2017 IEEE 35th International Conference on Computer Design

Subcomponent Timing-based Detection of Malware in Embedded Systems

Sixing Lu, Roman Lysecky, Jerzy Rozenblit
Department of electrical and Computer Engineering, University of Arizona
sixinglu@email.arizona.edu, rlysecky@ece.arizona.edu, jr@ece.arizona.edu

***Abstract*—Network-connected embedded systems require multiple lines of defense against malware. In addition to preventing malware by designing secure interfaces and software, anomaly-based detection is needed to detect malware that successfully infiltrates these defenses. Timing based anomaly detection strengthens embedded system security by detecting anomalies in the execution time of critical software tasks. However, existing timing based anomaly detection methods use a lumped timing model that aggregates the timing of the software, processor architecture, operating system scheduling, etc., and thereby incurs significant variability. We present a non-intrusive hardware detector supporting two novel timing models, including a lumped timing multi-range model that clusters timing into multiple range bounds, and a subcomponent timing model that defines bounds for timing subcomponents of events. Timing subcomponents include intrinsic software execution, instruction cache misses, data cache misses, and interrupts. The experimental results demonstrate that the detection based on subcomponent timing model achieves greater malware detection accuracy compared to the lumped timing model without increasing false positives.**

Keywords—***Timing-based detection; Timing subcomponents; Anomaly detection; Embedded system security; Non-intrusive***

I. Introduction

Historically, embedded systems were not dramatically affected by malware because such systems were physically secured and lacked network connections. With pervasive network access, network-connected embedded systems face increasing threats from malware. For example, the Internet of Things (IoT), which includes devices for healthcare, home monitoring, and smart city applications, among others, is expected to reach several billion devices by 2020 [20], and their security is of great concern given their impact on privacy and safety. A few notable threats, including injecting a fatal dose of insulin in an insulin pump, compromising smart TVs for click fraud, exploiting a vehicle's keyless entry system to steal cars, etc. [19], have demonstrated the potential impacts (life threatening in some cases) of malware.

While traditional proactive methods that seek to secure embedded systems by eliminating vulnerabilities are essential, they are not sufficient to fully secure systems. As system complexity grows, the likelihood of undiscovered vulnerabilities increases. Therefore, runtime detection is needed to detect the presence of malware. Signature-based detection methods [15] have limitations in detecting new and unknown malware [9], leaving systems vulnerable to zero-day exploits. In contrast, anomaly-based detection methods detect execution deviations from a pre-established specification of normal behavior, which can detect the anomalous execution of zero-day malware. Additionally, anomaly-based detection reduces the memory overhead required to store a large database of known malware as only the specification of normal behavior must be stored [3]. Several anomaly-based detection methods have been developed to detect anomalies in an embedded system's execution sequences [25], dataflow [1], memory accesses [23], and execution timing [7][12][13][24].

Modeling execution sequences is efficient and can be implemented at different granularities [16], but these detection methods are vulnerable to mimicry attacks. Mimicry attacks avoid detection by mimicking the correct system execution by interleaving normal operations with malicious operations [21]. Modeling dataflow and memory can detect malware intrusion related to memory data changes, but the normal system model is often very complicated, and data required for detection may not be accessible at runtime.

Modeling the timing of embedded systems increases the sophistication needed to construct mimicry malware, thereby achieving higher detection rates [7]. Attackers may be able to develop malware that mimics timing, but such attacks require a higher degree of sophistication. Since embedded systems are sensitive to timing, the scope of what a mimicry malware can accomplish without changing the timing is restricted. However, previous timing based anomaly detection methods use a lumped timing model to create the specification of the normal system behavior. A lumped timing model uses a single timing value (e.g., wall clock time) to represent the time of events being executed. Fine-grained lumped timing models have been utilized to define timing bounds of basic blocks [12], for which the fine granularity affords reasonably low timing variability and can thus detect malware's timing changes. The fine-grained timing models incur significant performance overhead for software-based implementations and significant area requirements for hardware-assisted implementations. Alternatively, coarse-grained timing models define timing bounds at the function, system call, or task level events, which reduce the overhead but introduce significant timing variability.

A lumped timing model with a single-range bound has the advantage of a simple implementation and low storage overhead, but has a few limitations. First, an event's execution may occur in different execution paths, application tasks, or execution scenarios and may operate on different data. As such, the timing for an event often naturally fits into multiple, non-overlapping timing ranges. Thus, the single-range bound of a lumped timing model is unable to detect malware whose timing falls between the individual non-overlapping ranges. Second, the lumped timing model includes all variability within the system that can significantly increase the timing bounds.

This research was partially supported by the National Science Foundation under Grant CNS-1615890.

1063-6404/17 $31.00 © 2017 IEEE
DOI 10.1109/ICCD.2017.12

Timing variability is introduced by the processor architecture (e.g., delays from cache misses, branch mispredictions), memory hierarchy (e.g., off-chip memory access latency), operating system (e.g., task scheduling and interrupts), etc. This variability may mask malicious timing changes, limits the effectiveness of malware detection and simplifies the mimicry malware creation for attackers.

Toward overcoming the disadvantages of lumped timing single-range models, we present two novel timing models, including a lumped timing multi-range model and a subcomponent timing model. The lumped timing multi-range model decomposes the lumped timing into multiple ranges using a standard clustering algorithm. The subcomponent timing model extracts several timing subcomponents from the lumped timing, including intrinsic software timing and incidental timing from interrupts, instruction cache (I$) misses, and data cache (D$) misses. We further design non-intrusive hardware-assisted detectors with these models for monitoring the timing of events using a processor's trace port with zero performance overhead. Using a smart connected pacemaker prototype, experimental results demonstrate the effectiveness of the subcomponent timing model in detecting sophisticated mimicry malware while keeping false positives low.

