POWERALERT: Integrity Checking using Power Measurement and a Game-Theoretic Strategy

Ahmed M. Fawaz, Mohammad A. Noureddine, and William H. Sanders
University of Illinois at Urbana-Champaign
{afawaz2, nouredd2, whs} @illinois.edu

***Abstract*—We propose POWERALERT, an efficient external integrity checker for untrusted hosts. Current attestation systems suffer from shortcomings, including requiring a complete checksum of the code segment, from being static, use of timing information sourced from the untrusted machine, or using imprecise timing information such as network round-trip time. We address those shortcomings by (1) using power measurements from the host to ensure that the checking code is executed and (2) checking a subset of the kernel space over an extended period. We compare the power measurement against a learned power model of the execution of the machine and validate that the execution was not tampered. Finally, POWERALERT randomizes the integrity checking program to prevent the attacker from adapting. We model the interaction between POWERALERT and an attacker as a time-continuous game. The Nash equilibrium strategy of the game shows that POWERALERT has two optimal strategy choices: (1) aggressive checking that forces the attacker into hiding, or (2) slow checking that minimizes cost. We implement a prototype of POWERALERT using Raspberry Pi and evaluate the performance of the integrity checking program generation.**

I. INTRODUCTION

Computer systems manage most aspects of our lives including those related to critical infrastructures, communication, finance, and healthcare. Those computers enable better control of the systems achieving more efficiency while promising reliability and security. In reality, the promise of security is elusive and is often disrupted by new exploits and attacks. Over time, those attacks are becoming more sophisticated, targeted, and elusive. Since most systems that are intended to be secure may be compromised by some attacker, intrusion resiliency as a protection strategy has a better chance at improving security than protection alone. The resiliency strategy considers compromises inevitable; it protects the best it can but moves to detect attacks and devises methods to control a compromise while maintaining an acceptable level of service. Critical to such a strategy is the ability to detect compromises and devise methods to find optimal responses that modify the system to maintain security and service goals.

Detection of compromise is conditioned on the ability to guarantee that sensors that monitor hosts are not themselves compromised. Sophisticated attacks, known as advanced persistent threats (APTs), target high-valued assets [16]. APTs, often meticulously planned, are slow and stealthy operations that span months to years. As seen in previous attacks, such as Stuxnet, APTs maintain stealthiness by using custom attack tools, compromising sensors to thwart detection, and using rootkits to change kernel operations.

As such, resiliency strategies are vulnerable to a well-planned adversary. The adversary will attempt to manipulate monitoring information by tampering with sensors, necessary for intrusion detection, leading the defender to believe in a false state of security. Thus, it is essential to validate the integrity of software running on an untrusted machine whose aim is to provide resiliency. Checking software integrity is challenging when faced with a dedicated attacker; the attacker will learn from previous experience and subvert the detection methods; she will manipulate measurements that originate from within the untrusted machine; she will attempt to hide to avoid detection. Even in the presence of trusted hardware within the untrusted machine, an attacker can still manipulate the runtime state of the machine after boot, or also exploit the trusted base to manipulate the stored trusted state. We propose a system that tackles the following question: *How to validate the integrity of software against a slow and stealthy attacker without any trusted components in the machine?*

We make the following requirements for the solution to our problem: it should (1) be independent of the machine to be checked, (2) use a trustworthy base, as opposed to a trusted one, that cannot be exploited by an attacker, and (3) allow for low-cost runtime integrity checking. Today's golden standard in security uses in-machine tamper-resistant chips (such as TPM or AMT) that hold secrets to generate chains of trust. Those trusted components are assumed to be uncompromisable; a property that we cannot verify. Thus, solutions that are dependent on such hardware are still vulnerable to undiscovered exploits which raises the security bar but does not alleviate the problem.

To address the problem, we propose POWERALERT, a low-cost out-of-box integrity checker that uses the physics of the machine as a trustworthy base. Specifically, POWERALERT directly measures the current drawn by the processor and uses normal behavior models to validate the behavior of the untrusted device; POWERALERT suspects the presence of an attacker if the measured behavior deviates from the normal model. This is based on the observation that an attacker attempting evasive or deceptive maneuvers will have to use extra energy thus drawing additional current. Traditional techniques that use side-channel information to validate the behavior of an untrusted machine are ineffective against persistent and adaptive attackers; they used network timing information [19] which is inaccurate as it depends on network conditions and can thus be subverted by a smart attacker. In our work, POW-

ERALERT uses the current signal as a trustworthy and accurate side-channel, sampled at 1 million samples per second, to measure the energy used- and the time needed to perform integrity checking.

POWERALERT tackles the classical problem of the *static defender*, in which an adaptive attacker can learn the protection mechanism and evade or subvert the defenses (Section VIII). In our work, we level the playing field by introducing three main contributions. First, we randomly generate a new integrity-checking program (IC-program) each time we attempt to verify the integrity of a machine; POWERALERT generates the IC-programs in a way that is unpredictable to the attacker (Section IV). Second, each time the POWERALERT-protocol is initiated, we set each IC-program to check a small subset of the system's memory. The subset changes each time a new IC-program is generated; such that over the lifetime of the machine, the whole space of addresses is checked many times. This helps in achieving two desirable goals: (1) it avoids the problem of the attacker predicting the fraction of the memory we are checking and taking precautions and (2) it minimizes the performance overhead needed to perform continuous-time integrity checking.

Operationally, each time POWERALERT checks the integrity of the machine it initiates the POWERALERT-protocol. It sends a randomly generated IC-program to the computer and measures the current drawn by the processor (Section III). During execution of the POWERALERT-protocol, the untrusted machine is expected to load the program, to run it, and to return the output. POWERALERT validates that the untrusted machine did not deviate for the expected script by (1) checking the output of the IC-program, and (2) comparing the frequency spectrum and timing information extracted from the measured current signal to a learned energy model (Section V). Any deviation serves as an indication that the untrusted machine did not perform the expected tasks.

Finally, we alleviate the problem of an attacker hiding its traces every time the POWERALERT-protocol is started by finding the defender's optimal initiation strategy. We find the optimal strategy by modeling the interactions between the defender and the attacker as a continuous- time game (Section VI). In the game, the attacker attempts to evade the defender's integrity checks by disabling its malicious activities while the defender has to balance the frequency of its checks along with the performance overheads that the target machine suffers. Our analysis of the Nash equilibria of the game reveals that the defender can control the attacker's behavior by changing the frequency of her integrity checks. We observer that (1) the defender can force the attacker to risk detection by decreasing the frequency of the checks, and (2) the defender can force the attacker to hide more often by increasing the rate of the checks. In the former case, the defender intends to maximize the detection rate while in the later, she minimizes her checking costs.

II. SYSTEM DESCRIPTION

In order to address the problem of dynamic integrity checking of software (mainly the static memory in the kernel) on an untrusted machine, we propose POWERALERT, an out-of-box device that checks the integrity of an untrusted machine. In this section, we describe our approach for the solution explaining the architecture of POWERALERT, protection assumptions, and the threat model.

A. Problem Description

When an attacker compromises a machine, they have complete control of the operations of the machine by injecting code that modifies memory, function pointers, kernel code, and device drivers. A rootkit, for example, typically changes functions pointers in the kernel's memory space to redirect execution to malicious functions. A defender wanting to check if the machine is compromised searches for modifications in the state of the machine. Unless the defender extracts the state of the machine without any processes running, the attacker can tamper with the reads to misrepresent the state of the machine. The problem is *how to validate that the results of integrity checking are untampered*.

