Time and Sequence Integrated Runtime Anomaly Detection for Embedded Systems

Article *in* ACM Transactions on Embedded Computing Systems · December 2017

DOI: 10.1145/3122785

CITATIONS
13

READS
468

2 authors, including:

Sixing Lu
The University of Arizona
13 PUBLICATIONS **82** CITATIONS

SEE PROFILE

Time and Sequence Integrated Runtime Anomaly Detection for Embedded Systems

SIXING LU and ROMAN LYSECKY, University of Arizona

Network-connected embedded systems grow on a large scale as a critical part of Internet of Things, and these systems are under the risk of increasing malware. Anomaly-based detection methods can detect malware in embedded systems effectively and provide the advantage of detecting zero-day exploits relative to signature-based detection methods, but existing approaches incur significant performance overheads and are susceptible to mimicry attacks. In this article, we present a formal runtime security model that defines the normal system behavior including execution sequence and execution timing. The anomaly detection method in this article utilizes on-chip hardware to non-intrusively monitor system execution through trace port of the processor and detect malicious activity at runtime. We further analyze the properties of the timing distribution for control flow events, and select subset of monitoring targets by three selection metrics to meet hardware constraint. The designed detection method is evaluated by a network-connected pacemaker benchmark prototyped in FPGA and simulated in SystemC, with several mimicry attacks implemented at different levels. The resulting detection rate and false positive rate considering constraints on the number of monitored events supported in the on-chip hardware demonstrate good performance of our approach.

CCS Concepts: • **Security and privacy → Intrusion/anomaly detection and malware mitigation**; • **Computer systems organization → Embedded and cyber-physical systems**;

Additional Key Words and Phrases: Embedded system security, anomaly detection, software security, timing based detection, medical device security

ACM Reference format:
Sixing Lu and Roman Lysecky. 2017. Time and Sequence Integrated Runtime Anomaly Detection for Embedded Systems. *ACM Trans. Embed. Comput. Syst.* 17, 2, Article 38 (December 2017), 27 pages.
https://doi.org/10.1145/3122785

1 INTRODUCTION

The prevalence and pervasiveness of embedded systems with network access is rapidly increasing. The push towards the Internet of Things will see a dramatic increase in the number of online embedded devices, expected to reach 30 billion devices by 2020. At the same time, malicious software is growing at an alarming rate, with estimates that 100,000 new malware are created every day [21]. As embedded systems continue to pervade everyday life, they will increasingly become targets for malicious attacks. Recent examples of malware and potential malicious attacks include decrypting SD card, injecting a fatal dose of insulin in an insulin pump [19], and disabling an automobile's brake system [22].

This research was partially supported by the National Science Foundation under Grant CNS-1615890.
Authors' address: S. Lu and R. Lysecky, Department of Electrical and Computer Engineering, University of Arizona, 1230 E Speedway Blvd, Tucson, AZ 85721; emails: sixinglu@email.arizona.edu, rlysecky@ece.arizona.edu.

https://doi.org/10.1145/3122785

Historically, embedded systems were not dramatically affected by malware because the systems lacked network connections and were "secured" by their physical locations [10]. For example, automotive electronics were secured within a locked compartment of the automobile. With the increase of network connectivity, this physical isolation no longer provides security for these embedded devices. Embedded systems are often critical components between the cyber and physical worlds, and have direct implications on the system safety, particularly for life-critical systems such as medical devices and automobiles. Moreover, embedded systems collect and store private information, which is often the target of malware. At the same time, protecting embedded systems is of critical challenges: (1) tight constraints on power, cost, area, performance, reliability, and the like often limit the complexity of security methods, (2) software updates may be infrequent (e.g., medical devices may require re-certification before deployment, and (3) users are less aware of security of embedded devices. Therefore, it is critical to devise new security models and detection methods for protecting network-connected embedded systems.

Multiple proactive approaches exist for securing embedded systems from malware, seeking to ensure embedded hardware and software are free from vulnerabilities before deployment, by designing secure protocols that limit remote access and by eliminating vulnerabilities that may be exploited through offline testing. However, given the increasing complexity of embedded systems, designing a system free of vulnerabilities is prohibitive. So there is also a critical need for reactive approaches that detect the presence of malware in deployed systems.

Anomaly-based detection methods define a system model that represents the correct system execution. Malicious activities can be discovered by detecting when the system execution does not match the specified system model. For example, execution sequences for correct execution behaviors can be represented with finite state automata [13]. At runtime, any transition between states that does not adhere to the finite state automata is reported as a malicious activity. Alternatively, execution sequences can be modeled as control-flow graphs [30] or function call graphs [2] that enumerate the possible sequences within the application software.

Although anomaly-based detection has the advantage of being able to detect zero-day exploits, as opposed to signature-based detection methods [29], existing approaches are susceptible to mimicry attacks [34]. Mimicry attacks are countermeasures to anomaly-based detection, which seek to avoid detection by constructing malware that mimics the correct system execution. By interleaving normal operations with malicious operations, mimicry attacks can match the correct system execution model. A finer monitoring granularity of execution behavior can improve the robustness against mimicry attacks. Several approaches have explored the granularity at which the system behavior is monitored, ranging from system calls, basic blocks down to instructions [8]. However, monitoring all possible paths within the control flow at a fine granularity (e.g., basic block) is infeasible due to the complexity of applications and the resource constraints of embedded systems.

Timing requirements offer a unique opportunity to strengthen embedded system security by detecting anomalies in the timing of software elements (e.g., tasks, functions, loops) in addition to detecting anomalies in the execution sequence. Timing of software elements is dependent on several factors including the instructions generated during compilation, cycles per instruction of the processor architecture, frequency of the processor, cache hierarchy, and so on. The potential benefit of timing-based detection is increased robustness to mimicry attacks, because mimicking timing is more difficult than mimicking the execution sequence alone. Embedded systems are often sensitive to changes in timing, such that even small differences in execution at one point may lead to noticeable (and detectable) perturbation at another point within the system execution.

In this article, we present a runtime anomaly detection (RAD) methodology and non-intrusive hardware design that seeks to systematically utilize system-level timing constraints to detect

malware, providing detection that is more robust with increased resilience to mimicry attacks. This approach utilizes a formal runtime security model [17] that defines the normal system behavior, which is comprised of arbitrary system events, event dependencies, and timing requirements between events, including best-case execution time (BCET) and worst-case execution time (WCET). The RAD hardware builds upon the proof-of-concept implementation [16] and detects both changes in the execution sequences and changes in timing within those sequences to find malicious activity, all without perturbing the system execution thus introduce zero performance overhead. This article further optimizes execution sequence and timing detection algorithms within RAD, and evaluates monitoring events selection methods based on timing distribution for mimicry malwares including *File Manipulation, Control Loop Delay, Information Leakage, Interrupt Disable*, and *Denial of Service* (*DoS*) attack and *Fuzz Attack*, given limited area of RAD hardware. Based on the network-connected pacemaker benchmark, this article creates a SystemC simulation to compare the detection rate and false positive with SecureCore approach.

2 RELATED WORK

2.1 Anomaly-Based Detection

Anomaly-based detection methods monitor the system execution to detect deviations from preestablished specification of normal behavior. Several approaches have been developed to specify normal behavior and detect deviation from that specified behavior. System calls are the interface between an application and operating system, and are one of the most common monitoring objects in anomaly-based detection. System calls are derived from execution path and can be abstracted as execution sequence either statically [3] or dynamically [12]. The execution sequence is typically represented using finite state automata, which is used to detect anomalies if there is no transition from the current state to one of the expected next states. Although system calls are relatively easy to monitor, granularity of probing system call sequences is large, so mimicry attacks are often successful in masquerading their behavior to evade detection.

