changed in the system-state input v by malware until the untrusted system measurement ends. That is, an initialization instruction that sets processor state words becomes *verifiable* if both its encoding in memory and the content of the words it modified become part of the input v of $\mathbf{C_{nonce}}$ [1]. Thus, both the instruction encoding in memory and its effect is in the processor state words are captured by the correct result $\mathbf{C_{nonce}}(v)$ and time bound t, which are verifiable.

Note that some instruction executions can become verifiable even if they do not (re)set a processor-state field. These instructions are placed between two verifiable instructions in straight-line code. Hence, their execution is captured by the external verifier's measurement.

VII. k-randomized polynomials as target claw-free functions

Let p be the largest prime that fits into a w-bit word and (r_j, x), $j = 0, \ldots, k-1$, be a *nonce*, where each r_j and x are drawn uniformly at random from $\mathbf{Z_p}$. Let $v = v_d, \ldots, v_0$, $v_i \in \mathbf{Z_p}$, be a string of constants each of which is stored in a w-bit word. A k-independent randomized polynomial of degree d is selected from family $\mathbf{H}$ by index $(d, nonce)$, which is denoted as (d, k, x) below; i.e.,

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

where $v_i \oplus s_i$ is represented by a $mod\ p$ integer [1]. The k-independent randomized polynomials have uniformly distributed output and the probability of finding a claw function of at most $3/p$ within optimal space-time bounds $\mathbf{m} = k + 22$ words and $\mathbf{t} = (6k-4)6d$ time units after initialization of $k+8$ general-purpose registers (GPRs) on cWRAM. If a new family of target claw-free functions $\mathbf{C_{m,t}}$ with uniformly distributed output over $\mathbf{Z_p}$ has a $1/p$ probability of finding a function claw, the k-independent randomized polynomials are good claw-free functions as long as they retain a unique optimal space-time bound for their program encoding on a real system.

As suggested in Section V, an external verifier can limit execution time $\mathbf{t}$ of a k-independent randomized polynomial on an untrusted system to prevent persistent-malware communication with a remote adversary program. That is, $\mathbf{t}$ must be lower than the round trip time, $\mathbf{T}$, to the remote adversary program on the fastest communication channel; viz., Figure 4(a). However, for a k-randomized polynomial, $\mathbf{t}$ depends on both its degree d and k. Although d, and hence the size of each segment v_s, can be easily minimized in random sequential evaluations [1], k grows with the number of GPRs available on a CPU core.

The larger the k, the longer the evaluation takes since the latency of coefficient generation increases with k. Hence, the larger the k, the smaller the v_s size must be to assure that the execution time is small enough and to detect any attempt to communicate with remote adversary programs. This is easily achieved in practice, because on all modern processors the value of k is small. For instance, on typical ARM processors with sixteen GPRs, only five registers can be used to hold random numbers, and thus k is at most five, whereas on

[6] In systems that allow granular frequency settings (e.g., bus-cycle multipliers) it may only be necessary to set the frequency to a slightly lower value than maximum if it can be shown that over-clocking cannot offset the time lost by malware manipulation.

MIPS processors with thirty-two GPRs, k would be at most twenty-one. In both cases at least three GPRs hold processor-required values. This allows a local verifier to prevent malware communication with a remote adversary program on typical commodity-system channels, even if the polynomials have a significant number of coefficients; e.g., over 500 coefficients and corresponding sizes of each segment v_s.

VIII. Conclusions

In this paper we show that small and simple external verifiers, whose trustworthiness can be easily proved, are necessary for detecting the persistent-malware-freedom of untrusted systems using carefully designed challenge functions; i.e., target claw free functions in optimal space-time bounds. These verifiers must measure the challenge functions' execution time in a given memory space and check the correct result, at system boot time.

The usefulness of establishing malware freedom on untrusted systems is illustrated by its relationship with software root-of-trust (RoT) establishment, secure state, and verifiable boot [1, 6]. A RoT state comprises *all* and *only* content chosen by the external verifier, and the verifier's code begins execution in that state. A state is secure if it satisfies a security predicate. Verifiable boot means that either a program is booted in a secure state or the boot fails. All these notions require persistent-malware freedom, as follows:
verifiable boot $\rightarrow$ *secure state* $\rightarrow$ *RoT state* $\rightarrow$ *malware-free state*,
where A $\rightarrow$ B means A *requires* B but B *does not require* A.