B. Solution Approach

POWERALERT is a trusted external low-cost box tied to the untrusted machine. Figure 1 shows the architecture of POWERALERT. The box runs an integrity checking protocol, POWERALERT-protocol, on the untrusted machine. The protocol starts by sending a challenge to the untrusted machine. The challenge is a randomly generated integrity checking program, called the IC-program. The machine is expected to respond to the challenge by running the program, which hashes a randomly selected part of the memory, and sending back the output. POWERALERT checks the response and compares it to the known state of the untrusted machine. The process is repeated over the life of the machine. Thus POWERALERT avoids checking the whole state of the machine, and instead checks small portions of the machine periodically. If the machine is not compromised, the IC-program will always respond correctly to the challenge; however, if the machine is compromised, the response will eventually be an invalid.

The attacker might try to deceive POWERALERT by adapting or running parallel operations (such running a virtual machine). In order to validate that only the IC-program is running, POWERALERT measures the current drawn by the processor of the machine and compares it to the current model for normal behavior. The power model is specific to the processor model and thus has to be learned for each machine. During the initialization of POWERALERT, the machine is assumed uncompromised. POWERALERT instruments the machine by measuring the current drawn by the processor while running operations semantically similar to the POWERALERT-protocol. POWERALERT learns a power model specific to the machine that is later used for validation. Finally, an attacker might attempt to adapt to POWERALERT's challenge by analyzing the IC-program and finding an optimized version that hides

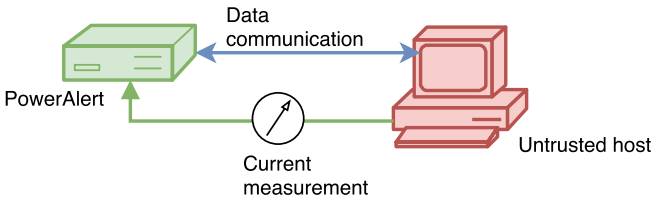


Fig. 1: The components of PowerAlert.

deception operations from affecting the side-channel measurements. So to prevent an attacker from adapting, POWERALERT randomizes the challenge, i.e. the IC-program, each time the POWERALERT-protocol is initiated.

C. Threat Model

We assume a fairly powerful attacker when it comes to the untrusted machine; the attacker has complete control over the software. However, we assume that the attacker does not modify the hardware of the machine; for example, the attacker does not change the CPU speed, or modify firmware. We do not assume any trusted modules or components on the machine to be tested. Our trust base is derived from the randomness of the protocol and physical properties of the CPU. We assume that the attacker is completely untrusted runs deceptive countermeasures to hide her presence and deceive the verifier, and attempt to reverse-engineer the integrity-checking program for future attempts. We aim for our approach to resist the following attacks:

Proxy attack: The attacker uses a proxy remote machine with the correct state to compute the correct checksum and returns the result to the verifier.

Active analysis: The attacker instruments the IC-Program to find memory load instructions in order to manipulate the program.

Static analysis: The attacker analyzes the IC-program to determine its control flow and functionality within the time needed to compute the result. The attacker can precompute and store the results, find location of memory load instructions, or find efficient methods to manipulate the IC-program [25].

Data Pointer redirection: The attacker attempts to modify the data pointer that is loaded from memory.

Attacker hiding: The attacker uses compression [21] or ROP storage [4] in data memory to hide the malicious changes when the POWERALERT-protocol is running.

Forced retraining: The attacker forces POWERALERT to retrain models by simulating a hardware fault resulting in a change in hardware.

D. Assumptions

In this work, we assume that POWERALERT is a trusted external entity that use a trusted untampered channel to connect to the untrusted machine. While this assumption can be relaxed by using authentication, we opt to address it in future work. We assume that POWERALERT has complete knowledge of the uncompromised state of the machine. POWERALERT uses the known uncompromised state to verify the output from the untrusted machine. Finally, the current measurements provide a trustworthy side-channel. Those measurements are directly acquired by tapping the power supply to the CPU and thus they cannot be tampered with; the learned models are based on the physical properties of the system which cannot be altered. Any attacker computation, such as static analysis of the IC-program, will manifest in the current signal.

III. POWERALERT PROTOCOL

We model the interaction between POWERALERT and the untrusted machine as a challenge-response protocol between a verifier and prover. We name the protocol that defines this interaction the POWERALERT-protocol. The goal of the checker is to verify that the prover has the correct proof; in this case, we are interested in the state of the kernel text and data structures. On a high-level, the verifier requests the state of a random subset of the kernel state and the prover has to produce the results. Instead of directly requesting the memory locations, the verifier sends a randomly generated function that hashes a subset of the kernel state. The verifier uses current drawn by the CPU, a side-channel measurement, to validate the expected runtime and energy of the response. The POWERALERT-protocol is repeated over time; positive results increase confidence that the kernel's integrity is preserved. In the following, we describe the interactions in the POWERALERT-protocol.

The protocol works as follows. At a random instance in time, based on the initiation strategy developed in section VI, the verifier initiates the POWERALERT-protocol. The verifier starts by randomly generating a hash function f, a random function to generate a random set of address positions $\mathcal{L}$. In this setting, the hash function f is the IC-program. The verifier sends the random parameters $< f, \mathcal{L} >$ to the prover. The prover is then supposed to load the hash function, f and run it with inputs $\mathcal{L}$. Meanwhile, POWERALERT measures and records the current drawn by the processor $i(t)$. Subsequently, the prover sends the output of the hash function back to the verifier. Finally, the verifier stops recording the current trace, confirms the output, and validates the expected execution with $i(t)$– the measured current drawn by the processor.

The verifier introduces uncertainty by changing the hash function, order of-, and the subset of-addresses. The uncertainty makes it hard for a deceptive prover to falsify the output. Changing the hash function prevents the attacker from adapting to the verifier's strategy; changing the addresses and nonce prevents the attacker from predicting the verifier's target.

In the following sections, we define the method for generating the hash functions, the strategy for picking a subset of memory addresses, and the method for measuring current and trace validation.

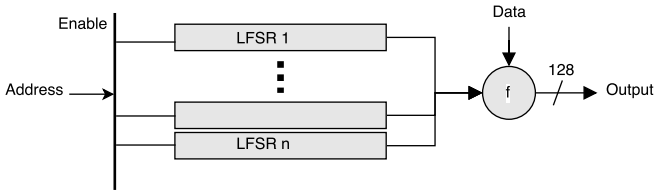


Fig. 2: General Architecture of the Hash Function

IV. Integrity Checking Program

In this work, we take a different approach to address the problem of the static defender. Instead of building the strongest mechanism possible [19], we build a changing mechanism that prevents the attacker from adapting. Specifically, we randomly generate a new IC-program each time the PowerAlert-Protocol is initiated. Moreover, the IC-program has to be resistant to active and static static analysis. To counter active analysis, we change the program every time and thus making it harder for the attacker to catch-up. To counter static analysis, the program is obfuscated by flattening the control flow structure so that attacker analysis will show up in the power trace. We present the method for generating the IC-program in the following sections.

A. IC-Program Structure

The IC-Program's purpose is compute a hash of a subset of the state of the untrusted machine, in order to assess the integrity of the machine. The general flow of the program is a loop that reads a new memory location and updates the state of the hash function. We obfuscate the high-level structure by flattening the control graph of the program using the technique in [27]. The obfuscated program makes it harder for the attacker to locate the load instructions necessary for a memory redirection attack. Any static or active analysis will be observed on the power trace and thus can be detected. Note that the program that is randomly generated is not polymorphic, that is the functionality of the program changes, not just the structure.

A new hash function is used for every run of the protocol. We chain randomly generated LFSRs, the outputs of which are combined using a Boolean function (such as XOR). Figure 2 shows the high-level configuration of the hash function. The outputs of the LFSRs are accumulated with the data in a k bit vector.