II. Related Work

Several efforts use event timing for malware detection. SHIELD [12] detects violations of single lumped timing bounds for basic block within an application, by instrumenting the software binary with special instructions. Similarly, iCUFFs [13] employs single lumped timing to detect code injection and software errors. However, their instrumentation introduces code size and performance overheads, ranging from 6.6% to 44%. SecureCore [24] models the normal system behavior as a distribution of a lumped timing model for each basic block. Anomalous execution is detected if the probability of an observed lumped timing value is lower than a specific threshold (e.g., 5%). The probabilistic threshold used in SecureCore increases the sensitivity to timing variability, and thus achieves good detection rates. However, the threshold also introduces false positives, which can be very high. For the file manipulation malware considered in this paper, even with a low threshold of 1%, SecureCore's false positive rate exceeds 10%. In addition, the fine-grained detection and uniform distribution requirements for each bin incurs a high memory overhead to store 1000s of bins for each basic block's timing distribution. Zimmer et al. [26] designed an intrusion detection method based on the lumped worst-case execution time (WCET) of an application's function call return path. This detection system is intrusive because it is implemented within the OS and requires disabling caches to measure the timing.

These software-based detection methods require modifying the software application to incorporate special instructions, which modifies original system behavior, or require a separate processor to perform the analysis needed to detect anomalies, which incurs high overheads. Alternatively, hardware-assisted approaches can eliminate or reduce this performance penalty. Rahmatian et al. [14] designed a hardware-assisted anomaly detector integrated within a processor datapath that monitors the instruction register to detect system calls. This approach provides faster detection compared to software-based methods and can detect malicious activity without affecting the system performance. However, the approach is susceptible to mimicry attacks and requires modification to the processor core. S3A [11] uses a trusted hardware component to detect if the execution time or activation period of periodical tasks within embedded control systems are too small or large. RAD [7] is a hardware-assisted detector that detects single lumped best-case execution time (BCET) and WCET bounds and monitors function/system calls. These approaches achieve good detection rates, but they all use a lumped timing model for coarse timing that introduces significant timing variability, which then allows some monitored events to be more easily mimicked.

III. System Assumptions and Threat Model

The focus of this paper is the design of timing models and hardware for anomaly-based malware detection given the following assumptions and pre-conditions:

1) The target malware is sophisticated mimicry malware that interleaves malicious code with normal execution by nullifying events (e.g., null pointer arguments, nonexistent files, arbitrary data) to mimic system execution [21]. Note that mimicking is not the malicious goal but rather a way of implementing malware. We integrate sequence-based anomaly detection [8] in all malware detectors by default, which can detect non-mimicry malware and necessitates the need for an attacker to use mimicry malware.
2) An attacker is able to create malware and remotely insert the malware into the system utilizing a software exploit, which may be unknown or known but unpatched at the time of insertion. The attacker is able to determine the execution order of the original software using binary analysis, simulation, or direct access to the software code.
3) Our target system is a smart connected pacemaker prototype that enables the implementation and analysis of different malware, which is not possible using standard embedded application benchmarks. We currently consider either a single core system or a multicore system with static task to processor assignment. However, the detector designs presented in this paper can be directly applied to any embedded applications.
4) The granularity of detection in this paper is at the level of system/function calls in the software application and interrupt service routines (ISR), which we collectively call events. However, detection with coarser or finer granularity (e.g., time between two system calls) is possible and would follow the same design principles.
5) Any system updates would simultaneously update the binary and the normal execution model, the latter of which is configured within the hardware detector, which is assumed to be trusted, during the secure boot process.

Typical threats for embedded systems used within healthcare applications include: a) manipulating cardiac log to deceive the physician into making incorrect diagnoses, b) leaking sensitive healthcare data to unauthorized third-parties, or c) interfering with the system execution to affect the essential functionality. While public malware repositories are available, the malware attack templates contained therein focus on how to

exploit specific vulnerabilities, which is not the focus on this paper. Instead, this paper focuses on detecting malware that has already exploited such a vulnerability and is actively executing in the system. As such, we used four mimicry malware that achieve these malicious goals, including a file manipulation malware, an information leakage malware, and two fuzz malware, which are constructed based on known malware.

IV. Timing Model Analysis

A. Lumped Timing Single-Range (LTSR) Model

A lumped timing single-range (LTSR) model uses a single timing value to represent the normal execution time of each event and defines the timing bound as a single range [BCET, WCET]. The problem of determining BCET and WCET has been widely studied for embedded systems and numerous tools have been developed to automate this process, in which this problem can be solved either by statically analyzing the software application or by dynamically training the system [23].

Figure 1(1) presents an example of how the LTSR model can detect anomalous timing for three events, specifically the *mutexunlock*, *fileexists*, and *genhttpheader* functions, from the smart connected pacemaker application for a fuzz malware (see Section VI). In Figure 1, LTSR can detect the malware when the malware timing (dashed line) has non-overlapping parts with normal single range (solid line). The probability of detecting malware is higher if the malware timing has less overlap with the normal execution's lumped timing.

The advantage of the LTSR model is its simple implementation in both the training and runtime detection. Only the minimum and maximum timing values of each monitored event need to be stored in the detector, which reduces storage overhead. The disadvantage of the LTSR is twofold. First, an event's execution may be located in different paths under different scenarios or with different arguments, such that the normal timing may fit into multiple non-overlapping ranges. Therefore, a single timing range cannot detect malware timing that is located between these ranges. Second, lumped timing includes all variability within the system and yields less sensitivity to malware timing (e.g., an interrupt can extend the normal range). Therefore, an extended normal timing range cannot detect malware timing that deviates slightly. For the first disadvantage, dividing the single range of the lumped timing into multiple ranges can increase detection accuracy, which leads to the idea of the lumped timing multi-range model. For the second disadvantage, decomposing the lumped timing into subcomponents can reduce the timing variability and thus increase detection accuracy, which leads to the idea of subcomponent timing model.