It's worth remembering that all formal access control models ever proposed require the notion of secure initial state, and all practical systems also require recovery and restart in a secure state after program abort or failure.

Acknowledgment

This paper benefited from insightful comments and valuable suggestions received from Bryan Parno, Raluca Popa, Vyas Sekar, Elaine Shi, and Miao Yu.

References

[1] V. D. Gligor and M. Woo, "Establishing Software Root of Trust Unconditionally," in *Proc. of the 2019 NDSS, San Diego, CA*. ISOC, 2019 (full paper in CMU - CyLab, Technical Report 18-003, Nov. 2018).

[2] C. Raiu, "Commentary in Equation: The Death Star of the Malware Galaxy," in *Kaspersky Lab*, Feb 19, 2015. [Online]. Available: https://securelist.com/equation-the-death-star-of-malware-galaxy/68750/

[3] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010. [Online]. Available: http://dx.doi.org/10.3233/JCS-2009-0393

[4] L. Mearian, "There's no way of knowing if the NSA's spyware is on your hard drive," *Computerworld*, vol. 2, 2015.

[5] Y. Li, Y. Cheng, V. Gligor, and A. Perrig, "Establishing software-only root of trust on embedded systems: facts and fiction," in *Proc. of the Security Protocols Workshop*, ser. LNCS, vol. 9379. Springer, 2015, pp. 50–68.

[6] V. D. Gligor, "A rest stop on the unending road to provable security (article and transcript of discussion)," in *Proc. of the Security Protocols Workshop, Cambridge, UK*, 2019.

[7] S. Goldwasser, S. Micali, and R. Rivest, "A paradoxical solution to the signature problem," in *In Proc. of the 25th IEEE Symp. on Foundations of Computer Science*, 1984.

[8] I. Damgard, "The application of claw free functions in cryptography: unconditional protection in cryptographic protocols," in *PhD Thesis, Aarhus University*, 1988.

[9] P. B. Miltersen, "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)*. Springer, 1996, pp. 442–453.

[10] A. Borodin, "Horner's rule is uniquely optimal," in *Proc. of the International Symposium on the Theory of Machines and Computations*, Z. Kohavi and A. Paz, Eds. Elsevier Inc, 1971, pp. 47–57.

[11] M. Elia, J. Rosenthal, and D. Schipani, "Polynomial evaluation over finite fields: new algorithms and complexity bounds," *Applicable Algebra in Engineering, Communication and Computing*, vol. 23, no. 3, pp. 129–141, Nov 2012.

[12] N. Kayal and R. Saptharishi, "A selection of lower bounds for arithmetic circuits," in *Perspectives in Computational Complexity – Progress in Computer Science and Applied Logic*, M. Agrawal and V. Arvind, Eds. International Publishing Switzerland, 2014, pp. 77–115.

[13] D. Genkin, L. Pachmanov, I. Pipman, A. Shamir, and E. Tromer, "Physical key extraction attacks on PCs," *Commun. ACM*, vol. 59, no. 6, pp. 70–79, 2016.

[14] D. Genkin, A. Shamir, and E. Tromer, "Acoustic cryptanalysis," *Journal of Cryptology*, pp. 392–443, 2017.

[15] M. Fitzi, D. Gottesman, M. Hirt, T. Holenstein, and A. Smith, "Detectable byzantine agreement secure against faulty majorities," in *In Proceedings of the 21st PODC*, 2002, pp. 118–126.

[16] M. Fitzi, M. Hirt, T. Holenstein, and J. Wullschleger, "Two-threshold broadcast and detectable multi-party computation," in *Advances in Cryptology — EUROCRYPT 2003*, E. Biham, Ed., 2003, pp. 51–67.

[17] M. Fitzi, "Generalized communication and security models in Byzantine Agreement," in *PhD Thesis, ETH Zurich*, 2003.