The chaining strategy defines the logic for enabling the LFSRs. The input of the enable logic is the memory address being processed, not the data itself. By using the memory address, the attacker will have a harder time to perform a memory redirection attack. The logic is constructed by creating a random binary tree of depth n. The tree defines the control flow of each loop in the IC-program. The control variable at each level is a unique memory address bit. Each level decides if an LFSR is enabled or not. For each node, an LFSR is enabled/disabled, and then the program counter jumps to either child by comparing an address bit. In case the node only has one child, the jump instruction will be omitted for a continuous execution.

B. LFSR generation

On the other hand, each LFSR is also randomly generating using randomly generated irreducible polynomials in a Galois Field. An LFSR is related to polynomials in Galois field $GF(2)$. The process for generating maximal LFSRs uses irreducible polynomial $p(x)$ of degree n. A maximal LFSR has the highest period, the period of the LFSR is the time it takes for the register to return to its initial state. A short period makes it easier to predict the output. A polynomial is irreducible if $x^{2^n} = x \mod p(x)$. For a polynomial $p(x)$, the n-bit Galois LFSR is constructed by tapping the positions in the register that are part of $p(x)$. In operation, bits that are tapped get XOR'ed with the output bit and shifted, while untapped bits are shifted without changed. The output bit is the input to the LFSR. We generate random polynomials and apply the Ben-Or irreducibility test [11]. The polynomials are generated by sampling the uniform distribution, $unif(1, 2^n - 1)$. For example, $p = 24577$ (binary representation 110000000000001) encodes an LFSR where bits 1, 14, and 15 are tapped, that is $p(x) = 1 + x^{14} + x^{15}$.

V. Power Analysis

We verify the execution of PowerAlert-protocol using the current drawn by the processor. We learn the normal Power Finite State machine (PFSM) model using training data from the machine. Then for each round of the PowerAlert-protocol we extract the power states and confirm that they are generated by the normal model.

In the following section we explain the method for current measurement, the method to extract the power states from a current signal, the high-level PFSM model, the method to learn the normal parameters of the model, and the method for validation. Finally, we use the learned model to aid in the parameter selection for IC-program generation.

A. Measurement Method

The current drawn by the processor is measured using a current measuring loop placed around the line. Our setup works for computers with motherboards that have separate power line for the processor. Our generation and verification algorithms are not limited to any sampling rate; in fact, the algorithms can be adapted for any sampling rate depending on the needed accuracy. We measure the current directly by tapping the line from the power supply to the CPU socket on the motherboard; as opposed to measuring the power usage by using the instrumentation provided by the processor as the data will pass through the untrusted software stack. Such data is susceptible to manipulation and cannot be trusted as an absolute truth. On the other hand, direct measurement provides a trusted side channel that we use to validate that an untampered PowerAlert-protocol is executed.

The measured current signal is either stored for model learning or processed in real-time for PowerAlert-protocol execution validation.

B. Extracting the Power States

We observe that the current drawn by a processor during an operation takes the form of multi-power states, where each state draws a current with a unique profile. Such behavior is consistent with the way a processor work: different operations use different parts of the processor's circuitry. As each part of the processor switches dynamic current passes through the transistors. Thus different combinations of the circuitry will draw current with different profiles.

We start by extract the segments in the current signal that belong to different power states, by finding areas of change of operation modes. First, we filter the signal $i(t)$ using a lowpass filter, $h_1(t)$ to remove high frequency noise from the signal, $i_l(t) = i(t) * h_1(t)$, leading to the orange signal in Figure 3a. Then we compute the derivative of the filtered signal, $I(t) = di_l(t)/dt$. The derivative will be near zero for the segments of $i(t)$ with near constant current level (after filtering) and mode changes will be non-zero. We filter the derived signal $I(t)$ with another lowpass filter, $h_2(t)$, to remove more high frequency noise, $I_f(t) = I(t) * h_2(t)$. Finally, we extract the segments between non-zero changes by computing a threshold of the signal using an indicator function $I_{>\lambda}(t)$. The indicator function is 1 if the absolute value of a signal is greater than λ. The transformation leads us to finding the segments of the signal with abrupt current changes, those segments are the power states. For each segment in $i_s = i(t)$ for $t_a < t < t_b$, we compute its average $I_s = \frac{1}{t_b - t_a} \int_{t_a}^{t_b} i_s(t)dt$ and frequency spectrum, $I_s(w) = \mathcal{F}\{i_s(t)\}$. Both indicators describe the current profile during the operation. The duration of each state is computed as $\tau = t_b - t_a$.

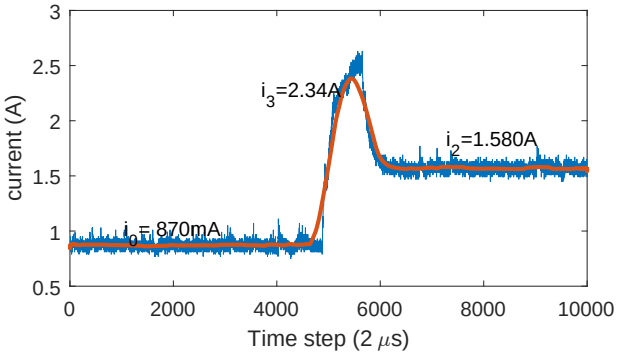
C. Power Execution Model

We model the operations that take place in POWERALERT-protocol, network and hashing operations, using an extended Power Finite State Machines (PFSM). The model and timing of each power state are used by POWERALERT to validate that the POWERALERT-protocol was running untampered.

The PFSM, proposed by Pathak [23], is a state machine where each state represents a power state S_k. We extend each power state to contain the physical characteristics of each state as a tuple (duration, average current, and frequency spectrum).

The POWERALERT-protocol starts by a network communication (network operation) between POWERALERT and the machine. Then the machine is supposed to load and run the IC-program (hash operation). The overall operation state machine is shown in Figure 4. A PFSM has an initial idle state S_0 with power profile $(\tau_0, I_0, I_0(w))$. When an operation starts, a network receive with a TCP socket, the PFSM moves deterministically to power state S_1. If the network operation is long, then the states cycles between S_0 and S_1 until network communication is complete; this is due to data being pushed from the network card to memory buffers. Figure 3b shows the current trace drawn during a network operation. After communication is done, the hosts loads the IC-program; PFSM moves from the idle state S_0 to S_2. Then the hosts runs the IC-program; PFSM moves to the S_3 state. Figure 3a shows

(a) Memory read and hash current trace



(b) Network operation current trace

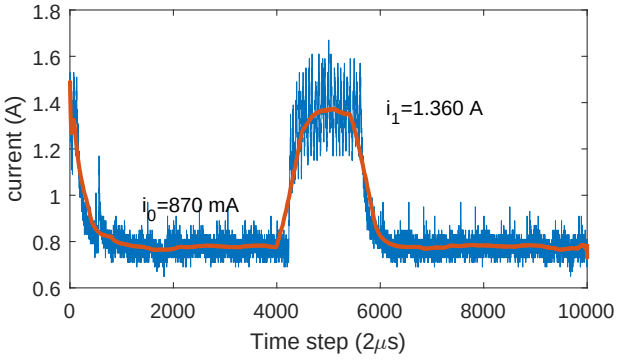


Fig. 3: Current drawn during network and memory read operations.

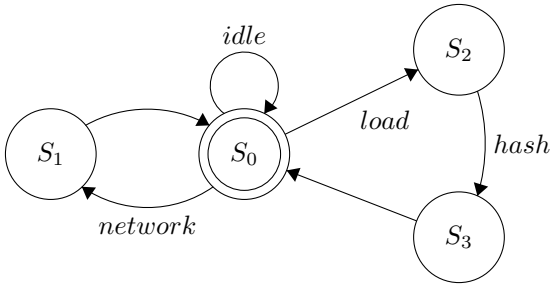


Fig. 4: Power Finite State Machine (PFSM) of POWERALERT-protocol

the current trace drawn during a hash operation. Finally, a network operation returns the output to POWERALERT; the PFSM moves from S_0 to S_1 and back to S_0. In the following, we explain the method for learning the normal PFSM of a machine.