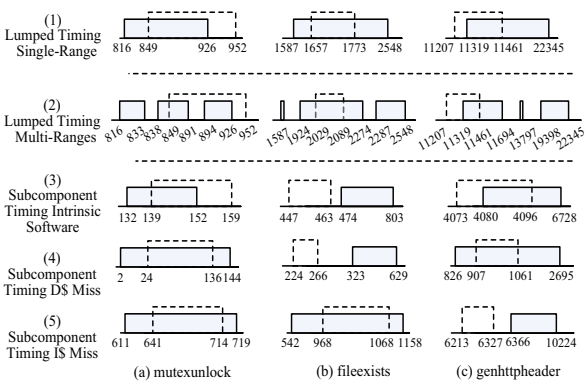

Figure 1. Comparison of normal timing bound (solid line) and a fuzz malware timing range (dashed line) for LTSR, LTMR, and ST models. The ST model includes separate timing bounds for each subcomponent. *Time values are in clock cycles. Not to scale*.

B. Lumped Timing Multi-Range (LTMR) Model

Compared to the LTSR model, the lumped timing multi-range (LTMR) model divides the normal single range into multiple ranges, which represents timing behavior at a finer granularity. As shown in of Figure 1(2), using the LTMR model can detect malware timing that falls outside the bound of each range (e.g., malware timing values ranging from 891 to 894 cycles in Figure 1(2)(a)), which shows the potential to increase detection rate with multi-ranges.

The finer timing detection of multiple ranges comes at the expense of more complex training and higher hardware costs. The number of ranges for an event depends on the path an event belongs to, arguments passed to the event, and influences from other events. Several approaches can be used to determine these normal timing ranges and train the timing models. One approach is to execute the application under different scenarios (e.g., different data inputs) that define different ranges. An alternative approach is to analyze all data post-collection by statistically dividing all lumped timing into ranges.

A larger number of normal timing ranges provides finer detection granularity but would result in longer training time and more storage overhead at runtime. We currently consider a LTMR model with three timing ranges, which allows for a direct comparison with the subcomponent timing model that is composed of three timing subcomponents.

C. Subcomponent Timing (ST) Model

The timing of specific operations is affected by the underlying system architecture, operating system, and execution environment, which can lead to unpredictable timing behaviors (e.g., cache behaviors). For example, the execution time of a function call is influenced by the instructions generated during compilation, processor frequency, pipeline structure, cache/memory access delays, interrupt execution, context switches, etc. Therefore, the timing of events can vary widely, such that detecting malicious execution may be difficult. Consider the timing variability introduced by an interrupt executing during another event. The interrupt adds a time delay to perform a context switch to the ISR, execute the routine, and perform a context switch back to the original task. In a lumped timing model, the execution time of the ISR will be lumped into the WCET bound. Nevertheless, the execution of a malicious operation may only require a small amount of code that has a comparatively low time delay. Thus, the prolonged WCET may make the anomaly detection insensitive to the malware timing.

Fortunately, the information from the microprocessor's trace port can be utilized to analyze the low-level execution behavior of system operations to separate the timing into several subcomponents. We define two classes of timing, namely *intrinsic timing* and *incidental timing*. *Intrinsic timing*

is the timing intrinsic to the execution of specific software and its data inputs, in the absence of delays or interference from the system architecture, OS, or other tasks. In other words, the intrinsic timing is the ideal software execution time, which is relatively stable. *Incidental timing* is the timing due to the execution environment in which the software is executed, and incorporates several subcomponents. Within the current approach, incidental timing subcomponents include I$ misses and D$ misses. These subcomponents can detect malware by identifying changes in the temporal and spatial characteristics of instruction and data addresses. For example, an information leakage malware that increases the number of writes to data memory may evict data stored in the D$, thereby leading to increased data cache misses for other operations.

To determine accurate timing bounds for intrinsic and incidental subcomponents, the interference from the execution of interrupts and other tasks must be isolated. The frequency of interrupts can be sporadic, and failing to isolate the interference would lead to pessimistic WCETs, which in turn makes detecting malware more difficult. Once isolated, the intrinsic and incidental timing subcomponents represent the tightest possible bounds on the execution time of software operations. The resulting subcomponent timing (ST) model is more accurate, more sensitive to malware execution, and provides greater malware detection ability.

Figure 1(3)(4)(5) presents the normal subcomponent timing ranges in solid lines (with single normal range for each subcomponent) and the malware timing ranges for three software events in dashed lines. Using the ST model, the *mutexunlock*'s normal intrinsic timing only has a 65% overlap with the malware's intrinsic timing, while the LTSR and LTMR models yield a 75% and 72% overlap, respectively. Although the LTSR and LTMR models are able to detect the malware, the smaller timing overlap in the ST model's intrinsic timing leads to a higher detection rate. For many events, the timing variability is often attributed to a single subcomponent, such as the D$ miss subcomponent for *fileexists* and the I$ miss subcomponent for *genhttpheader* in Figure 1(b)(c).

The advantage of ST model is reduced timing variability, as the timing variability of each subcomponent is isolated. This in turn makes mimicking timing behavior of an event harder and increases detection accuracy. However, the tradeoff is a more complex data collection process.