[18] A. Groce, J. Katz, A. Thiruvengadam, and V. Zikas, "Byzantine agreement with a rational adversary," in *Proc. of Int'l. Col. on Automata, Languages, and Programming*, A. Czumaj, K. Mehlhorn, A. Pitts, and R. Wattenhofer, Eds. Springer, LNCS 7392, 2012, pp. 561–572.

[19] X. Bei, W. Chen, and J. Zhang, "Distributed consensus resilient to both crash failures and strategic manipulations," in *CoRR, arXiv*, 2012.

[20] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *Proc. of ACM CCS*, 2009, pp. 400–409.

[21] A. Perrig and L. van Doorn, "Refutation of "On the Difficulty of Software-Based Attestation of Embedded Devices"," pp. 1–6, 2010. [Online]. Available: https://sparrow.ece.cmu.edu/group/pub/perrig-vandoorn-refutation.pdf

[22] M. Bach, "How ambient temperatures affect your PC," in *Puget Systems, Technical Report, August 15*, 2012. [Online]. Available: https://www.pugetsystems.com

[23] K. De Vogeleer, G. Memmi, P. Jouvelot, and F. Coelho, "Modeling the temperature bias of power consumption

for nanometer-scale CPUs in application processors," in *Proc. of International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, 2014, pp. 172–180.

[24] K. De Vogeleer, G. Memmi, and P. Jouvelot, "Parameter sensitivity analysis of the energy/frequency convexity rule for nanometer-scale application processors," *Sustainable Computing: Informatics and Systems*, vol. 15, 2017.

[25] G. Hager, J. Treibig, J. Habich, and G. Wellein, "Exploring performance and power properties of modern multi-core chips via simple machine models," *Concurrency and Computation:Practice and Experience*, pp. 189–210, 2013.

[26] J. Pallister, S. Hollis, and J. Bennett, "The impact of different compiler options on energy consumption," in *Proc. of First LPGPU Workshop on Power-Efficient GPU and Many-core Computing, New York*, 2013.

[27] D. Molka, D. Hackenberg, R. Schoene, and M. Mueller, "Characterizing the energy consumption of data transfers and arithmetic operations on x86-64 processors," in *Proc. of First International Conference on Green Computing, Chicago*, 2010, pp. 123–133.

[28] E. Vasilakis, "An instruction level energy characterizationof ARM processors," in *Technical Report FORTH-ICS/TR-450, University of Crete*, 2015.

[29] T. Yuki and V. S. Rajopadhye, "Folklore confirmed: Compiling for speed = compiling for energy," in *Proc. of Languages and Compilers for Parallel Computing, C. Cascaval and P. Montesinos (eds)*. LNCS vol. 8664, Springer, 2013, pp. 169–184.

[30] T. Krovetz and P. Rogaway, "Fast universal hashing with small keys and no preprocessing: The polyr construction," in *Information Security and Cryptology (ICISC)*. Springer Berlin Heidelberg, 2001, pp. 73–89.

[31] A. Fog, "Instruction tables," 2018. [Online]. Available: https://www.agner.org/optimize/instruction_tables.pdf

[32] T. Granlund, "Instruction latencies and throughput for AMD and Intel x86 processors," 2017.

[33] J. Pelner and J. Pelner, "Minimal bootloader for Intel architecture," in *Intel Corporation*, 2010.

[34] G. Paolini, "How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures," in *Intel Corporation*, 2010.

[35] L. Carter and M. Wegman, "Universal classes of hash functions," *Journal of Computer and Systems Sciences*, vol. 18, no. 2, pp. 143–154, 1979.

[36] T. Krovetz, "Message authentication on 64-Bit architectures," in *Selected Areas in Cryptography*, E. Biham and A. M. Youssef, Eds. Springer Berlin Heidelberg, 2007.

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

[38] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic program optimization," *Commun. ACM*, vol. 59, no. 2, February 2016 2016.

[39] V. Srinivasan, T. Sharma, and T. Reps, "Speeding up machine-code synthesis," in *Proc. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA, 2016, pp. 165–180.