Other researches have demonstrated that modeling the control flow of software can achieve more precise detection compared to modeling system calls. Zhang et al. [37] proposed a method that monitors control flow sequences within a sliding window of N jumps. Mimicking the behavior of correct execution becomes more difficult as the granularity of monitoring the control flow becomes finer. However, monitoring all possible paths within the control flow is infeasible due to the complexity of software applications. Mao and Wolf [18] compared the effect and cost of storing different types of control flow information, including return addresses, instruction opcodes, and custom hash functions. While a customized hash function can reduce memory requirements, determining a hash function without collision is a challenge. In addition, once the hash function is known, the detection method becomes susceptible to mimicry attacks as with other approaches.

Researches have also focused on detecting malicious activity by monitoring the dataflow of an application [7]. The dataflow of a software application captures the data dependencies between operations, including the value or range of variables and arguments passed to function calls. For example, the length of an argument to a function can be statistically modeled using Markov chains [11], such that argument lengths outside the stochastic interval indicate probable malicious activity. Dataflow can also be modeled as the relationship between arguments for different control flow operations [4]. Similarly, arguments to function calls also belong to an application's dataflow and can be modeled to detect semantic attacks and mimicry attacks [5]. Modeling dataflow can detect malware related to memory data changes, but the normal system model would be very complicated, and data may not be accessible at runtime.

In addition to approaches that monitor application execution, some research monitors other system behaviors to detect anomalous execution. Memory Heat Map [35] creates a heat map representing the accesses times for individual memory regions, which reflects system activity patterns. Change in the execution from expected patterns is then used to detect malware. Liu et al. [15] designed a hardware-based component that detects stack buffer overflows by monitoring microarchitecture features, specifically monitoring branch mis-predictions for return addresses in the stack. Malware detection based on architectural features can provide resistance to some mimicry attacks but requires long execution windows to accurately observe or predict patterns, which results in high memory overhead and long time-to-detection.

2.2 Timing-Based Anomaly Detection

Several previous efforts have used timing of system events within malware detection methods. The SHEILD approach [25] monitors the timing of individual basic blocks to detect anomalies. SHEILD instruments the software executable with special instructions to verify the worst-case execution time (WCET) for basic blocks. While this approach is capable of detecting code injection attacks, SHEILD comes at the expense of increased area, increased code size, and a 6.6% performance overhead, which can be detrimental to embedded systems with tight timing constraints. Similarly, iCUFFs [26] utilizes timing information to detect software attacks as well as soft errors, but comes at the expense of a 44% performance overhead.

SecureCore [36] monitors the timing distribution of each basic block within different execution paths. The SecureCore approach inserts instructions at the start and end of each basic block, and stores the execution information within a scratchpad memory. A separate processor core analyzes the execution trace within the scratchpad to detect specific execution of the basic blocks that deviate from the expected distribution. SecureCore achieves good detection rates, but suffers from a high false positive rate, and its fine-grained detection requires high memory usage to store timing distribution, with 100s to 1000s of bins for each basic block.

Zimmer et al. [38] proposed a time-based intrusion detection method that verifies time bounds for function call return paths to detect code injection attacks. Although periodically waking up verification procedures or reducing the number of checkpoints can decrease the overhead, this method does not explore how to reduce checkpoints without sacrificing detection accuracy.

2.3 Hardware-Based Anomaly Detection

The software-based detection methods mentioned above either require modifying the software application to incorporate special instructions or require a separate processor to perform analysis of monitored events to detect anomalies, both of which incur high overheads. Hardware-based approaches can eliminate performance penalty. Rahmatian et al. [28] designed a hardware-based anomaly detection component integrated within the processor datapath that monitors the instruction register to detect system calls. These approaches provide faster detection compared to software-based methods and are able to detect malicious activity without affecting the system performance. However, the approaches are susceptible to mimicry attacks and require modification to the processor core.

In resource-constrained embedded systems, hardware-based anomaly detection requires area- and performance-efficient detection methods. Because exhaustive coverage of all system behaviors is infeasible or prohibitive for complex applications, finding the most detect-efficient part of the application is important. However, reducing the detection space is a hard problem in anomaly detection without *a priori* knowledge of the malware. Maxion and Tan [20] defined a similarity metric to evaluate different sub-sequences within an application and demonstrated that sub-sequence

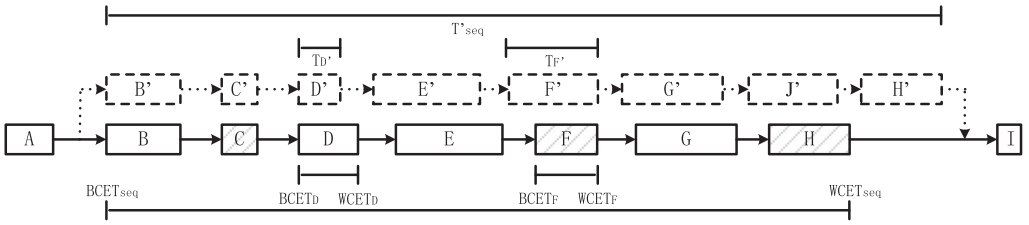


Fig. 1. Execution order of specific control flow events. A-I are normal system sequence, B'-H' are mimicry malware sequence.

selection has a significant impact on detection capability. Hence, a key challenge in hardware-based anomaly detection is the trade-off between coverage and area.

In contrast with previous work, the RAD approach has several advantages. First, the RAD approach detects both changes in the execution sequences and timing to detect malicious activities, supporting events defined at any level of granularity. Second, the on-chip RAD hardware non-intrusively interfaces with the processor's trace port to ensure the system execution is not perturbed, thereby incurring no performance penalty. Third, the RAD does not require any modification to processor core, and can be used readily integrated into either soft or hard processor cores. At last, the RAD provides a method to reducing detection space by reducing detection events based on timing distribution.

3 SYSTEM DEFINITIONS, ASSUMPTIONS, AND THREAT MODEL

We assume anomaly-based malware detection is needed to ensure protection from zero-day exploits and mimicry attacks. As such, mimicry malware [34] is the primary threat to embedded systems using anomaly-based detection methods. The following outlines the definitions, assumptions, and threat considered in the presented anomaly detection methodology:

(1) *Event.* A single instruction execution or a set of instructions execution identified by a start instruction and an end instruction.
(2) *Control Flow Event.* An event whose execution decides which of two or more paths should be followed (e.g., a branch instruction execution identified by a single instruction or a function execution identified by function call and function return).
(3) *Nullified Control Flow Event.* An event that is executed but does not perform the intended operation. For example, a function/system call can be nullified by substituting null arguments (e.g., null pointer, non-existent file) in place of the actual arguments. Nullified control flow events are necessary for mimicry malware to avoid detection, but the method by which nullification occurs must ensure those events do not lead to other changes in the system behavior (e.g., null pointer exception), which would make the attack easier to detect or lead to system failure.
(4) *Mimicry Malware.* Mimicry malware execution sequence M:

$$M \cap A \neq \emptyset, \quad where\ M \subseteq \Sigma,\ A \subseteq \Sigma$$

where Σ is the execution sequence of security-relevant events, and A is the execution sequence of events defined as normal system behavior. For example, given a control flow events sequence for the normal system operation (e.g., event A to I) in Figure 1, an attacker may identify a subset of function/system calls that achieves the desired malicious activity (e.g., event C, F, and H). Assume an anomaly-based detection method monitors all A-I

events in that sequence. To avoid detection, a mimicry malware must mimic the execution sequence covering the first event to the last event required for the malicious activity (e.g., event B to H). A mimicry malware will nullify the events that are not required for the malware activity itself (e.g., nullify event D to D'). As a result, the detector cannot detect the abnormal execution sequence.