V. Non-intrusive Detector Design

Hardware-assisted malware detection requires no additional software code to specify or detect the occurrence of events. Instead, a hardware detector can interface with the trace port of the processor to analyze trace signals and detect both the occurrence and timing of events. Processor trace ports are common interfaces provided by processor manufacturers, and widely used within systems-on-a-chip (SOCs) [6][7]. Interfacing to the trace port enables the runtime detector to be non-intrusive, which in turn does not affect the execution of the application (i.e., zero performance overhead). The proposed approach accesses the processor trace port on-chip, and doesn't modify or require the external interface, thus ensuring security of the trace interface. Additionally, the hardware detector is used both at runtime to detect malware and at design time to automatically collect timing to train the normal model.

A. Non-intrusive Malware Detection based on LTSR Model

To create the LTSR model, the lumped execution time is measured from when the processor fetches the monitored event's first instruction until the processor fetches the same event's last instruction. This measurement process should not affect the application's execution timing behavior. Therefore, we designed a hardware module, *FindEventID*, to monitor the processor's trace port and detect the execution of events by observing the program counter (PC). As shown in Figure 2 (a) and (c), when the PC from the processor's trace port matches a monitored event's start address, an associated timer is activated. When the PC matches the event's end address, the timer is stopped. The timer's value thus represents lumped execution time (in cycles) of the event. This lumped execution time is compared to the current minimum and maximum timing value to determine if these timing bounds must be updated. After executing the normal system for all possible execution scenarios and for a sufficient duration for each scenario, the BCET and WCET of all monitored events can be achieved and read from the hardware. These bounds will be configured into the hardware detector for runtime detection. In the experimental system, the operating frequency of the hardware detector is the same as embedded processor. However, hardware FIFOs can be used to interface to a higher frequency processor while running the detector at a lower frequency [10].

For the LTSR model, the training process has been embedded within data measurement process. The interface in Figure 2(a) and *FindEventID* module in Figure 2(c) are also used in the runtime detector. The detector consists of registers to hold each event's start address, end address, and [BCET, WCET]s. A timer to measure each event's total elapsed cycles is enabled and disabled according to the event's start and end addresses, similar to the data measurement phase. The LTSR detects anomalous timing by detecting when that time is either less than the BCET or greater than the WCET. In the former case, as soon as the end address is observed, the timing is compared to the BCET, and the anomalous execution is immediately detected. In the latter case, the anomalous execution is detected when the elapsed time exceeds the WCET.

B. Non-intrusive Malware Detection based on LTMR Model

Post-collection analysis is used to create the LTMR model for the following reasons: 1) achieving multi-range timing by separately measuring timing under all different paths is infeasible, 2) timing values under different scenarios may overlap, and 3) separating timing ranges during data measurement complicates the hardware design. Thus, we use a similar data collection process for the LTMR as used for the LTSR model. After collecting all timing data, the timing values are clustered into multiple ranges. Instead of saving one BCET and WCET during data collection, all training data needs to be collected, such that multiple bounds can be determined.

The multi-range clustering can be done using statistical or machine learning algorithms. Here, hierarchical clustering [5] is used to automatically cluster the lumped timing into multiple ranges by measuring the distance (e.g., average Euclidean distance) between each timing sample. Although using

clustering algorithms for one-dimensional data may be overkill, we do not adopt kernel density estimation [2] or other algorithms because: 1) the distance-based method contributes to detection since closer timing values share similar attributes, 2) hierarchical clustering makes full use of the data, 3) hierarchical clustering can automatically control the resulting number of clusters, which ensures the multi-range model is comparable to other models, and 4) alternative algorithms (e.g., KNN, K-means) do not result in much difference in the resulting ranges. One can use other statistical methods for range division, but optimization of separating data under different paths is beyond this paper's scope.

The training process and hierarchical clustering is performed offline using MATLAB's machine learning toolbox [18] after the timing measurement. The three basic steps in hierarchical clustering are: 1) find data samples closest in terms of distance, 2) group objects into hierarchical cluster tree, and 3) cut the hierarchical tree into the desired number of clusters.

The detector interface and structure for the LTMR model (Figure 2(b) and (c)) has the same structure as the LTSR model but requires storing and verifying three bounds for each event.

C. Non-intrusive Malware Detection based on ST Model

Figure 2(d) presents the interface to a MicroBlaze processor trace port and the structure of the hardware detector for the ST model. The detector interface uses 10 trace port signals: *PC* is the current program counter; *Valid* is a one bit signal indicating in which cycles the *PC* is valid; *ExptTaken* is a one bit signal indicating if an exception occurs while executing the current *PC*; *ExptType* is a 5-bit signal used to detect interrupts; *I$Req* and *D$Req* are one-bit signals indicating if the instruction/data address is within the instruction/data range; *I$Hit* and *D$Hit* are one-bit signals indicating if instruction/data address is present in the instruction/data cache; *I$Rdy* and *D$Rdy* are one-bit signals indicating if the instruction/data access is completed. The *Subcomponent Separation Analysis* module monitors trace signals, maps PC addresses to event IDs, and measures the subcomponent timing of each event. For each monitored event, the hardware detector stores the [BCET, WCET] bounds for intrinsic timing, D$ miss timing, and I$ miss timing. If the timing for any of the three subcomponents is violated at runtime, the hardware detector asserts a non-maskable interrupt to the processor indicating the presence of malware.