[40] E. Blem, J. Menon, and K. Sankaralingam, "Power struggles: revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures," in *Proc. of 19th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 1–12.

[41] S. J. Murdoch, "Hot or Not: Revealing hidden services by their clock skew," in *Proc. of ACM CCS*, 2006, pp. 27–36.

[42] Y. S. Shao and D. Brooks, "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor," in *Proc. of IEEE International Symposium on Low Power Electronics and Design (ISLPED), Beijing*, 2013, pp. 389–394.

IX. Appendix A
Different Space-Time Bounds for Horner-Rule Programs

When implemented on commodity processor architectures, the space-time optimality of a k-randomized polynomial depends primarily on the performance of the Horner-rule steps in $\mathbf{Z_p}$, where p is an odd prime. The optimal implementation of both the loop control computation is easily achieved on these processors. The Horner-rule steps are defined on *unsigned integers* coefficients a_i and input x as $a_{i+1} \times x + a_i \ (mod\ p)$, $i = d-1, \ldots, 0$. Their implementation on different commodity processor architectures illustrates the how optimal space-time bounds differ on different instruction set architectures (ISAs).

Division-based Implementations. The $mod\ p$ implementation of the Horner-rule steps avoids all register carries and overflows. 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. 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. Ordinary integer division by constant p can be simulated by a multiplication and a shift.

Division-less Implementation. In commodity processors the mod instruction is always more expensive than other instructions such as multiplication or addition [31, 32] in terms of both execution time and energy use. In fact, when computing a Horner-rule step all division instructions, not just the mod, can be avoided in optimal implementations at the cost of higher memory bounds.

A Horner-rule step can be implemented by a unsigned integer multiplication and two addition instructions [35]. Reductions $mod\ p$, where p is the *largest prime* that fits into a w-bit word (i.e., where $p = 2^w - b$ is a pseudo Mersenne prime) are performed efficiently without divisions and optimal implementations exist in cryptographic libraries. Register carries are either handled by single conditional additions or avoided by judicious choice of input x. In a full polynomial evaluation by the Horner rule the reductions $mod\ p$ are postponed until the final Horner-rule value is output.

Recall that the Horner-rule step is expressed as $z = a_{i+1} \cdot$

$x + a_i \ (mod\ p)$, where $i = d-1, \ldots, 0$. 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} 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 $b \cdot R_{hi}$ can be avoided by picking input $x \leq \lfloor \frac{2^w}{b} \rfloor$ at the cost of a negligibly higher collision probability in the output of a k-randomized polynomial.

In the full evaluation of a 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[7]. The conditional test is implemented by a single three-operand instruction or by two instructions when only two-operand instructions are supported.

Krovetz and Rogaway [30] succinctly illustrate an optimal division-less implementation of the Horner-rule step with x86−32 code using only eight instructions (without counting the final $mod\ p$ reduction) where $p = 2^{32} - 5$. A MIPS processor would require two additional *move* instructions since its R_{hi} and R_{lo} registers are not directly addressable. The memory bounds of these programs far exceed the four-instruction implementation using $mod\ p$, 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 [30, 36]; 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 ($p = 2^{127} - 1$).

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 programs even on a single processor ISA. However, every *distinct* space-time optimal program has a different instruction encoding for the Horner-rule program, and hence a *different code identity*. This means that each space-time optimal implementation yields a different and unique evaluation result depending on its code identity; see Section IV. Thus, an adversary cannot increase the chances of circumventing the establishment of malware-free states by choosing one optimal implementation versus another.

Optimal space-time programs that minimize the time bound are often preferable in devices with large primary memories where evaluations may take up to a few minutes; e.g., for unusually large k [1]. In practice, 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 [37, 38]. 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 [38]. Program synthesis tools may also be applicable [39].

[7] 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 $0 \leq a, b < 2^{61}$. Hence, $z = (z \ \mathbf{div}\ 2^{61}) + (z \ \mathbf{mod}\ 2^{61})$ [36]. 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.