D. Learning the models

For each machine, we assume that we start the from an initial uncompromised state. We establish a power behavioral baseline, build the PFSM and a language for each operation, and learn an execution time model. We initiate the POWERALERT-protocol multiple times and store the current signal

for every run. For each signal the power states are extracted using the method in section V-B and the current profiles of are averaged. Moreover, we establish the idle power state by measuring power when no applications are running.

In our test machine, an AMD Athlon 64 machine running Linux 4.1.13, the average current drawn during the idle state is $870mA$, the average current drawn during the load phase is $2.34A$, the average current drawn during the hash phase is $1.580A$, and the average current drawn during the network operation phase is $1.360A$.

The idle state current depends on many factors including the semiconductor manufacturing process. The manufacturing process determines static power consumption (subthreshold conduction and tunneling current) which is the current draw when the gates are not switching. Thus the current levels in the generated are unique to the machine and need to be learned for each machine. We decided not fold in semiconductor aging into the power model. Aging causes degradation of the transistor leading to failures; however the time scale where aging affects performance is in the order of 5 years. Specifically, aging has no effect on dynamic power [13] but it does affect threshold voltage. The static power is proportional to the threshold voltage [30]. Studies have shown that the threshold voltage varies within 1V during thermal accelerated aging [5] which causes a 0.4% increase in static power. We consider this increase insignificant to incorporate into the model especially that it requires years to happen.

In the following, we learn a timing model using the training data from the machine to be inspected and we propose the method of validating the execution of the POWERALERT-protocol using the learned PFSM and the timing model.

Retraining the models: In order to retrain the model when needed; for example when an operating system is updated. We opt for the following procedure: (1) backup the data in permanent storage, (2) wipe storage, (3) install a clean OS, (4) collect training data and learn the models, and (5) restore permanent storage. This process, given our assumption of no hardware attacks, ensures that the attacker cannot interfere with the training process, as the persistent storage is removed during the training phase.

E. Learning Power State Timing

We use timing information in our system as part of the validation process. Specifically, we confirm that an adversary is not trying to deceive POWERALERT by extracting the timing information (duration) for each power state and compare it to a learned model. By extracting the timing information using the power signal we control the accuracy of the measure as opposed to using network RTT in remote attestation schemes which are affected by the network conditions. Moreover, we have confidence that the timing was not manipulated as it was extracted from an untampered source. We learn the execution time model for the hash phase and the network phase.

All IC-programs have a a complexity $\mathcal{O}(c \cdot N)$ where N is the input size and c is the number of instruction per loop, and use the same type of instructions as any IC-program. We postulate that any IC-program of equal input size N and c number of instructions will have the same execution time. Thus to obtain the training data for learning the timing model, we generate IC-programs for different input size and instruction count and find the execution duration per program. The experiments are repeated multiple times and the results are averaged. We use multivariate regression to learn a model of the execution time of the IC-program. The model uses predictor variables $x = [N, c]$ and a response variable $y = t$ (execution time). For our test machine, the duration of the hash phase has is fitted into a multi variate model $y = 1.3958 + 0.081x(1) - 0.017x(2) + 0.008x(1) \times x(2)$ with mean error $\sigma = 5.4542\mu s$. The mean error of the model is significant because it determines the leeway the adversary. If the error is high, then the attacker has a wide gap to employ evasion techniques. Figure 5 shows the impact of an attacker injecting instructions into the program. The plot shows the current signal measured during the hash phase. The blue signal is the normal behavior, and the orange signal is the tampered one. Both signals have the same power states. However the tampered signal stays longer in the second state.

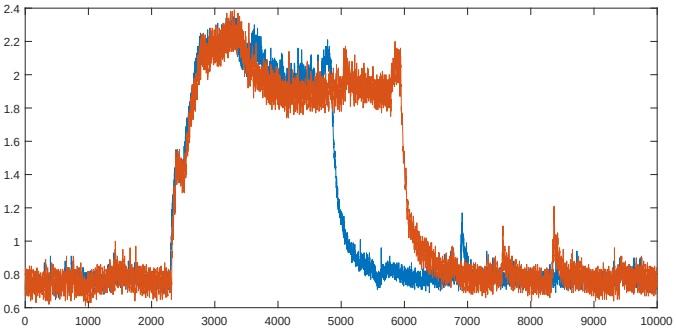


Fig. 5: Timing difference in current signal due to tampering.

During the network phase, the machine is either receiving data or sending the result. When the POWERALERT-protocol is initiated, the CPU performs an IO operation to transfer data from the network card. While when the machine returns the results, the CPU performs an IO operation to transfer data to the network card. We learn the timing model of the network phase by varying the number of bytes to be transferred and then measuring the time it takes to transfer those bytes. Using our test machine, $y_n = 0.129 \times x + 12.48$ is the linear model that predicts the timing for the network phase as a function of the number of bytes (x). The mean error of the model is $\sigma_n = 1.902\mu s$. The model's constant component is the time it takes the OS to create the buffers and the linear component is the time to transfer the data over the buses using DMA. The linear coefficient models the bus speed.

F. Measurement Validation

POWERALERT measures the current drawn in during POWERALERT-protocol and attempts to validate that the trace is generated due to the POWERALERT-protocol PFSM in Figure 4. The power states are extracted from the

current signal resulting in a sequence of states $\mathcal{S} = S(0), S(1), S(2), \ldots, S(n)$. A state is detected by matching the frequency spectrum of the current signal to the stored spectra in the PFSM, and then the duration of each state is compared to the learned timing model. For each phase POWERALERT computes the different between the measured duration and the model estimate, an alarm is raised if the difference exceeds the maximum value of the model error and the sampling error.

If errors are not detected during state estimation, POWERALERT then matches the sequence of states to the regular language which is generated by the POWERALERT-protocol PFSM learned by POWERALERT during the training phase (Figure 4).

$$L = (S_0, S_1)^+ (S_0, S_2, S_3, S_0)(S_0, S_1).$$

The first part of the language, $(S_0, S_1)^+$, is the protocol initiation phase. The second part, (S_0, S_2, S_3, S_0), is the hashing phase. Finally, the last part, (S_0, S_1), is the output phase. If the language is not matched an alarm is raised.

G. Parameter Search

The parameters of the IC-program determine the execution duration; if the execution is within the sampling error or model error, then an attacker can hide. Thus, we design the IC-program such that added instructions can be detected.

In order to find the optimal parameters for the IC-program, the optimization in Equation (V-G) is solved by POWERALERT. The optimization minimizes the total running time constrained by the hardware sampling rate, the cost to run the IC-program, and the coverage required. Where N is the input set size, c is the IC-program size, $y(\cdot)$ and $y_n(\cdot)$ are the learned models, σ_m and σ_n are the model errors, σ_s is the sampling error, γ is a tolerance factor, $cost$ is the maximum cost, and $coverage$ is the minimum coverage.

$$\begin{aligned} \underset{N,c}{\text{minimize}} \quad & y(N, c) \\ \text{subject to} \quad & y(N, c+k) - y(N, c) > \gamma \cdot \max(\sigma_m, \sigma_s) \\ & y_n(c) > \gamma \cdot \max(\sigma_n, \sigma_s) \\ & c < cost, \ N/N_{total} > coverage. \end{aligned}$$

We compute the parameters of the IC-program for $k = 4$ with a tolerance factor $\gamma = 10$, cost $= 300$, and coverage $= 0.000001$. When the sampling rate is 1 MHz, the input set size is 2,019 bytes for a program size of 40 instructions. While when the sampling rate is 200 KHz, the optimal input size becomes 3,269 for a program size of 40 instructions. Our computations show that the higher the sampling rate the smaller the input IC-program has to be. So if the designer invests more in the hardware capabilities of POWERALERT, the attacker's leeway will be tighter and with low overhead to the machine.