(5) *Nullification rate*: To execute the same malicious activity, different implementations of mimicry malware may mimic different length of the execution sequence depended on different vulnerabilities. Thus, a nullification rate γ is defined as:

$$\gamma = \frac{nullified\ events}{total\ events\ in\ mimicked\ sequence}$$

For example, in Figure 1, events along the dotted path are malware events. A mimicry malware attack may only use the sequence of events B-H in original application. Within that sequence, C, F, and H are necessary for the malicious activity, and the remaining events will be nullified. So the nullification rate for this mimicry malware is: $\gamma = 4/7 = 0.57$. The nullification rate indicates the difficulty to implement and detect a mimicry malware. Generally, a higher nullification rate leads to easier detection.

In this article, we mainly focus on the detection of mimicry malware, with the following assumptions:

(1) The target software application consists of multiple software tasks or threads executing on a single processor core. Although the presented approach and hardware design can be extended to multicore systems, that work is beyond the scope of this article.
(2) The attacker either has access to system software or can create a control flow graph of the system execution by simulation.
(3) The attacker is able to remotely insert the malware into the system utilizing a software exploit, which may be unknown, or known but unpatched, at the time of insertion.
(4) The attacker knows the granularity of events (e.g., system calls, function calls) monitored at runtime but not the specific events being monitored.
(5) The mimicry malware nullifies all monitored events in the malware path that are not required by the malware. Nullified events are created such that the system execution will not crash and the execution sequence will not be changed.
(6) The RAD hardware can only be programmed during a secure boot process or secure firmware update.

To evaluate the RAD approach, we consider several common software attacks that break confidentiality, integrity, and availability. Although simple attacks can also be easily detected by our design, we consider attacks that that are sophisticated enough to avoid detection by sequence-based anomaly detection methods. We mainly consider several mimicry malware, including *File Manipulation, Information Leakage, Interrupt Disable*, and *Control Loop Delay*. We also consider a *Denial of Service* (*DoS*) attack that breaks availability and a *Fuzz Attack* that breaks integrity.

4 RUNTIME ANOMALY DETECTION METHODOLOGY

Figure 2 presents the design flow of the RAD methodology. The system requirements include both execution sequence constraints that can be defined at the level of system calls, functional calls, or control flow operations, and timing constraints within those execution sequences. Note that the specification of system requirements is typically already an essential task within the design of embedded systems. The security analysis combines control flow events (e.g., system calls, functions

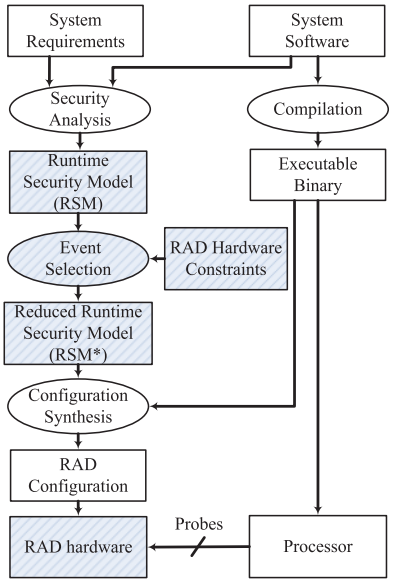


Fig. 2. Design flow for runtime anomaly detection utilizing system requirements including execution order and timing information.

calls) in the software application and timing information from the system requirements to generate a runtime security model. The runtime security model can be reduced by selecting a subset of events to monitor in order to meet hardware area constraints. The reduced runtime security model will be configured into RAD hardware. The RAD hardware interfaces directly to the processor's trace port to observe the execution of the software application. Processor trace ports and external trace interfaces are commonly provided by processor manufacturers [1, 23] and widely used within systems-on-a-chip (SOCs), although these interfaces are typically used for debugging and testing [33]. By interfacing the RAD hardware between the processor's trace port and the SOC's external trace interface, the RAD hardware can observe the system execution without affecting the execution of the application itself (i.e., zero performance overhead). The on-chip RAD hardware asserts a non-maskable interrupt whenever anomalous execution is detected, which triggers a recovery or mitigation process. This article, however, focuses on detection and does not require or advocate a specific recovery or mitigation approach.

4.1 Runtime Security Model (RSM)

The Runtime Security Model (RSM) specifies the normal system behavior, which will be verified at runtime. Figure 3(a) illustrates four types of events utilized within the RSM, which cover all scenarios in a software application. A *branch event* is an event that conditionally modifies the system execution based upon a decision variable (e.g., branch instruction). A *triggering event* is an event that triggers the execution of a corresponding *triggered event*, denoted by a *triggering dependency* (dashed arrow). For example, using POSIX threads, signaling a condition variable in one thread triggers the execution of another thread waiting on that condition variable. In this example, the *cond_signal()* call is the *triggering event*, and the *cond_wait()* call is the *triggered event*. A *regular event* is a single event that is neither a branch, triggering, nor triggered events. Finally, a *pair event* is two related events (e.g., pair of instruction addresses for function call and return

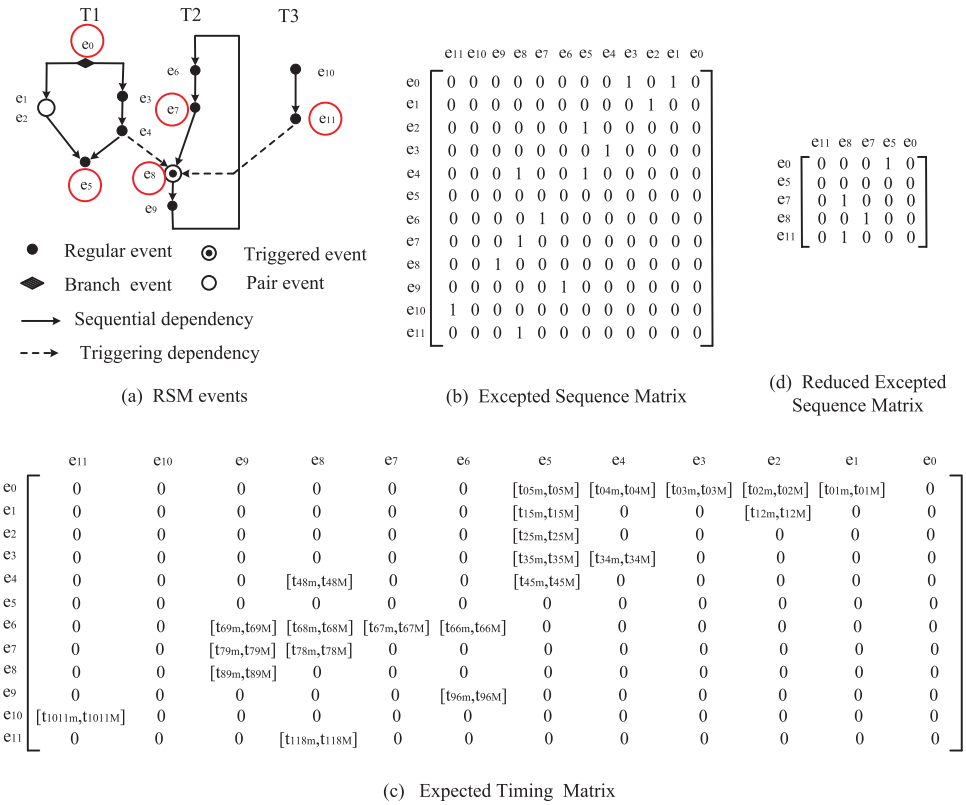


Fig. 3. Runtime Security Model: (a) illustration of events within the RSM and specification of normal system behavior, (b) expected sequence matrix, (c) expected timing matrix, and (d) reduced expected sequence matrix.

locations). The execution ordering between two events is indicated by the sequential dependency (solid arrow).