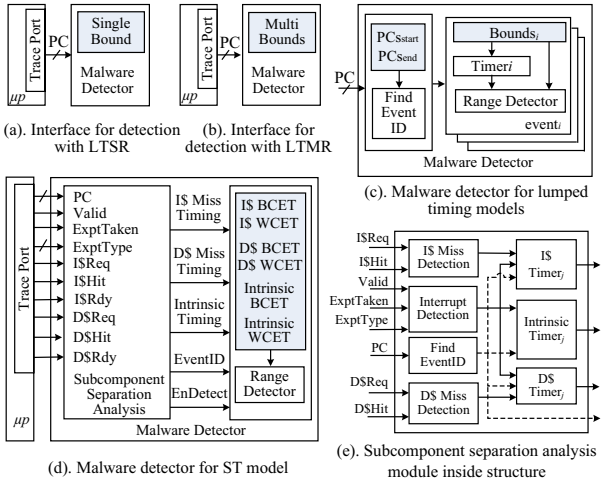

Figure 2. Hardware-assisted malware detector interface using (a) LTSR, (b) LTMR, (c) hardware detector for lumped timing models, (d) interface and hardware detector for ST model, and high-level overview of (e) subcomponent timing analysis module showing for one event. *Note: Sequence-based detection is embedded but not shown.*

Figure 2(e) presents an overview of the internal architecture for the *Subcomponent Separation Analysis* module used in both training and detection phases. Each monitored event has 3 timers to record the subcomponent timing. *I$Timer* and *D$Timer* are enabled when I$ or D$ misses are detected using the trace signals, respectively. *I$Timer* and *D$Timer* are stopped when the instruction or data access has completed. When either *I$Timer* or *D$Timer* is enabled, the *IntrinsicTimer* is disabled, thereby separating the timing subcomponents. During the execution of an interrupt, while the ISR is executing, all timers for the interrupted events are disabled.

Algorithm 1 presents the pseudocode for the ST model detection. *EnDetect* is a master enable signal for all timers associated with each event, and its j^{th} bit is set to 1 when event$_j$ is detected. *EnInt* indicates if the execution of an event has been interrupted. An I$/D$ miss is detected when the corresponding *$Req* is 1 but the *$Hit* is 0, in which case the *$timers* are

Algorithm 1: Detection Algorithm with the ST Model

Input: Trace Signals, StartEvent, EndEvent, I$BCET, D$BCET, IntriBCET, I$OWCET, D$OWCET, IntriOWCET
Output: Anomalous event index *j*

```
1. j = FindEventId(PC);
2. if ( PC == StartEvent[j] ) then
3.     EnDetect[j] = true;            IntriTimer[j] = -IntriBCET [j];
4.     I$Timer[j] = -I$BCET [j];   D$Timer[j] = -D$BCET [j];
5. if ( PC == EndEvent[j] ) then
6.     EnDetect[j] = false;
7.     if ( I$Timer[j] > 0 or D$Timer[j] > 0 or IntriTimer[j] > 0 ) then
8.         return j;
9. if ( EnDetect[j] ) then
10.    if ( I$Req and not I$Hit )   then EnI$[j] = true;
11.    elseif ( I$Rdy )             then EnI$[j] = false;
12.    if ( D$Req and not D$Hit ) then EnD$[j] = true;
13.    elseif ( D$Rdy )             then EnD$[j] = false;
14.    if ( ExptTaken and ExptType == 01010 )   then EnInt[j]=true;
15.    elseif ( PC == ISRReturn )                then EnInt[j] = false;
16.    if ( not EnInt[j] and not EnI$[j] and not EnD$[j] ) then
17.        IntriTimer[j]++;
18.    if ( not EnInt[j] and EnI$[j] ) then
19.        I$Timer[j]++;
20.    if ( not EnInt[j] and EnD$[j] ) then
21.        D$Timer[j]++;
22.    if ( I$Timer[j] > I$OWCET[j] or D$Timer[j] > D$OWCET[j] or
          IntriTimer[j] > IntriOWCET[j] ) then
23.        return j;
```

Table 1. Hardware components required for storage and detection at runtime for increasing number of monitored events.

# of Events	Model	Hardware Components						
		Timer	*Addr. register*	*Timing bounds register*	*Seq. detect. register*	*Comp (==)*	*Comp (<)*	*Comp (>)*
10	LTSR	10	20	20	15	10	10	0
	LTMR	10	20	60	15	30	30	20
	ST	30	20	60	15	30	30	0
30	LTSR	30	60	60	35	30	30	0
	LTMR	30	60	180	35	90	90	60
	ST	90	60	180	35	90	90	0

enabled, but only when the $ miss happens outside of an interrupt handler. Once the instruction or data is ready, the corresponding timers are disabled. If the execution is neither interrupted nor waiting on a cache miss, the *IntrinsicTimer* is enabled. To simplify the hardware, the timers use an offset timing in which the timers are initialized with -BCET. This strategy enables the BCET bound to be verified by checking the timer's most significant bit, rather than using a comparator. The offset timing requires storing the WCET as an offset WCET (OWCET), calculated as $WCET_j$ - $BCET_j$. The WCET bound is verified by checking if the timer exceeds an event's OWCET.

D. Hardware Detectors Overhead Analysis

The detectors for all three models have been implemented in hardware and integrated within the smart connected pacemaker prototype (Section VI). Table 1 presents the number of hardware components required to support the LTSR, LTMR, and ST models. Within the hardware, each event's start and end addresses are stored using 24-bit registers, as 24 bits is the number of bits needed to address the application's code segment. Note that further reduction in register size is possible, such as identifying the least number of bits to differentiate addresses for all monitored events.

For sequence detection, the hardware requires registers to store information defining the expected event sequence, which includes one register for each event. Additionally, the hardware has five registers used for dynamic detection of execution sequence violations. The size of all sequence detection registers in bits is equal to the number of events.

For timing measurement and analysis, the bounds of single-range or multi-range models are stored in 26-bit registers, which are sufficient for the largest timing values across all monitored events. The *FindEventID* module requires equality comparators to match events' addresses with the current PC. The *RangeDetector* module requires magnitude comparators (i.e., less than) to verify upper bounds. For the LTSR and ST models, only a single logic gate is required to verify the most significant bit. However, because the LTMR model has only one timer for lumped timing, this optimization can only be applied to the first range's lower bound verification.