VI. ATTACKER-VERIFIER GAME

We model the interaction the interactions between the *defender* (POWERALERT) and the *attacker* using a continuous time game with actions performed asynchronously. In this game, the defender initiates the POWERALERT-protocol at random times with a pre-defined strategy while the attacker tries to anticipate the defender's strategy and disables the malicious changes to the kernel in order to avoid detection. The attacker's goal is to coincide her actions with the defender's actions so that her malicious activity is hidden when the POWERALERT-protocol is taking place.

In what follows, we find the Nash equilibrium strategy for the game when the defender chooses her action times by sampling an exponential distribution, then the attacker's best strategy is to hide her activities by sampling a different exponential distribution in response to the defender. The Nash equilibrium strategy is the optimal strategy for the attacker and defender such that if either the attacker or defender deviate, their payoff decreases.

A. Formalization as a Game

We present a continuous time game in which a defender (P_d) makes a move to detect an attacker and an attacker (P_a) makes a move to hide in order to avoid detection. A player's strategy defines the time instants at which she wants to make her move. For the attacker, Player a, it is the set $S_a = \{t_{a,0}, t_{a,1}, t_{a,2}, t_{a,3} \ldots\}$. For the defender, Player d, it is the set $S_d = \{t_{d,0}, t_{d,1}, t_{d,2}, t_{d,3} \ldots\}$. A strategy S_i is a renewal strategy if the action inter-arrival times are independent and identically distributed [3].

The attacker's action consists of hiding her malicious activity for a specified period of time (duration α). Such an action would restore the machine to its untampered state, thus avoiding detection in case the defender initiates the POWERALERT-protocol. After the period α elapses, the attacker restores the malicious state of the machine.

The defender's action consists of initiating the POWERALERT-protocol to validate the victim machine's a portion state, the ratio of checked state is p_c; the ratio p_c is the coverage set by POWERALERT. p_c is also the probability of detection when an attacker is present in the machine. An action succeeds when the defender detects the attacker has modified a memory location. The action fails when the detection does not detect an attacker. A failed action does not necessarily mean the absence of malicious activities; the attacker might have changed memory locations not checked by the POWERALERT-protocol.

B. Analysis of Nash Equilibrium

In this section, we study the game in which both players use exponential strategies. First, we find the form of the expected payoff of both players for any renewal strategy; then we compute the expected payoffs for the exponential strategy. Finally, we find the best response strategies for both players and the Nash equilibrium of the game.

Let $y = Z_a(t)$ be the age of the renewal process for the attacker; it is the time since the last move, i.e., $y = t - t_a$. Let $x = Z_d(t)$ be the age of the renewal process for the defender, i.e., $x = t - t_d$. Let the size-bias density function of random variable X with PDF f be $f^*(z) = \frac{1-F(z)}{\mu}$, where $\mu = E[X]$ and $F(z)$ is the CDF of X. The size-bias cumulative distribution function $F^*(z) = \frac{\int_0^z 1-F(x)dx}{\mu}$. Based on the results in [9], $\lim_{t\to\infty} f_{Z(t)}(z) = f^*(z)$ and $\lim_{t\to\infty} F_{Z(t)}(z) = F^*(z)$. In the following compute the probability of overlap between the defender's and attacker's actions. Consider the following cases:

- No overlap, $x \leq y$ or $y + \alpha \leq x$:

$$C_1(t) = \int_{y=0}^{+\infty} \int_{x=0}^{y} f_{a,t}(y) f_{d,t}(x) dx dy$$
$$C_1^* = \int_0^{+\infty} f_a^*(y) \left[F_d^*(y)\right] dy$$

$$C_2(t) = \int_{y=0}^{+\infty} \int_{x=y+\alpha}^{\infty} f_{a,t}(y) f_{d,t}(x) dx dy$$
$$C_2^* = \int_0^{+\infty} f_a^*(x) \left[1 - F_d^*(y+\alpha)\right] dx$$

- Overlap, $y \leq x \leq y + \alpha$:

$$C_3(t) = \int_{y=0}^{+\infty} \int_{x=y}^{y+\alpha} f_{a,t}(y) f_{d,t}(x) dx dy$$
$$C_3^* = \int_0^{+\infty} f_a^*(y) \left[F_d^*(y+\alpha) - F_d^*(y)\right] dy$$

- Attacker running, $y > \alpha$:

$$C_4(t) = \int_{\alpha}^{+\infty} f_{a,t}(y) dy$$
$$C_4^* = \int_{\alpha}^{+\infty} f_a^*(y) dy$$

We compute the time-averaged payoff for the attacker as:

$$\beta_a = \underbrace{-c_l(p_c)(C_1^* + C_2^*)}_{\text{Detection}} + \underbrace{c_a \times C_4^*}_{\text{Attack running}} - \underbrace{c_a \alpha \frac{1}{E[X_a]}}_{\text{Hiding}}.$$

The first term is the cost incurred when the action succeeds and the attack is detected; the second term is the gains of the attacker when the attack is running; and the third term is the costs of the action of hiding.

We also compute the time-averaged payoff for the defender:

$$\beta_d = + \underbrace{c_w(p_c)(C_1^* + C_2^*)}_{\text{Detection}} - \underbrace{c_m \times \frac{1}{E[X_d]}}_{\text{Action}}.$$

The first term is the benefit of detecting the attack (this is when the integrity check determines that the state was tampered), and the second term is the cost incurred when an action is performed.

Consider the case of the exponential underlying random variable. Specifically, let $X_a \sim exp(\lambda_a)$ and $X_d \sim exp(\lambda_d)$. In the following we consider two cases, the defender wanting to minimize the cost of the checks (Theorem 1), or the defender wanting to inflict damage on the attacker (Theorem 2).

First, observe that $f_{a,t}(z) = \lambda_a e^{-\lambda_a z}$, $f^*(z) = 1 - F(z)/E(X) = \lambda_a(e^{-\lambda_a z})$, and $F^*(z) = 1 - (e^{-\lambda_a z})$. So the probabilities for the cases above are as follows:

$C_1 = 1 - \frac{\lambda_a}{\lambda_a + \lambda_d}$ $\quad C_2 = \frac{\lambda_a e^{-\alpha\lambda_d}}{\lambda_a + \lambda_d}$
$C_3 = -C1 + 1 - \frac{\lambda_a e^{-\alpha\lambda_d}}{\lambda_a + \lambda_d}$ $\quad C_4 = e^{-\alpha\lambda_a}$

In computing the best response for each player, we find the rate that maximizes the payoff given the rate of the other player. If the payoff function is convex then the global maximum represents the best response, $\frac{d\beta_x}{d\lambda_x} = 0$. For $\alpha \lll \frac{1}{\lambda}$, the exponential terms in the probabilities are approximated as $e^{-\alpha\lambda} \approx 1 - \alpha\lambda$. Using the approximation, we compute the positive root λ_a^* of $\frac{\partial\beta_a}{\partial\lambda_a} = 0$.

$$\lambda_a^* = \lambda_d \left(\sqrt{\frac{p_c c_l}{2c_a}} - 1 \right)$$

With $p_c c_l \gg c_a$, $\frac{\partial\beta_a}{\partial\lambda_a} > 0$ for $\lambda_a < \lambda_a^*$ and $\frac{\partial\beta_a}{\partial\lambda_a} < 0$ for $\lambda_a > \lambda_a^*$. Thus the best response strategy for the attacker given the defender rate is:

$$BR_a(\lambda_d) = \lambda_d \left(\sqrt{\frac{p_c c_l}{2c_a}} - 1 \right). \tag{1}$$

Theorem 1. *For a defender wanting to minimize its cost, the game with exponential strategy has a Nash equilibrium with* $\lambda_d^* = \Lambda_{d,0}$ *and* $\lambda_a^* = \Lambda_{d,0} \left(\sqrt{\frac{p_c c_l}{2c_a}} - 1 \right)$.