The RSM supports various levels of modeling, including control flow, system call, function call, and basic block levels by defining the corresponding regular and pair events. The RSM is formally defined using an Expected Sequence Matrix (ESM) and an Expected Timing Matrix (ETM). The ESM is a binary matrix that defines the expected ordering of events, where each entry (i, j) indicates if event e_j is a successor of event e_i. Note that the entry in the matrix is the event with a single instruction address. Figure 3(b) models the normal system execution behavior using the ESM for three software tasks. For example, e_1 and e_3 in T1 are the expected events following e_0, so the first row in the ESM is 000000001010. A triggered event can be triggered by more than one trigging event (e.g., e_8 will occur only after e_7 has occurred, and either e_4 or e_{11} has occurred). This triggering dependency is defined the same way as a sequential dependency in the ESM.

Figure 3(c) presents the ETM that defines the timing requirements between events, where each entry (t_{ijm}, t_{ijM}) specifies the best-case and worst-case execution time as [BCET, WCET]. In this example, the timing requirement from e_2 to e_1 is 0 because e_1 occurs before e_2, and the timing not in the requirement can also set to 0 (e.g., $e_8 - e_6$). As events may repeatedly occur (e.g., an event within a loop), an entry (i, i) defines the minimum and maximum time between two occurrences of this event, such as event e_6's timing requirement $[t_{66m}, t_{66M}]$.

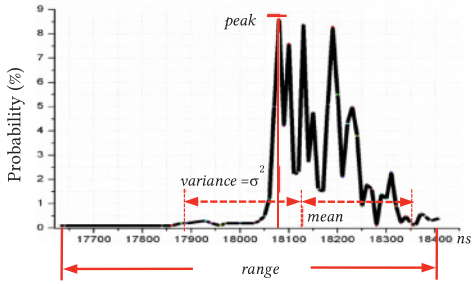


Fig. 4. Timing distribution illustrating range, variance, and peak probability for an event.

With the RAD hardware, the ESM and ETM can be configured independently, such that different subsets of events monitored at runtime for sequence and timing may be used. Section 4.3 discusses how the ESM and ETM models are utilized in hardware to detect malware at runtime. Given the design requirement and software tasks, the ESM can be created using static analysis tools, and given timing requirement, the ETM can be created using either static analysis or experimental measurement from executing normal system sufficient number of times.

4.2 Control Flow Event Selection

The RSM may include all operations in the software application, and it may be infeasible to verify the event dependencies and timing requirements for all events within the on-chip RAD hardware. In order to reduce the RSM, we present several event selection methods to select a subset of the events within the RSM to create the reduced RSM (RSM*), which will be configured within the hardware detector. The event selection methods determine which events should be monitored at runtime given a constraint on the number of events supported in hardware. Figure 3(d) presents an example of ESM in RSM* when selecting e_0, e_5, e_7, e_8, e_{11} (circled events) to monitor.

The goal for selecting events is to maximize the detection accuracy subject to constraint for the number of events that can be monitored in hardware. Numerous methods exist for selecting the monitored event subset (e.g., maximize code coverage). However, given the focus on mimicry attacks, which are resistant to sequence-based detection, we consider selecting pairs of events for which mimicking the timing is very difficult, thereby maximizing the expected detection rate. An attacker's behavior cannot be predicted beforehand, and this challenge leads us to analyze various properties of the timing distribution for control flow events. Using *in situ* execution traces of the target embedded system for sufficiently long executions, the resulting timing behaviors can be analyzed to generate the corresponding timing distribution for each system event. Figure 4 presents an example timing distribution for one control flow event, illustrating several metrics utilized for event selection. We considered three different metrics used in selecting monitored events within the RSM*.

Range. Range is defined as the difference between the BCET and WCET. Selecting control flow events with a small execution range may make mimicking the execution behavior more difficult, since changes in timing would have a larger effect on the overall timing of the event. So, the smaller the range, the more difficult it is to nullify the event without detection or to utilize the event for malicious activity.

Variance. The mean is the average execution time between two events across all execution traces, and the variance indicates how much individual execution time varies from the mean. Control flow events with small variance in execution time will have a timing distribution that is more closely

centered on the mean execution time. If the execution time of malware varies more than the normal execution behavior, detecting malicious activity may be easier.

Peak Probability. Peak probability is the maximum probability of a specific execution time between two events or for a single event. For example, the peak probability for the event in Figure 4 is 8.7%, meaning that 8.7% of the time the execution time for that event is identical. Selecting events whose timing has a higher peak probability means the timing is more concentrated, and deviations from the peak execution time may be easier to detect.

We also consider the correlation of the timing that captures dependencies between control flow events. If two events are correlated, then changing the execution of one event will change the execution time of both correlated events. Monitoring both strongly correlated events (i.e., correlation coefficient greater than 0.5) may be redundant. Monitoring only one event of the correlated events can directly detect changes in the monitored event and indirectly detect changes in the correlated event.

Using these metrics, we evaluate and analyze three greedy event selection methods presented in Algorithm 1. *Minimum range selection* selects events with minimum range, *minimum variance selection* selects events with minimum variance, and *maximum peak probability selection* selects events with maximum peak probability.

4.3 Runtime Anomaly Detection Hardware

With the RSM*, which is presented as the ESM and ETM illustrated in Figure 3, the RAD hardware verifies the execution sequence and execution timing for each monitored control flow event at runtime, and detects anomalous execution behavior. The RAD hardware directly interfaces with the trace port of the microprocessor, specifically monitoring the program counter (PC) addresses to identify control flow events. When a mismatch between the runtime execution and RSM* is detected, the RAD hardware notifies the system by asserting a non-maskable interrupt.

ALGORITHM 1: Monitored Event Selection Algorithm

Input: $m \in$ {range, variance, peak probability}, C = max events constraint, E = {candidate events}
Output: selected events subset S

```
1:  S ← {}
2:  if m = range      then sort E by increasing range
3:  if m = variance   then sort E by increasing variance
4:  if m = variance   then sort E by decreasing peak probability
5:  while size of |S| < C do
6:      e ← E.firstevent;
7:      if m = range      then S = S + {e} , E = E − {e};
8:      if m = variance   then S = S + {e} , E = E − {e};
9:      if m = range      then S = S + {e} , E = E − {e};
10:     if coff (k, e) > 0.5 where k ∈ S then S = S − {k};
11: return S;
```

Runtime anomaly detection using the RSM* should be designed with the following goals: (1) maximize detection accuracy, (2) minimize false positives, (3) minimize time-to-detection, and (4) minimize area and power consumption. Detection accuracy mainly depends on both the granularity of the RSM and the event selection method. False positives mainly depend on comprehensive system requirements and sufficient training for the ETM. Time-to-detection is the time between when a malware event executes and the anomalous execution behavior is detected.

Time-to-detection and hardware area cost mainly depends on detection granularity and the hardware implementation of the detection algorithm.

4.3.1 Hardware Runtime Anomaly Detection Algorithms. To detect anomalous execution sequence efficiently at runtime, the RAD hardware uses two binary vectors to dynamically record the execution states. These vectors are utilized to capture both conjunction and disjunction relationships between execution sequence. The expected vector (*ev)* records the indices for those events that are expected to execute next. For example, if the next event executed could be e_i, then the i^{th} bit in *ev* will be set to 1. The *ev* is updated each time a monitored event occurs. The triggering vector (*tv)* records the indices for triggering events that have occurred. For example, when e_4 in Figure 3(a) occurs, the 4^{th} bit in the *tv* is set to 1, resulting in *tv* = [000000010000]. An event's index bit in the *tv* is cleared when the triggered event occurs (e.g., when e_8 in Figure 3(a) occurs, the 4^{th} and 11^{th} bit in *tv*, which correspond to events e_4 and e_{11}, are cleared). The indices of all triggering and triggered events indices are stored in the triggering event mask (*TGM*) and triggered event mask (*TDM*), respectively. Note that in our convention, a vector is updated at runtime and a mask is statically configured. The task start mask (*TSM*) defines which event starts the execution for each software task (e.g., the first event in all interrupt handlers is a start event). The number of bits within *ev, tv, TGM, TDM,* and *TSM* is equal to the number of monitored events. The task mask (*TM*) is maintained by each task to differentiate between tasks and handle with context switch across tasks.