As the number of monitored events increases, the required hardware components increase linearly as shown in Table 1. LTMR and ST have the same rate of increase for 26-bit registers and comparators, which increase 3X faster than LTSR, while all three detection models need the same number of 24-bit registers. ST requires 3X the number of timers as LTSR and LTMR. The LUTs and FFs required for the detecting all events in pacemaker application is less than 1% of the pacemaker prototype's total area, and dynamic power consumption is less than 2% of the overall system power consumption.

VI. Experimental Evaluation

A. Smart Connected Pacemaker Benchmark

To evaluate the timing based anomaly detection, we developed an FPGA-based prototype for a smart connected pacemaker using a Xilinx Spartan-6 XL45 FPGA, presented in Figure 3. The smart connected pacemaker prototype enables the implementation and analysis of different vulnerabilities and

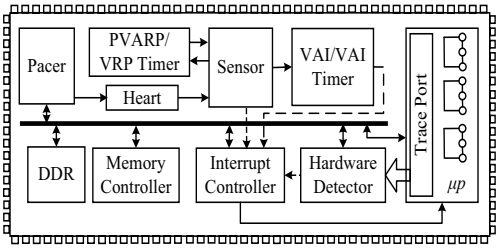

Figure 3. Smart connected pacemaker prototype system.

malware, and includes a simulated patient heart, a cardiac sensor, an impulse pacer, and four timers. The simulated patient heart generates irregular beats and reacts to the impulse pacer signal controlled by the pacemaker software. The cardiac sensor interfaces to the simulated heart model and sends the measured heart signals to the microprocessor using interrupts. The output from the cardiac sensor also controls the Atrio-Ventricular Interval (AVI) timer and the Ventriculo-Atrial Interval (VAI) timer. The VAI/AVI timers are used to maintain the appropriate delay between the atrial/ventricular activation and the ventricular/atrial activation, and will generate an interrupt if the AVI/VAI exceeds a specific interval configured by the cardiac physician. The PVARP/VRP timers filter noise in the ventricular and atrial channels, respectively [4][17].

The pacemaker software, which executes on a MicroBlaze processor, consists of three tasks and four ISRs. The ISRs interact with the pacemaker's cardiac sensor and timers, and have the highest priority. ISR operations include atrial and ventricular pacing, and recording ventricular and atrial activity. The calculation task calculates the Upper Rate Interval (URI) and records cardiac activity to a log file. A fault-exam task analyzes the cardiac activity and detects a high URI, which indicates the pacemaker cannot pace the heart correctly or that pacemaker's cardiac sensor has malfunctioned. In the event of a high URI, the pacemaker immediate transmits a message to alert the physician. The communication task is responsible for communication, by which the physician can configure the pacemaker's settings or a home monitoring device to access daily logs of the cardiac activity. Including the events in all tasks and ISRs, there are 45 events in total that are monitored.

B. Malware Implementation

We utilized four mimicry malware targeting the pacemaker. The file manipulation malware manipulates the cardiac activity log to deceive the physician, with the intent of leading a physician to an incorrect diagnosis or a potentially life-threatening misconfiguration. This malware involves reading the cardiac activity log file, manipulating the data, and writing the modified data back to the log file. The file manipulation malware affects the communication task's execution and mimics the software's execution sequence to avoid sequence-based anomaly detection. The information leakage malware covertly reads data in the cardiac activity log of the patient within the calculation task, and sends this information to a malicious data center within the fault-exam task. The information leakage malware also increases the execution frequency of the fault-exam task in order to rapidly leak a large amount of data. The fuzz attack malware [22] manipulates the system execution by randomizing data values and

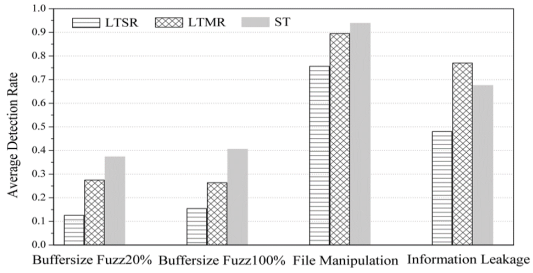

Figure 4. Average detection rate across all events for all malware.

system/function call arguments, which is usually implemented by interpolating data in memory. Two variants of the fuzz malware are considered, one where the fuzz malware randomizes the log buffer size by up to 20% and one where the fuzz malware randomizes the same buffer size up to 100%. The fuzz malware indirectly affects the system timing, and impact the system's time-sensitive control operations, which is a common intent of fuzz attacks. We analyzed the fuzz attack to evaluate the effectiveness of timing models to detect such attacks, which are one of the hardest mimicry malware to detect.

C. Experimental Results

To characterize the normal system execution for the smart connected pacemaker, we combined system-level timing requirements with experimental training of timing bounds for the monitored events. To train each timing model, we executed the system 1000 times under all execution scenarios (i.e., healthy/unhealthy patient, different physician configurations) to measure the lumped and subcomponent timing bounds of each event. 1000 samples were sufficient for our system to achieve a low false positive rate. However, the collection and model building processes are automated and facilitated by the hardware, so training can be efficiently scaled to larger sizes. After creating the timing models, we configured each model in the hardware detector. To evaluate the detection rate, we executed each malware 100 times within the pacemaker prototype. We further evaluated the false positive rates using cross-validation. Specifically, 1000 timing values were randomized, and for each slice of 100, the false positive rate was calculated using the remaining 900 as the training set, and averaged across these 10 slices.