X. Appendix B Measurement of Space-Time Bounds

A. *Generality and robustness of space-time measurements*

Is the necessity of space-time optimal bounds for baseline measurements intended to detect persistent malware a general and robust condition in practice, given that it is implied by an energy model designed for other purposes [24]; viz., Section III-B? To answer this question, we note the energy model's generality and robustness: experimental evidence shows that it applies to all types of single-core CPU configurations, from low-end to high-end, and to all instruction-set architectures.

To address the generality question, we observe that De Vegeleer et al.'s energy model implies a *convexity rule* between E_{sys} and frequency f of single-core CPUs[8]. That is, there exists a unique point f_{opt} where E_{sys} for a given computation $\mathbf{C_{nonce}}(\cdot)$ is optimal at a constant temperature[9] [24]. Experimental evidence shows that the convexity rule applies to single-core architectures of all modern processors [24–26, 29] and for most instructions (e.g., integer arithmetic and logic) of some processors, such as ARM Cortex A7 [28].

To address the robustness question, we note the *ISA independence* of this and other energy models. In all processors, the energy expended by each integer instruction relative to other integer instructions follows the same ordering relations as the latency of each instruction relative to the other integer instructions [27, 28]. Furthermore, energy measurements performed on a variety of RISC and CISC processors show that the minimum energy consumption is independent of the specific instruction set [40].

B. *Simplicity of measuring space-time bounds*

Measurement simplicity offers practical justification for space-time measurements instead of energy consumption. First, in space-time measurements, the external verifier need not vary the temperature, voltage, and frequency at which these measurements are taken. Instead, it can simply measure time in $\mathbf{C_{nonce}}$'s clock cycles, cc, at any constant temperature, frequency, and voltage values thereby making cc is independent of these values; see Section III-B. This helps remove the effects of temperature on CPU power, voltage and frequency [23], and hence execution time. Measurements in different geographic areas will yield consistent results [41].

Second, the external verifier need not measure the ratio of

[8] This rule is derived by expanding the expression of P_{cpu} in terms of dynamic power and magnitude of leakage currents, taking advantage of the voltage/frequency linearity, and normalizing E_{sys} by P_{back} and cc.

[9] The temperature is kept constant to remove its influence on $P_{cpu,i}$ [23]

background power to CPU power, P_{back}/P_{cpu}, to set the clock frequency f as a function of f_{opt} for energy minimization [24]. For instance, for mid- to high-end servers, $P_{back}/P_{cpu} \geq 1$ and $f_{opt} > f_{max}$. Thus, the verifier would have to set the CPU's frequency to f_{max} before measuring the minimum energy of $\mathbf{C_{nonce}}(\cdot)$'s program on input v. For low-end servers, $P_{back}/P_{cpu} < 1$ and $f_{min} < f_{opt} < f_{max}$. Here, the external verifier must set the CPU frequency to f_{opt} to measure the minimum energy of $\mathbf{C_{nonce}}(\cdot)$'s program on input v. In contrast, for measuring optimal space-time bounds, the verifier often sets the CPU frequency to f_{max}, which may differ from f_{opt}; see Section VI.

Third, whenever the per-instruction energy consumption is *modeled* – not measured – the accuracy of up to 5% [42] is unlikely to be useful for malware detection. In contrast, an instruction's clock-count latency is a common measurement parameter [31, 32], and a precise program cycle count, cc, can be obtained; see Section X-D.

C. Measurement atomicity

To perform accurate time measurement, the external verifier must be *locally connected* to the system [1]. Local connectivity assures that the *nonce* input and $\mathbf{C_{nonce}}(v)$ output transmission take place atomically in a fixed small amount of time via the local system bus. Hence, these times can be measured accurately by the external verifier in bus cycles, or equivalently in CPU clock cycles.

External verifiers must disable synchronous events (e.g., a watchdog time interrupt) on the untrusted system. Otherwise, they could trigger after the timely result is read by the verifier, which could reload a malware program from the secondary storage or network into the primary memory and execute it. Note that even if the verifier could be internal to the system, this concern would still arise since the asynchronous event could trigger between the last instruction of $\mathbf{C_{nonce}}$ and the first instruction of the time measurement; i.e., a classic time-of-check-to-time-of-use exploit.