ALGORITHM 2: Runtime Anomalous Sequence Detection Algorithm

Input: PC, ESM, TSM, TDM, TGM, TM
Output: anomalous event index *i*
Repeat each valid PC

```
1:   i = FindEventId ( PC );
2:   if  TSM[i] == 0 ||( TSM[i] == 1 && Task[i] == PrevTask) then
3:       if ev[i] != 1 then
4:           return i;
5:   if TDM[i] == 1 && tv[i] != 1 then
6:           return i;
7:   if  TGM[i] == 1 then
8:       tv = (ESM[i] & ~TM[ Task[i] ]) |tv;
9:   else
10:      tv[i] = 0;
11:  ev = (ev & ~TM[ Task[i] ]) | (ESM[i] & TM[ Task[i] ]);
12:  PrevTask ← Task[i];
```

Algorithm 2 presents the pseudocode for the RAD sequence detection algorithm. The input of this algorithm is PC address from the processor trace port, the ESM and necessary masks, and the output is the anomalous event index. When a monitored event's PC address is detected (e.g., e_8), the detector firstly verifies if the event is expected to happen. For example, if the previous event detected was e_7, then current *ev* = [000100000000] according to the ESM in Figure 3(b). The corresponding bit for e_8 is 1, which confirms e_8 is an expected event. At the same time, if e_8 were a triggered event, indicated by *TDM*, the detector would verify if the corresponding bit for e_8 is 1 in *tv*. *tv* vector maintains a history that records if triggering events e_4 or e_{11} had happened. After verification, both *ev* and *tv* vector are updated accordingly. Note that *ev* only updates the bits within task and *tv* only updates the bits of other tasks, so that context switches will not affect sequence detection.

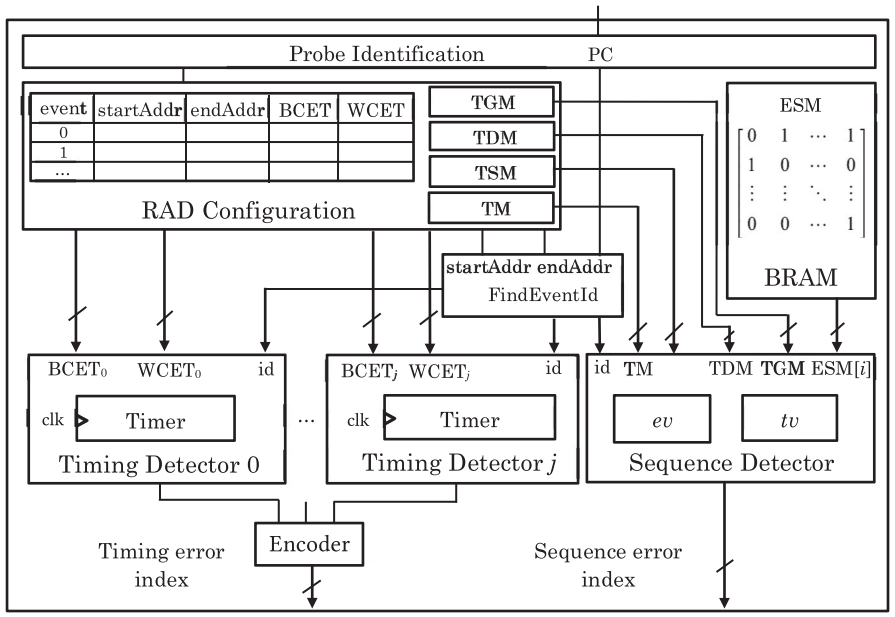


Fig. 5. Runtime anomaly detection hardware architecture.

Compared to finite state automata or other sequence based detection model, the RSM model is simpler, faster, and well suited for hardware implementation. Because only a single row of the ESM and a single bit of each vector/mask must be accessed for each valid PC, the resulting hardware avoids a complex decoding process and enables the ESM to be efficiently stored in block RAM (BRAM). Thus, the resulting hardware for anomalous sequence detection only requires a small number of registers for the vectors, masks, and current task index.

As malware that mimics the execution sequence can escape detection from a sequence-only detection method, we further incorporate timing information within the RSM and developed algorithm for detecting anomalous timing. The ETM is an integer matrix and can be very sparse, so the non-zero entries in the ETM are stored as an array in the RAD hardware. For each timing detection target (e.g., timing between two events or event with two instruction addresses, such as a function call and return), a timer is used to record time elapse between detecting start event and the end event. Algorithm 3 presents the pseudocode for the runtime anomalous timing detection algorithm. The input of the algorithm is PC address from trace port of the processor, and the output is the anomalous event's index. To reduce the area required for comparing the elapsed time with the BCET and WCET, the RAD hardware is configured using the range [-BCET, WCET-BCET] instead of the range [BCET, WCET]. When the end event is detected, the RAD hardware checks the most significant bit of the timer. If the bit is 1 (i.e., the current timer value is negative), the execution time is less than BCET, then the anomalous event index is returned. Instead of waiting for the end event to be detected, a comparator is used to compare the current time with WCET each clock, which minimizes the time-to-detection.

4.3.2 Hardware Architecture and Implementation. Figure 5 presents the overall system architecture integrating the RAD hardware. The RAD hardware is configured with the RSM* and monitors the execution of the system to detect sequence and timing anomalies. The RAD hardware consists of five main components, *Probe Identification, RAD Configuration, BRAM, Sequence Detector*, and

Timing Detector. The *Probe Identification* detects the control flow event, which can be dynamically detected by observing the microprocessor's trace port to find matching PC values for software instructions. The *RAD Configuration* stores monitored events' instruction addresses, timing requirements, and the *TGM, TDM,* and *TM* masks. The ESM is stored in a *BRAM*, since it is a compact binary matrix and only a single row for the current event needs to be read. The *Sequence Detector* implements Algorithm 2 and determines if execution violates the expected sequence. For each event with timing requirements, the *Timing Detectors* implement Algorithm 3. Internally, each *Timing Detector* utilizes a single timer for measuring the elapsed time between the start and end events.

ALGORITHM 3: Runtime Anomalous Timing Detection Algorithm

Input: PC, WCET_BCET = {WCET*j*-BCET*j*|j ∈ event pairs}
Output: anomalous event index *j*
Repeat each clock cycle

```
1:    j = FindEventId ( PC );
2:    if PC == MonitorStartEvent [j] then
3:        timer[j] = -BCET[ j];
4:        DetectEnable = 1;
5:    else if PC == MonitorEndEvent [j] then
6:        if timer[j].highestbit == 1 then
7:            return j;
8:        else
9:            DetectEnable = 0;
10:   if DetectEnable == 1 && timer[j]>WCET_BCET[j] then
11:       return j;
12: timer[j] ++;
```

To evaluate hardware area requirements, we implemented the RAD hardware on a Xilinx Spartan-6 XL45 FPGA for 8 to 128 single-instruction events using 32-bit register for all PC addresses and 32-bit timers for all *Timing Detectors*. The RAD hardware operates at the same 100-MHz frequency as the software application executing on a MicroBlaze processor implemented within the same FPGA, which ensures the RAD hardware can detect all relevant events from the processors' trace port. The required lookup tables (LUT), flip-flops (FF), and BRAMs are presented in Table 1. Overall, the size of the RAD hardware scales linearly in LUTs and FFs relative to the number of monitored events. Since the ESM and ETM are stored within the registers and BRAM within the RAD hardware, no additional external memory or storage is needed.