Proof. Examining the defender's payoff reveals that it is strictly decreasing at this rate:

$$\frac{d\beta_d}{d\lambda_d} = -\lambda_d^2 - 2\lambda_a\lambda_d - \lambda_a^2 \left(1 + \frac{p_c \alpha}{c_m} \right) < 0.$$

Note that the root of the payoff function, $\beta_d(\lambda_d^*) = 0$, determines the sign of the payoff function. When the root is positive, the payoff is positive in the interval $0 < \lambda_d < \lambda_d^*$; it becomes negative after the root. On the other hand, if $\lambda_d^* < 0$, then the payoff is strictly negative for all λ_d.

The following equation is the closed-form root of the defender's payoff as a function of the attacker's strategy:

$$\lambda_d^*(\lambda_a) = \frac{\sqrt{\lambda_a^2 c_m^2 + 2\lambda_a c_m(\alpha+1)p + (\alpha-1)^2 p^2}}{2c_m} - \frac{\lambda_a c_m - p\alpha + p}{2c_m}. \tag{2}$$

For all positive values of λ_a and α, the root is positive: $\lambda_d^* \geq 0$. Thus the defender will always have a positive payoff. To maximize the defender's utility, the defender plays at the smallest possible rate $\Lambda_{d,0} > 0$. The best response function of the defender is:

$$BR_d(\lambda_a) = \Lambda_{d,0}.$$

The Nash equilibrium of this game is $\lambda_a^* = BR_a(BR_d(\lambda_a^*))$. □

We consider the case in which the defender decides to inflict damage on the attacker.

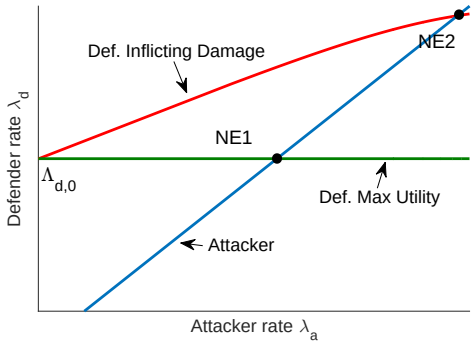


Fig. 6: Game profile with pure Nash equilibrium.

Theorem 2. *For a defender attempting to inflict damage on the attacker, a Nash equilibrium strategy exists such that* $\lambda_d^* = p_c \frac{\alpha\mathcal{X}-1}{c_m\mathcal{X}}$ *and* $\lambda_a^* = \lambda_d^*(\mathcal{X}-1)$ *where* $\mathcal{X} = \sqrt{\frac{p_c c_l}{2c_a}}$.

Proof. Increasing λ_d increases the best response rate of the attacker, thus forcing the attacker to hide more frequently. We propose that the defender play a strategy that leads to $\beta_d = 0$. The goal of this strategy is to harm the attacker before the detection succeeds. We define the best response strategy for the defender in response to the attacker's λ_a as the root of the payoff function:

$$BR_d(\lambda_a) = \frac{\sqrt{\lambda_a^2 c_m^2 + 2\lambda_a c_m(\alpha+1)p + (\alpha-1)^2p^2}}{2c_m} - \frac{\lambda_a c_m - p\alpha + p}{2c_m}.$$

On the other hand, the attacker's best strategy is highlighted in Equation (1). Assuming that $c_m \ll p_c\alpha$, then the best attacker response is to linearly follow the defender's rate:

$$BR_a(\lambda_d) = \lambda_d\left(\sqrt{\frac{p_c c_l}{2c_a}} - 1\right).$$

The Nash equilibrium is computed as $\lambda_d^* = BR_d(BR_a(\lambda_d^*))$. □

Figure 6 shows the game profile for both defender goal and the Nash equilibria due to said strategies. Our analysis shows that we have two Nash equilibria depending on the goals of the defender. If the defender is interested in inflicting damage on the attacker, the defender plays a Nash equilibrium (NE 2) that uses lots of resources for the sake of detecting the attacker; at this equilibrium, the utility of the defender is kept at 0. This NE forces the attacker to hide more often to avoid detection, thus stopping malicious activity. However, if the defender wants to maximize his utility, then he selects the smallest possible rate as a strategy (NE 1). This equilibrium has the attacker playing at a relatively slow rate in response to the defender's slow rate. The defender slowly checks the state of the system and eventually detects the attacker while being unpredictable to an adaptive attacker.

VII. Discussion

In this section, we discuss some security details related to the implementation of this system. Specifically, we discuss the attack surface of PowerAlert and the security concerns with the IC-Program. Moreover, we consider the practicality of our solution, and it's important despite the existence of TPMs.

A. Implementation Details

Each PowerAlert device has a client on the untrusted machine. The client is a low-level module that communicates with PowerAlert. The client is implemented for placement in the kernel or the hypervisor. The communication channel between PowerAlert and the client can be over any medium such as Ethernet, USB, or serial. All those channels are feasible because of the proximity between PowerAlert and the untrusted machine. The use of serial or USB communication is advantageous because it limits the attacker to physical attacks, making man-in-the-middle and collusion attacks harder. If the attacker has physical access to the machine, then she could tamper with PowerAlert.

The client receives the IC-Program as machine code over the communication channel. PowerAlert signs the code, their keys are exchanged during the initialization phase of the system. The signed program allows the machine to attest that PowerAlert is the generator. We propose using a stream cipher as it has better performance than public-private key ciphers or block ciphers. Finally, the client on the machine has to pause execution of other programs when the protocol is initiated. In our implementation, the IC-program is run on a single core using `spin_lock_irqsave()`, while the other cores' executions are blocked by running a sequence of NOP instructions.

The hardware requirements for PowerAlert are minimal. We implemented a prototype using Raspberry PI 2. The prototype uses an ADC to convert the current measurements from the current loop to a digital signal. The ADC uses a sampling rate of 500KHz; most low-cost hardware can handle this sampling rate. Moreover, power state extraction is only performed when the PowerAlert-Protocol is initiated, the operation does not need to be real-time.

B. Comparison to TPM

PowerAlert does not rely on specialized hardware within the untrusted machine such as TPM or Intel's AMT. However, PowerAlert and trusted modules are orthogonal systems; whereas TPMs provide a method for secure boot, dynamic integrity checking is still costly and harder to enforce. PowerAlert provides an external security solution that can be tied to a security management across a wide network. In fact, PowerAlert can use Intel's AMT as a communication channel. Finally, our work demonstrates the need for measurements that do not pass through or origin in the untrusted machine. Such measurements reduce the risk of attacker tampering and mimicry. Having PowerAlert be an external box as opposed to being an internal module aids in separating the boundaries between the entities. The clear boundary allows us to find a

clear attack surface, enables easier alerting capabilities, and easier methods to update POWERALERT when vulnerabilities or new features are added.