D. Clock-cycle accurate measurements

To perform deterministic measurement of $\mathbf{C_{nonce}}$'s time bound, many of a system's architecture features must be disabled whereas others must be *set* to fixed values *verifiably*; see Section VI. In addition, we use of a latency-bound family of programs $\mathbf{C_{m,t}}$ which ensures that complex integer computations (e.g., coefficient computation, Horner's rule steps, read-after-write dependencies) cannot be sped up by using accelerated floating point instructions; e.g., multiplications, divisions. These programs ensure that pipelining yields a fixed time-measurement result. However, this does not guarantee deterministic time measurement for all $\mathbf{C_{nonce}}$ program instances because a CPU core may execute some instructions *out-of-order* to fill stalls caused by memory accesses, for instance. Out-of-order execution of instruction sequences must be verifiably disabled for accurate cycle-time clock measurements on modern processor architectures [34]. That is, measurements must serialize instruction execution before, during, and after the execution of $\mathbf{C_{nonce}}$ instructions until the result output. Serialization removes all effects of out-of-order instruction execution.

Paolini [34] illustrates how to perform accurate clock-cycle measurement of a computation – after disabling asynchronous events – by using the *cpuid*, *rdtsc*, and *rdtscp* instructions and saving the values returned by those instructions; e.g., *edx* and *eax* for Intel I-32 and corresponding "r" registers for IA-64.

- *cpuid* is executed at any privilege level to serialize instruction execution with no effect on program flow. It forces the CPU to complete every preceding instruction before continuing program execution.
- *rdtsc* reads the high-order 32-bits of the timestamp counter (*tsc*) into register *edx* and the low-order 32-bits into the *eax* register.
- *rdtscp* waits until all preceding instructions have been executed before reading *tsc*, without preventing execution of subsequent instructions after *tsc* reading.

Using these instructions, the $\mathbf{C_{nonce}}$ *instructions* are bracketed to yield accurate timestamp counter readings, as shown below. These readings are subtracted to yield accurate clock cycle counts, cc. The detailed explanation of the skeleton measurement program below is found in Intel's documentation [34].

```
cpuid
rdtsc
mov edx, [start_tsc_high]
mov eax, [start_tsc_low]

C_nonce instructions on input v

rdtscp
mov edx, [end_tsc_high]
mov eax, [end_tsc_low]
cpuid
```

Briefly, the first *cpuid* instruction serializes the execution of instructions above and below the *rdtsc* instruction, without affecting the timestamp counter reading by *rdtsc*. The *rdtscp* instruction reads the timestamp counter after ensuring that all the $\mathbf{C_{nonce}}$ *instructions* complete execution. The second *cpuid* instruction guarantees that instructions which follow it cannot be executed before *rdtscp*. An instruction that subtracts the first saved timestamp value from the second and stores it in memory follows the second *cpuid* instruction. The stored value counts the CPU cycles used by the $\mathbf{C_{nonce}}$ *instructions*, cc, accurately. This cc value is returned to the external verifier along with the computation result; e.g., it can be *xor*-ed into the computation result before output. Note that the instructions bracketing the $\mathbf{C_{nonce}}$ *instructions on input v* from above follow the *nonce* input and precede the loading of the *nonce* into the general purpose processor registers. The instructions bracketing the $\mathbf{C_{nonce}}$ *instructions on input v* from below precede the output instructions.

E. False negatives and false positives

Use of target claw free functions in the simple protocol outlined in the introduction shows that the probability of a false negative is a small constant divided by the largest prime p that fits in a processor word. For example, when using k-independent randomized polynomials in the cWRAM for a single device, this probability is bounded by $9/p$ [1]. Repeating the protocol n times drives this probability to zero; e.g., for $n = 2$ and $p = 2^{32} - 5$, this probability is lower than 2^{-52}. Similar low probabilities should hold for real systems.

The false positive rate depends on the accuracy of the external verifier's measurements, as illustrated in Sections X-C and X-D above. If the external verifier is connected to the local system bus [1, 6], these measurements can be clock-cycle accurate and the rate of false positives can be insignificant.