We note that the hardware area can be reduced if the application's program memory address locations are known and within a fixed region. For example, if an application's program memory addresses are between 0×08010000 and $0 \times 0801FFFF$, the number of bits needed to store the PC address for start and end events can be reduced to 16 bits, and single comparator for the remaining 16 bits can be used. Area can further be reduced if a maximum range for all timing requirements is known *a priori*. For example, the timer's bit widths could be reduced to 26 bits if the maximum range for (WCET-BCET, |BCET|) is 60,000,000 clock cycles. Table 1 further presents the area requirements for two optimized RAD hardware implementations, one using 16-bit PC addresses and one using both 16-bit PC addresses and 26-bit timers. Using 16-bit PC addresses reduces LUTs and FFs by 14% to 18%, and using both 16-bit PC addresses and 26-bit reduces LUTs and FFs by 22% to 36%.

Table 1. Hardware Area Required for: (1) RAD Hardware using 32-Bit Registers for PC Addresses 32-Bit Timers, (2) Optimized RAD Hardware using 16-Bit Registers for PC Addresses, and (3) Optimized RAD Hardware using Both 16-Bit Registers for PC Addresses and 26-Bit Timers

# of events	32-bit PC addresses 32-bit timers			16-bit PC addresses 32-bit timers			16-bit PC addresses 26-bit timers		
	LUT	*FF*	*BRAM*	*LUT*	*FF*	*BRAM*	*LUT*	*FF*	*BRAM*
8	1,119	800	9K	950	671	9K	875	654	9K
16	2,002	1,494	9K	1,673	1,239	9K	1,542	1,146	9K
32	3,865	2,891	9K	3,187	2,382	9K	2,463	2,145	9K
64	8,415	6,884	18K	7,006	5,861	18K	6,387	5,336	18K
128	16,505	13,658	2 × 18K	13,636	11,609	2 × 18K	10,642	10,502	2 × 18K

Based on our prior experience designing non-intrusive profilers [24], the RAD hardware's energy consumption is estimated to be less than 5% of that of the processor. Additionally, we anticipate that low-power design techniques can be used to further reduce energy consumption, such as leveraging the fact that monitored events occur at a much slower rate than the processor frequency to reduce the RAD hardware's operating frequency, although the evaluation and optimization of energy consumption is left as future work.

5 EXPERIMENTAL EVALUATION

We developed both an FPGA prototype and a SystemC simulation framework for the RAD approach to evaluate the timing-based anomaly detection and event selection metrics. The network-connected pacemaker application and all malware attacks were implemented on the FPGA prototype to collect the event execution sequence and timing for both normal and malware execution for all monitored events. The collected execution data is then input into the simulation framework to simulate the RAD hardware and determine the detection and false positive rates. We utilized a simulation framework in addition to the FPGA prototype to enable: (1) controllable and repeatable execution of the normal application execution and malware execution, (2) efficiently evaluating different sets of monitored events, and (3) comparison with other anomaly-based detection methods, namely SecureCore, whose memory requirements exceed the available BRAMs within the FPGA. We note that the FPGA prototype was used to validate all false positive rates for normal execution of the benchmark application, and to validate the detection rates for the RAD approach for all malware considered.

5.1 Network-Connected Pacemaker Application

We developed a network-connected pacemaker prototype using a Xilinx Spartan-6 XL45 FPGA. The prototype enables the implementation and analysis of different vulnerabilities and malware, which is not possible using standard embedded application benchmarks. The hardware components for the network-connected pacemaker include a pacer, a simulated heart, a cardiac activity sensor, and four timers [14, 31], as presented in Figure 6(b). The simulated patient's heart generates irregular beating activities and can react to the pacer's signal. The cardiac activity sensor interfaces to the simulated heart model and sends the measured heart activity to the microprocessor using interrupts.

The output from the cardiac activity sensor also controls the Atrio-Ventricular Interval (AVI) timer and the Ventriculo-Atrial Interval (VAI) timer. The VAI timer is used to maintain the appropriate delay between the atrial activation and the ventricular activation, and will generate an interrupt if the AVI exceeds a specific interval configured by the cardiac physician. Similarly, the

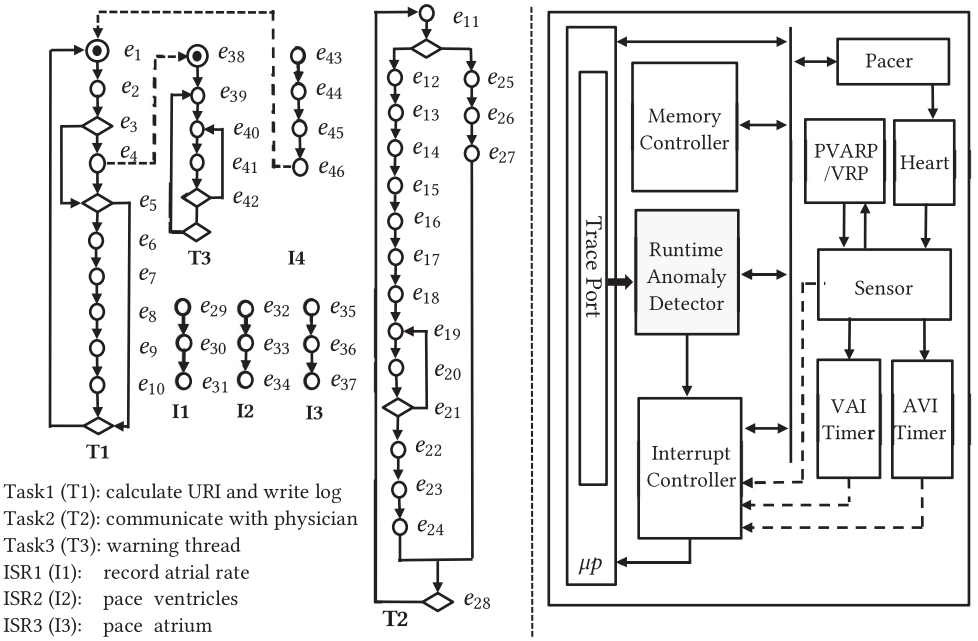


Fig. 6. Benchmark of network-connected pacemaker. (a) High-level overview of RSM for the software tasks. (b) Overview of embedded system hardware architecture for the benchmark.

AVI timer is used to maintain the appropriate delay between ventricular and atrial activity. The Ventricular Refractory Period (VRP) timer and Post-Ventricular Atrial Refractory Period (PVARP) timer filter noise that causes undesired pacemaker behavior in the ventricular and atrial channels, respectively.

Figure 6(a) presents a high-level overview of the RSM for the software tasks within the network-connected pacemaker. The software consists of three tasks and four interrupt service routines (ISRs). The ISRs interact with the pacemaker's cardiac activity sensor and timers, and have the highest priority. ISR2 and ISR3 perform the atrial and ventricular pacing, respectively. ISR1 and ISR4 record ventricular and atrial activity, respectively. Task T1 calculates the Upper Rate Interval (URI) and records cardiac activity to a daily log file. Task T3 analyzes the cardiac activity and detects a high URI, which indicates the pacemaker cannot pace the heart correctly or that pacemaker's cardiac activity sensor has malfunctioned. In the event of a high URI, the pacemaker immediately transmits a warning message to alert the physician. Task T2 is responsible for communication, by which the physician can configure the pacemaker's settings or a home monitoring device can access daily logs of the cardiac activity.