Figure 4 presents the average detection rate across all monitored events for the four malware considered. Overall, detection with the LTMR model outperforms the LTSR model, achieving on average a 17% higher and at best a 24% higher detection rate than the LTSR model. Detection with the ST model achieves the highest detection rate for fuzz attacks and file manipulation malware, achieving a maximum detection rate 0.94. For the information leakage malware, the LTMR model achieves the highest detection rate, which is primarily due to a 1.00 detection rate for the event *xemacif*. The LTSR and ST models cannot detect that event, which results in the lower average detection rate for this malware. On the other hand, the LTMR model has the highest average and worst case false positives, as shown in Table 2, which is 0.3% and 0.2% higher than LTSR and ST on average, respectively.

Figure 5 presents the single event detection rate and the cumulative detection rate for the fuzz buffer size 20% malware,

Table 2. Average and worst case false positive rate for all models.

Average False Positive Rate			**Worst-case False Positive Rate**		
LTSR	LTMR	ST	LTSR	LTMR	ST
0.15%	0.46%	0.27%	0.2%	0.7%	0.6%

which affects the calculation and communication tasks. The single event detection rate demonstrates the detection performance of each model separately for each event. For most events, the ST model achieves a higher single event detection rate than LTSR and LTMR. For some events, such as *filesleek*, the LTSR model is unable to detect the malware, whereas the LTMR and ST have a detection rate of 1.00. This shows that multi-ranges and subcomponent timing both have an advantage in detection compared to single-range detection. One exception is event *tcpwrite*, for which the LTSR model achieves a higher detection rate. This is because the interference of interrupts is included in the LTSR model but excluded in the ST model. However, in most cases, the interference of interrupts negatively affects the detection performance.

The LTMR model's performance is bifurcated, having either a low detection rate or a high detection rate for different events. Specifically, five events have detection rates greater

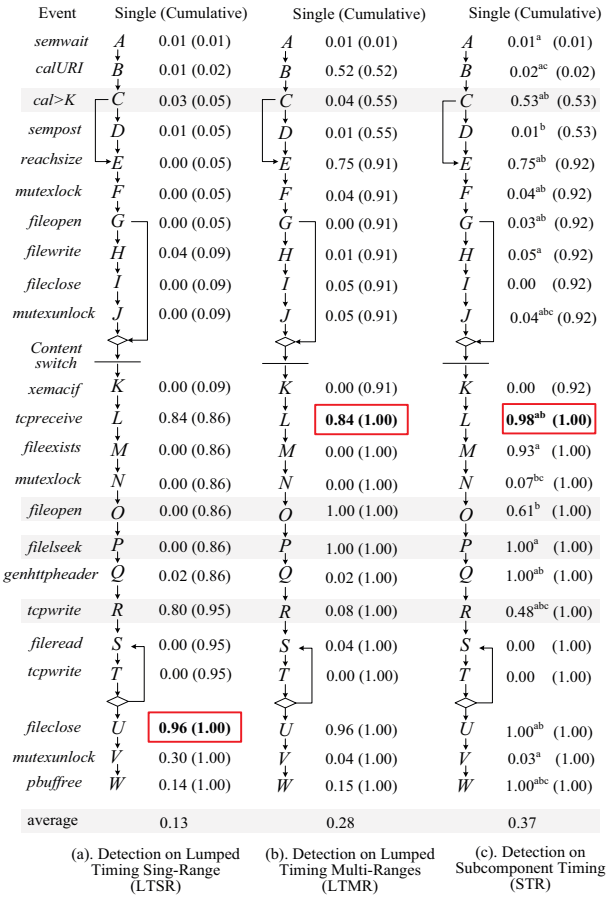

Figure 5. Single event and cumulative detection rate (in parentheses) for fuzz buffer size 20% malware using (a) LTSR, (b) LTMR, and (c) ST models. Superscript labels indicate if malware was detected by [a]intrinsic, [b]instruction cache, or [c]data cache timing. Letters indicates monitored events, and arrows indicate control flow between events.

than 0.75, and 17 events have detection rates less than 0.15. For example, the detection rate of LTMR for event *cal>K* is as low as 0.01, while the ST model's detection rate is 0.53. The reverse situation occurs for event *fileopen*, for which the LTMR model achieves a 1.00 detection rate, whereas the ST model has a 0.61 detection rate. The reason is that the malware's lumped timing happens to be between two normal lumped timing ranges.

The cumulative detection rate is the overall rate of detecting malware within the execution sequence of events. When a cumulative detection rate of 1.00 is reached, it indicates that the malware's execution is detected for all malware executions. Using the LTSR model, the cumulative detection rate reaches 1.00 at the 21st event. But, using the LTMR and ST models, the cumulative detection rate reaches 1.00 at the 12th event. In addition, the LTMR and ST models achieve a cumulative detection rate of 0.91 and 0.92, respectively, at just the 5th event. For the ST model, Figure 5 is annotated to indicate which subcomponent(s) detected the malicious execution. The intrinsic timing and I$ miss timing contribute the most to the detection rate and can detect malware for 13 events, whereas the D$ timing detects malware for only 5 events. This behavior is expected as the intrinsic timing and I$ timing have less variability, and thus tighter timing bounds.

In conclusion, the ST model has both the best single event and best cumulative detection accuracy, with only 0.12% increase in false positives. Detection with the LTMR model outperforms the LTSR model, and has better detection than the ST model in a few isolated cases, but overall has lower average and cumulative detection rates and a higher false positive rate.