C. Space of IC-Programs

A large space of IC-programs is required to prevent reuse. The maximum number of IC-programs that can be generated is the product of the maximum number of binary trees multiplied by the maximum number of irreducible polynomials. Thus, the total number of IC-programs that can be generated is $D_{d,n} = M_d(2) \times t_n$. Where, $t_n = \sum_{i=0}^{2^n} \left(\prod_{k=2}^{i} \frac{i+k}{k} \right)$ is the maximum number of binary trees with depth n, computed using the Catalan number, and $M_d = \frac{1}{d} \sum_{k|d} \mu(k) 2^d$ (the necklace polynomial) is the maximum number of irreducible polynomials of degree d in $GF(2)$. The total number of programs for $n = 40$ and $d = 5$, $D_{5,40} = 1.97 \times 10^{26}$

D. Performance Impact

We deem the performance impact of POWERALERT acceptable. When the POWERALERT-protocol is initiated and the IC-program starts execution, execution of all other tasks on the machine is paused. This is needed in order to ensure that no other tasks interfere with the current measured from the CPU. The POWERALERT-procotol is initiated, on average, once every minute for 0.9 ms for the aggressive defense strategy in Nash Equilibrium 2. The graphical degradation will not be noticeable by a user, because, in terms of graphical responsiveness, the pixel response time should not exceed 4 ms [22]. Moreover, CPU overhead due running the protocol is insignificant at 0.18%.

E. Limitations

In this work, we did not study the variability of the power model. While we argued that the power model will not vary over time due to degradation, however, it does vary due to hardware process variation and operation temperature variation. Moreover, we did not study the false positives of the approach because it is determined by three parameters, the sampling rate, the error of the power model and γ (used to find the size of the IC-program). The designer is encouraged to pick those parameters to decrease the false positive rate to near zero.

VIII. SECURITY ANALYSIS

POWERALERT uses current measurements, timing information, and diversity of the IC-Program to protect against subversion of integrity checking. The power measurements are used to limit the operation of the machine to just the IC-Program while diversity limits the attacker's ability to adapt to our checking mechanisms. In this section, we list the methods in which POWERALERT addresses the attacks discussed in Section II-C.

Proxy attack: In this attacker, the attacker attempts to forward the IC-Program to a remote machine to compute and return the result via the same network link. POWERALERT detects this attack by examining its effects on the current trace and the timing of the network phase. Using the current trace, POWERALERT will observe that network operations took longer than expected as more bytes were transferred between the CPU and the network card. The size of the IC-Program, which was picked by the optimization process described in Section V-G, ensures that our hardware will pick up the retransmission. Any physical attack, such as tapping of the network line or firmware changes to the NIC, are not within our purview.

Active analysis: In this attack, active reverse engineering is used to learn the usage patterns of the IC-Program. POWERALERT changes the IC-Program each time the POWERALERT-protocol is initiated; the diversity renders the information learned by the attacker from the previous run obsolete. The probability that a program will ever get repeated is $1/10^{20}$. Moreover, it is practically impossible for the attacker to predict our next IC-Program. The attacker has to predict the random numbers generated by POWERALERT's random number generator; in this work, we require POWERALERT to use a true random number generator that uses some physical phenomena as opposed to a pseudorandom number generator that can be predicted by a dedicated attacker.

Static analysis: Analyzing a flattened control flow is NP-hard [27]. Thus it will not be possible for the attacker to analyze the program without significant computations. Note that we combine control flow flattening with IC-Program diversity; thus even if the attacker successfully analyzes the IC-Program the solution is not useful for the next run of the protocol.

Data pointer redirection attack: In this attack, the attacker stores an unmodified copy of the data in another portion of memory. When an address is to be checked, the attacker changes the address to be checked to that of the unmodified data. The IC-Program uses the address and the memory content when computing the hash function. To compute a valid hash, the attacker has to change the address to the location of the copy while retaining the original address. In our IC-Program design phase, the designer sets the smallest number of instructions that can be added to the program such that the execution difference is detected when POWERALERT's hardware specifications are taken into account (the sampling rate).

Note that this measure is more effective when combined with the IC-Program diversity. Each time a new IC-Program is generated, the attacker has only one chance to find an injection scheme such that the final number of instructions is less than the threshold we design for. The new program in the next iteration will require a new injection method and thus any runtime method to automatically find the optimal method will require computations that will be detected by our current measurements.

However, the attacker can redirect the data pointer by changing the page table pointer (register cr3); this attack is hard to thwart, we might consider using the System Management Mode (SMM) execution mode which disables paging [29].

Attacker hiding: If an attacker attempts to hide, he or she

must predict when the POWERALERT-protocol will be initiated. POWERALERT's random initiation mechanisms ensure that the attacker cannot predict those instances. Our game-theoretic analysis shows that when the defender is using an exponential initiation strategy, the attackers best strategy is to hide more often if the defender is aggressive. Note that because POWERALERT is using a random strategy, the attack will not always correctly predict the strategy. Thus, some of POWERALERT actions will be run when the attacker is not hiding, leading to detection. The attacker's strategy, to be stealthy, can delay detection but cannot prevent it.

Forced retraining: In this attack, the attacker forces POWERALERT to retrain by simulating a hardware fault that requires a CPU change, to lead POWERALERT to a compromised model. If this occurs, POWERALERT's process is to wipe the permanent storage, retrain using a clean OS, and then restore data. Since we assume that the attacker does not modify the hardware state, by removing permanent storage, we prevent the attacker from affecting the retraining process.

IX. RELATED WORK

1) Timing Attestation: Seshadri *et al.* propose Pioneer [24] extended by Kovah *et al.* [19] a timing-based remote attestation system for legacy system (without TPM). The timing is computed using the network round trip time. The work assumes that the machine can be restricted to execution in one thread. The issue with the work is that the round trip time is affected by the network conditions which the authors do not explore, a heavily congested network will lead to a high variation on the RRT causing a high rate of false positives. Moreover, the restriction of execution in one thread can be evaded by a lower level attacker. In later work the authors discuss the issues of Time Of Check, Time Of Use attacks, we talk the problem in our work. Later work adapted timing attestation to embedded devices [10].

Hernández *et al.* [14] implement a monitor integrity checking system by estimating the time it takes for a software to run. The timing information is sent from the machine to a remote server that uses a phase change detection algorithms to detect malicious changes. The issue of this work is that the timing information is sent by the untrusted machine and thus the information can be easily manipulated. Armknecht *et al.* [2] propose a generalized framework for remote attestation in embedded systems. The authors use timing as a method to limit the ability of an attacker to evade detection. The framework formalizes the goals of the attacker and defender. The authors provide a generic attestation scheme and prove sufficient conditions for provable secure attestation schemes.

2) Power Malware Detection: Several researchers use power usage to detect malware. In WattsUPDoc, Clark [6] collect power usage data by medical embedded devices and extract features for anomaly detection. The authors exploit the regularity of the operation of an embedded device to detect irregularities. The authors however do not investigate mimicry attacks. Kim *et al.* [17] use battery consumption as a method to detect energy greedy malware. The power readings are sent from the untrusted device to a remote server to compare against a trusted baseline. The problem of this work is that the power readings can be manipulated by the attacker as the data is sent through the untrusted software. PowerProf [18] is another in-device unsupervised malware detection that uses power profiles. The power information is similarly passed through the untrusted stack and is thus susceptible to attacker evasion through tampering.

3) Hardware Attestation: Secure Boot [7] verifies the integrity of the system, with the root of trust a bootloader. Later on Trusted Platform Modules (TPMs) uses Platform Configuration Registers (PCRs) store the secure measurements (hash) of the system. Both methods are static in that the integrity is checked at boot time. Dynamic attestation on the other hand can perform attestation on the current state of the system. Such features are supported by CPU extensions (for example Intel TXT). El Defrawy *et al.* propose SMART [8], an efficient hardware-software primitive to establish a dynamic root of trust in an embedded processor, however the authors do not assume any hardware attack.

4) VM based Integrity checker: OSck [15] proposed by Hofmann *et al.* is a KVM based kernel integrity checker that inspects kernel data structures and text to detect rootkits. The checker runs as a guest OS thread but is isolated by the hypervisor. Most VMM introspection intergrity checker assume a trusted hypervisor. Those techniques are vulnerable to hardware level attacks [20], [26], [28]. In our work we do not have any trust assumption as the attestation device is external to the untrusted machine.