To detect anomalies at runtime, the RSM's ESM incorporates system-level functional requirements and control flow execution information extracted from the software application as shown in Figure 6(a). The system-level functional requirements, which focus on the safety requirements of the network-connected pacemaker, are:

(1) When the ventricle is paced, the ventricular activity recording interrupt should be triggered. So when I3 occurs, I1 should be triggered immediately after I3;

Table 2. System Timing Requirements for a Network-Connected Pacemaker

Requirement	Purpose	Time in ms
Tp	Time duration atrial activity must be sensed	110
T_Q	Time duration ventricular activity must be sensed	100
T_{pulse}	Time current pulse must be maintained	1
AVI	Time for ventricle to fill following an atrial contraction	150
VAI	Ensure an atrial pulse following a ventricular activity	850

(2) When the atrium is paced, the atrial activity recording interrupt should be triggered. So when I2 occurs, I4 should be triggered immediately after I2;
(3) Atrial activity should follow ventricular activity, so I1 should occur after I4;
(4) Ventricular activity should follow atrial activity, so I4 should occur after I1.

The system/function call-level control flow event sequences for all software tasks are automatically analyzed using static analysis tools [9] to create the ESM, after which the system-level functional requirements must be manually integrated.

Similarly, the RSM's ETM integrates system-level timing requirements, and timing information for all control flow event calls, function calls, and ISRs within the applications. Within the ETM, the timing bounds for system/function calls are generated using experimental data collected by dynamically running the normal system, specifically the pacemaker prototype, which can be automated. The system-level timing requirements are then manually integrated with the ETM. The system-level timing requirements for the pacemaker are presented in Table 2. These timing requirements affect our RSM in following aspects:

(1) The time between detected atrial activity and the I1 interrupt should be less than 110ms;
(2) The time between detected ventricular activity and the I4 interrupt should less than 100ms;
(3) The duration of the pulse releasing process should be 1ms, which happens in I2 and I3.
(4) The time between I2 and I4 should be less than 150ms;
(5) The time between I3 and I1 should be less than 850ms.

Determining precise BCET and WCET for control flow events can be challenging. As such, the possibility of reporting normal system behavior as potential malicious activity (i.e., false positives) exists both when using analysis tools and when experimentally measuring these timing values. The accuracy of the BCET and WCET using an experimental setup will depend on the number of system executions used to build the RSM, which will in turn affect the false positive rate of the RAD. The false positive rate is measured by the number of times RAD asserts alert divided by number of times system is executed.

The network-connected pacemaker application's complete RSM consists of 44 pair events, which corresponds to 88 individual events, which can be fully configured within the RAD hardware detector supporting 128 individual events, presented in Table 1. However, for smaller RAD hardware implementations, event selection methods will be needed to determine the reduced RSM*.

Figure 7 presents the RAD's false positive rate for the network-connect pacemaker with increasing number of system executions utilized to build the RSM. For each RAD configuration, we executed the system 1000 times without malware to determine the resulting false positive rate for all system call events. A false positive rate less than 15% is achieved using 600 executions, and a false positive rate of 0% is achieved using more than 900 executions. The result indicates that, after executing more than 900 times, the range of BCET and WCET is unlikely to expand. Therefore,

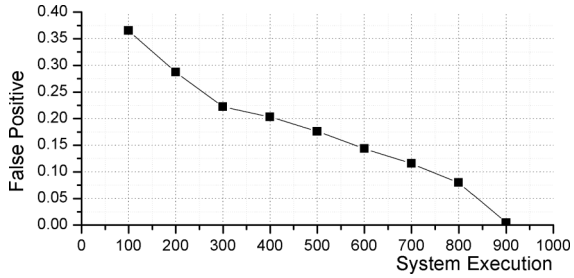


Fig. 7. False positive rate for RAD as a function of the number of system executions used to build the runtime security model.

Table 3. Overview of Malware Types Considered

Malware Type	Involved Tasks	Attack Implementation
File Manipulation*	T1, T2	Reads the cardiac activity log files, modifies the recorded activity and writes the modified activity back to the log file to deceive the physician
Control Loop Delay*	I3	Extends the pacing time to the patient generated by the pacer
Information Leakage*	T1, T3	Reads the cardiac activity log files, send the file to a malicious IP address
Interrupt Disable*	I1, I2, I3, I4	Compromise interrupt controller and disable one or multiple interrupts
Denial of Service (DoS)	T2	Continual transmit packets to target port to deny the access from the physician
Fuzz Attack	T1, T2	Randomizes the log buffer size or *physician configuration*, thus fuzzes the input of multiple functions

*indicates mimicry malware.

we used an automated setup to execute the system 1000 times, requiring 1 hour, to determine estimates for the BCET and WCET for the timing specification of the normal system behavior.

For each malware type and nullification level, the system was executed 100 times to collect timing for all control flow events within the malware. The normal and malware timing information is input into the simulation framework to evaluate detection rate and time-to-detection of the RAD design.

5.2 Malware Attacks

Table 3 summarizes the malware types we considered, which include multiple mimicry malware that do not change the application's execution sequence, a DoS attack that disables the communication between the pacemaker and the physician, and a Fuzz Attack that disrupts the execution by modifying (or fuzzing) parameters of system/function calls. The DoS and Fuzz Attack are very hard to detect in current intrusion detection systems, since they do not require code injection. Each of the malware types is representative of real attacks observed by cybersecurity researchers, and the malware collectively affects all software tasks within the network-connected pacemaker. Note that we do not utilize malware attack libraries because the template attacks contained therein focus on

exploiting specific system vulnerabilities, which affects how malware is injected within a system. But the template attacks' behaviors do not mimic the execution sequence of our application and would be rapidly detected.

The four mimicry malware include *File Manipulation, Control Loop Delay, Information Leakage*, and *Interrupt Disable*. The File Manipulation malware manipulates the cardiac activity log file to deceive the physician, with the intent of leading the physician to incorrect diagnosis or a potentially life-threatening misconfiguration of the pacemaker. The implementation of this malware involves reading cardiac activity log file in T1, and manipulating the log file in T2. The Control Loop Delay malware distorts parameter of the pacer configuration in I3, and extends pacing time or increases the pacing current intensity, thereby threatening patients' safety. The Information Leakage malware covertly reads data in cardiac activity log of the patient in T1, and sends this information to a malicious data center in T3. The Information Leakage malware also increases the execution frequency of T3 in order to collect sufficient data in a short period of time. The Interrupt Disable malware disconnects the hardware sensor or pacer from the software analyzer and controller by covertly disabling the associated interrupt, thus affects pacemaker's normal functionality. Either one or multiple interrupts are considered within the Interrupt Disable malware.

The two non-mimicry malware include *DoS* and *Fuzz Attack*. The DoS is a common attack in networks and it is analyzed in our experiments to evaluate its impact on system timing. The network-connect pacemaker application has three open ports, and the port used to send logs to the physician in T2 is flooded with packets [32]. The Fuzz Attack manipulates system execution by randomizing data values and system/function call arguments, which is usually implemented by interpolating data in memory. We analyzed the Fuzz Attack to evaluate its impact on execution timing and detection thereof using the RAD approach. Note that fuzzing system/function call arguments could be utilized within mimicry attacks to nullify events, instead of using null arguments.

5.3 Simulation Setup

A data-driven SystemC simulation framework was developed to simulate the network-connected pacemaker benchmark, malware, and RAD hardware. The structure of the simulation is the same with prototype presented in Figure 6. Each control flow event in the benchmark is scheduled as an atomic execution, using the execution timing collected from the FPGA prototype. This *execution replay model* can precisely simulate the execution of the FPGA prototype (i.e., replay the precise sequence and cycle accurate timing collected for each recorded execution of the application).