VII. Conclusions and Future Work

We presented a non-intrusive malware detection approach that uses information available from a processor's trace port to separate the execution time into intrinsic timing, I$ miss timing, and D$ miss timing, while eliminating the effects of interference from interrupts and context switches. Experiments with a smart connected pacemaker and four mimicry malware demonstrate that the ST model achieves an average per event detection rate of 0.66, which is 0.24 higher than the LTSR model. In addition, the ST model detects malware faster, reaching a 0.92 detection rate after 5 events and 100% detection rate after only 12 events, which is a 52% fewer events than the lumped timing models. Future work includes further analysis of the ST model using statistical analysis (e.g., cumulative distribution functions) within sliding execution, employing machine learning methods (e.g., support vector machines), and analyzing the tradeoffs in detection rate, false positive rates, hardware area requirements, and energy consumption.

References

[1]. Bhatkar, S., A. Chaturvedi, R., Sekar. Dataflow Anomaly Detection. Symposium on Security and Privacy, pp. 15-62, 2006.

[2]. Botev, Z.I., J.F. Grotowski, and D.P. Kroese. Kernel density estimation via diffusion. Annals of Statistics Vol. 38, No. 5, pp. 2916–2957, 2010.

[3]. Chandola, V., A. Banerjee, V. Kumar. Anomaly Detection: A Survey. ACM Computing Survey, 41(3), 2009.

[4]. Jiang, Z., M. Pajic, S. Moarref, R. Alur, R. Mangharam. Modeling and Verification of a Dual Chamber Implantable Pacemaker. International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 188–203, 2012.

[5]. Kaufman, L., Rousseeuw, P. J. Finding Groups in Data: An Introduction to Cluster Analysis. John Wiley, New York, 1990.

[6]. Lee Y., J. Lee, I. Heo, D. Hwang, Y. Paek, Integration of ROP/JOP monitoring IPs in an ARM-based SoC, Conference on Design, Automation & Test in Europe, March 14-18, 2016.

[7]. Lu, S., M. Seo, R. Lysecky. Timing-based Anomaly Detection in Embedded Systems. Asia South Pacific Design Automation Conference, pp. 809-814, 2015.

[8]. Lu. S., and R. Lysecky. Analysis of Control Flow Events for Timing-based Runtime Anomaly Detection. Workshop on Embedded Systems Security, 2015.

[9]. McAfee Labs. Threats Report 2015. http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2015.pdf.

[10]. Mu J., K. Shankar, R. Lysecky. Profiling and Online System-Level Performance and Power Estimation for Dynamically Adaptable Embedded Systems. ACM Transactions on Embedded Computing Systems (TECS), Vol. 12, No. 3, Article 85, pp. 1-20, 2013.

[11]. Mohan, S., S. Bak, E. Betti, H. Yun, L. Sha, M. Caccamo. S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. ACM Conference on High Confidence Networked Systems, 2013.

[12]. Patel K., S. Parameswaran. SHIELD: A Software Hardware Design Methodology for Security and Reliability of MPSOCs. Design Automation Conference, pp. 858-861, 2008.

[13]. Patel, K., S. Parameswaran, R. Ragel. Architectural Frameworks for Security and Reliability of MPSOCs. IEEE Transactions on Very Large Scale Integration Systems, No. 99, pp. 1–14, 2010.

[14]. Rahmatian, M., H. Kooti, I. Harris, and E. Bozorgzadeh. Hardware-Assisted Detection of Malicious Software in Embedded Systems. IEEE Embedded Systems Letters (ESL), Vol.4, No.4, pp.94-97, 2012.

[15]. Ramilli, M. Bologna, M. Prandini. Always the Same, Never the Same. IEEE Security & Privacy, Vol. 8, No. 2, pp. 73-75, 2012.

[16]. Sharif, M.I., K. Singh, J. T. Giffin, W. Lee. Understanding Precision in Host based Intrusion Detection. International Symp. on Research in Attacks, Intrusions and Defenses. Vol.4637, pp.21-41, 2007.

[17]. Singh, N.K., A.J. Wellings, A.L.C. Cavalcanti. The Cardiac Pacemaker Case Study and its Implementation in Safety-Critical Java and Ravenscar Ada. International Workshop on Java Technologies for Real-time and Embedded Systems, 2012.

[18]. Statistics and Machine Learning Toolbox User's Guide. https://www.mathworks.com/help/pdf_doc/stats/stats.pdf

[19]. Symantec. Internet Security Threat Report. https://www.secure128.com/download/istr-21-2016-en.pdf, 2016.

[20]. Verizon. State of the Market: Internet of Things. https://www.verizon.com/about/sites/default/files/state-of-the-internet-of-things-market-report-2016.pdf, 2016.

[21]. Wagner, D., P. Soto. Mimicry Attacks on Host based Intrusion Detection Systems. ACM Conference on Computer and Communications Security, pp. 255-264, 2002.

[22]. Wasicek A., P. Derler, E. A. Lee, Aspect-oriented Modeling of Attacks in Automotive Cyber-Physical Systems, Annual Design Automation Conference, p.1-6, 2014.

[23]. Wilhelm, R., J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, P. Stenstrom. The Worst-Case Execution-Time Problem-Overview of Methods and Survey of Tools. ACM Transactions on Embedded Computing Systems, Vol.7, No.36, pp.1-47, 2008.

[24]. Yoon M.-K., S. Mohan, J. Choi, J.-E. Kim, L. Sha. SecureCore: A Multicore-based Intrusion Detection Architecture for Real-Time Embedded Systems. Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013.

[25]. Zhang, T., X. Zhuang, S. Pande, W. Lee. Anomalous Path Detection with Hardware Support. Conference on Compilers Architectures and Synthesis for Embedded Systems, pp.43-54, 2005.

[26]. Zimmer, C., B. Bhat, F. Mueller, S. Mohan. Time-Based Intrusion Detection in Cyber-Physical Systems. ACM/IEEE International Conference on Cyber-Physical Systems, pp. 109-118, 2010.