5) Checksum Diversity: Wang *et al.* [27] propose using diversity of probe software for security. The authors obfuscate the control flow by flattening the probing software in order to make it harder for an attacker to reverse engineer the program for evasion. While the flattened control flow is hard to statically analyze, the programs are susceptible to active learning thus allowing an attacker to adapt over time. Giffin *et al.* [12] propose self-modifying to detect modification of checksum code modification. The experiments show an overhead of 1 microsecond to each checksum computation, the method is however costly for large programs adding second per check. The authors in [1] use randomized address checking and memory noise to achieve unpredictability.

X. CONCLUSION

In this work we presented POWERALERT, which is an external integrity checker that uses power measurements as a trust base to achieve resilience against a stealthy attacker. By using the power signal, POWERALERT is relying on an untainted, trustworthy, and very accurate side-channel to observe the behavior of the untrusted computer. POWERALERT initiates the checking protocol by sending a randomly generated integrity checking program to the machine. The diversity of the IC-program prevents the attacker from adapting; we showed that the space of IC-programs is practically impossible to exhaust. The untrusted machine is expected to run the IC-program and send its output back to POWERALERT. While the IC-program

is executed, POWERALERT measures the current drawn by the processor to compare it to a learned model. Any deviation from the expected output is an indication of tampering by an attacker. To understand how often checking should be initiated, we modeled the interaction between POWERALERT and the attacker using a time-continuous game. Our analysis shows that POWERALERT can either force the attacker into hiding or have the attacker risk detection.

REFERENCES

[1] T. AbuHmed, N. Nyamaa, and D. Nyang, "Software-based remote code attestation in wireless sensor network," in *Proceedings of the 28th IEEE Conference on Global Telecommunications*, ser. GLOBECOM'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 4680–4687. [Online]. Available: http://dl.acm.org/citation.cfm?id=1811982.1812159

[2] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, "A security framework for the analysis and design of software attestation," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13. New York, NY, USA: ACM, 2013, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516650

[3] U. N. Bhat and G. K. Miller, *Elements of applied stochastic processes*. J. Wiley, 1972.

[4] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 400–409. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653711

[5] J. R. Celaya, P. Wysocki, V. Vashchenko, S. Saha, and K. Goebel, "Accelerated aging system for prognostics of power semiconductor devices," in *2010 IEEE AUTOTESTCON*, Sept 2010, pp. 1–6.

[6] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, W. Xu, and K. Fu, "Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices," in *Presented as part of the 2013 USENIX Workshop on Health Information Technologies*. Berkeley, CA: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/healthtech13/workshop-program/presentation/Clark

[7] D. L. Davis, "Secure boot," Aug. 10 1999, uS Patent 5,937,063.

[8] K. Eldefrawy, A. Francillon, D. Perito, and G. Tsudik, "SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust," in *NDSS 2012, 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA*, San Diego, UNITED STATES, 02 2012. [Online]. Available: http://www.eurecom.fr/publication/3536

[9] W. Felleb, *An Introduction to Probability Theory and Its Applications*, 2nd ed. John Wiley and Sons, New York, 1957.

[10] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A minimalist approach to remote attestation," in *Proceedings of the Conference on Design, Automation & Test in Europe*, ser. DATE '14. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2014, pp. 244:1–244:6. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616606.2616905

[11] S. Gao and D. Panario, "Tests and constructions of irreducible polynomials over finite fields," in *Selected Papers of a Conference on Foundations of Computational Mathematics*, ser. FoCM '97. New York, NY, USA: Springer-Verlag New York, Inc., 1997, pp. 346–361. [Online]. Available: http://dl.acm.org/citation.cfm?id=270376.270489

[12] J. T. Giffin, M. Christodorescu, and L. Kruger, "Strengthening software self-checksumming via self-modifying code," in *Proceedings of the 21st Annual Computer Security Applications Conference*, ser. ACSAC '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 23–32. [Online]. Available: http://dx.doi.org/10.1109/CSAC.2005.53

[13] B. Greskamp, S. R. Sarangi, and J. Torrellas, "Threshold voltage variation effects on aging-related hard failure rates," in *IEEE International Symposium on Circuits and Systems*, May 2007, pp. 1261–1264.

[14] J. M. Hernández, A. Ferber, S. Prowell, and L. Hively, "Phase-space detection of cyber events," in *Proceedings of the 10th Annual Cyber and Information Security Research Conference*, ser. CISR '15. New York, NY, USA: ACM, 2015, pp. 13:1–13:4. [Online]. Available: http://doi.acm.org/10.1145/2746266.2746279

[15] O. S. Hofmann, A. M. Dunn, S. Kim, I. Roy, and E. Witchel, "Ensuring operating system kernel integrity with osck," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011, pp. 279–290. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950398

[16] A. Juels and T.-F. Yen, "Sherlock holmes and the case of the advanced persistent threat," in *Presented as part of the 5th USENIX Workshop on Large-Scale Exploits and Emergent Threats*. Berkeley, CA: USENIX, 2012. [Online]. Available: https://www.usenix.org/conference/leet12/sherlock-holmes-and-case-advanced-persistent-threat

[17] H. Kim, J. Smith, and K. G. Shin, "Detecting energy-greedy anomalies and mobile malware variants," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '08. New York, NY, USA: ACM, 2008, pp. 239–252. [Online]. Available: http://doi.acm.org/10.1145/1378600.1378627

[18] M. B. Kjærgaard and H. Blunck, *Unsupervised Power Profiling for Mobile Devices*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 138–149.

[19] X. Kovah, C. Kallenberg, C. Weathers, A. Herzog, M. Albin, and J. Butterworth, "New results for timing-based attestation," in *2012 IEEE Symposium on Security and Privacy*, May 2012, pp. 239–253.

[20] X. Kovah, C. Kallenberg, J. Butterworth, and S. Cornwell, "Senter sandman: Using intel txt to attack bioses," *Hack in the Box*, 2015.

[21] Y. Li, J. M. McCune, and A. Perrig, "Viper: Verifying the integrity of peripherals' firmware," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 3–16. [Online]. Available: http://doi.acm.org/10.1145/2046707.2046711

[22] R. B. Miller, "Response time in man-computer conversational transactions," in *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, ser. AFIPS '68 (Fall, part I). New York, NY, USA: ACM, 1968, pp. 267–277. [Online]. Available: http://doi.acm.org/10.1145/1476589.1476628

[23] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system call tracing," in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 153–168. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966460

[24] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla, "Pioneer: Verifying code integrity and enforcing untampered code execution on legacy systems," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP '05. New York, NY, USA: ACM, 2005, pp. 1–16. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095812

[25] M. Shaneck, K. Mahadevan, V. Kher, and Y. Kim, *Remote Software-Based Attestation for Wireless Sensors*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 27–41. [Online]. Available: http://dx.doi.org/10.1007/11601494_3

[26] W. Song, H. Choi, J. Kim, E. Kim, Y. Kim, and J. Kim, "Pikit: A new kernel-independent processor-interconnect rootkit," in *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 37–51. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/song

[27] C. Wang, J. Hill, J. C. Knight, and J. W. Davidson, "Protection of software-based survivability mechanisms," in *Proceedings of the 2001 International Conference on Dependable Systems and Networks (Formerly: FTCS)*, ser. DSN '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 193–202. [Online]. Available: http://dl.acm.org/citation.cfm?id=647882.738073

[28] R. Wojtczuk and J. Rutkowska, "Attacking smm memory via intel cpu cache poisoning," *Invisible Things Lab*, 2009.

[29] F. Zhang, K. Leach, K. Sun, and A. Stavrou, "Spectre: A dependable introspection framework via system management mode," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*. IEEE, 2013, pp. 1–12.

[30] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects," *University of Virginia Dept. of Computer Science Technical Report*, 2003.