The simulation framework further supports a *stochastic model* that simulates the execution of the normal system behavior using probabilistic distribution for each system event. The stochastic model enables simulating additional system executions and different execution scenarios without collecting all data from the FPGA prototype, and enables simulating other malware detection approaches to compare the resulting false positive rates and detection rates. The stochastic model uses a statistical regeneration process to create a representative timing distribution for each control event.

The executions time of an event may not obey a single standard Gaussian distribution, considering different execution paths and architecture influence. For example, Figure 8(a) is the histogram of 1000 timing samples for a single event collected from the FPGA prototype. Notice that the histogram does not match a single Gaussian distribution. There are two widely used fitting approaches in statistics for this type of distribution, high-order polynomial curve fitting and mixture of skew-normal fitting [27]. Both approaches work well for fitting and prediction but are not suitable to regenerate samples, because these approaches would end with very complex

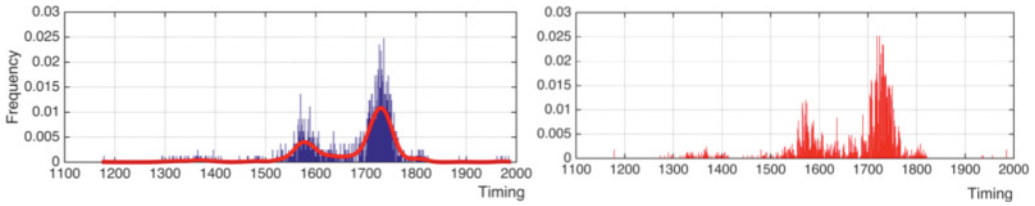


Fig. 8. Comparison of fitting effect between weighted joint normal distribution and Kernel Density Estimation (KDE) with (a) normalized histogram of 1000 timing samples from prototype and KDE distribution curve with 2000 bins and (b) normalized histogram of 2000 timing samples generated by Algorithm 4.

ALGORITHM 4: Timing Data Regeneration

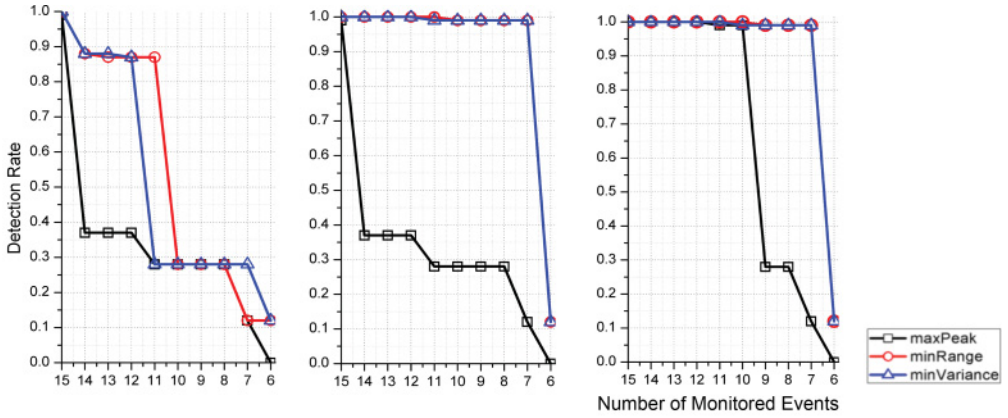
Input: Raw timing data samples $\overline{t}$ for one event
Output: Regenerated timing sample s

calculate histogram $[\overline{\boldsymbol{x}}, \overline{\boldsymbol{y}}]$ based on $\overline{t}$
for each x in $\overline{\boldsymbol{x}}$ ***do***
  if y is peak ***then***
    $\overline{peak}.add\,(y)$
    $\overline{\mu}.add(\text{x})$
$\overline{interval}$ = DivideIntervals ($\overline{\boldsymbol{x}}$, $\overline{peak}$)
$\overline{\boldsymbol{\sigma}}$ = NormalFitting ($\overline{interval}$, $\overline{\mu}$)
for each interval ***do***
    $\overline{weight}\ .add(\ \sum_{x \in interval_i} y\ )$
$gddEngine$ = GenerateDiscreteDistributions ($\overline{weight}$)
$gndEngine$ = GenerateNormalDistributions ($\overline{\boldsymbol{\mu}}$, $\overline{\boldsymbol{\sigma}}$)
while $s \notin interval_s$ ***do***
    $r \leftarrow$ random generated by $gddEngine$
    $s \leftarrow$ random generated by $gndEngine$
return s

non-monotonic expressions and may not have solutions for the inverse cumulative distribution function.

Kernel Density Estimation (KDE) is another statistical approach to estimate distribution that isclosely related to the histogram of sample data and can generate discrete values [6]. However, KDE would smooth the peak frequent samples even with a large number of bins, as shown the curve in Figure 8(a). The resulting KDE distribution curve smooths the effect of peak probability in the input timing histogram, and flattens timing samples with the highest frequency density. However, these frequently occurring execution times are important for accurate timing analysis, because they represent crucial scenarios during execution. Furthermore, because the bin size of KDE affects the number of regenerated timing samples and the resulting smoothing, re-fitting is required whenever a different number of regenerated timing samples is needed by the simulation. Thus, KDE is not a flexible fitting method for the RAD simulation framework.

In this article, we present a weighted-joint normal distribution fitting that fits timing samples into separate normal distributions, thereby maintaining timing accuracy for execution with peak frequency density. The data regeneration algorithm (Algorithm 4) is based on the timing data histogram and seeks to fit the samples into multiple normal distributions. The algorithm divides the timing histogram into intervals based on peak frequency density, effectively determining a



(a) Nullification rate $\gamma = 0.4$ (b) Nullification rate $\gamma = 0.6$ (c) Nullification rate $\gamma = 0.7$

Fig. 9. Detection rate with decreasing number of monitored control flow events, using minimum range, minimum variance, maximum peak probability methods.

set of peaks and valleys. Each interval is then independently fitted to a normal distribution. To determine interval, the algorithm finds timing samples, μ, that have the peak frequency density, y, amongst μ's neighbors. An interval with a higher y results in a narrower bound, and each interval does not have overlap.

The random regeneration of sample s involves three steps. The first step is randomly deciding which interval is used based on a weighted discrete distribution, where the weight of each interval is the sum of the frequency for all time samples in this interval. The second step is randomly generating a timing s based on the normal distribution of that interval. The last step is filtering out any timing samples s that are not located within the interval from which s was generated, thus eliminating the problem of distribution tail overlap and adhering to timing boundaries of input samples.

Figure 8(b) presents the normalized histogram of 2,000 samples of the same event regenerated using the weighted-joint normal distribution fitting. To evaluate the accuracy, we used Chi-Square goodness of fit test, which is commonly used to compare regenerated data and expected data. The sum of squared error from weighted-joint normal fitting is 31.7, which is less than 32.5 from KDE fitting. That is because the algorithm fits with more subdivisions of the raw sample data and maintains each peak frequency density. More accurate data fitting algorithm may be utilized and embedded into the simulation in future work, but is beyond the scope of this article.

5.4 Malware Detection Results

In practice, the size of the reduced RSM configured within the RAD hardware is constrained by the size of the RAD hardware manufactured within the SOC. Thus, we analyze the performance of RAD detection for different constraints on the number of monitored events within the reduced RSM.

5.4.1 File Manipulation Malware. Figure 9 presents the detection rate of *File Manipulation* mimicry malware with RAD detection, constrained on the number of monitored events, ranging from 15 down to 6 events, for the three considered event selection methods: (1) minimum range, (2) minimum variance, and (3) maximum peak probability. Detection rates are based on executing the system with the mimicry malware 100 times. Note that among the least six monitored events, four of them monitors timing requirements in Table 2 and they are included by all metrics.

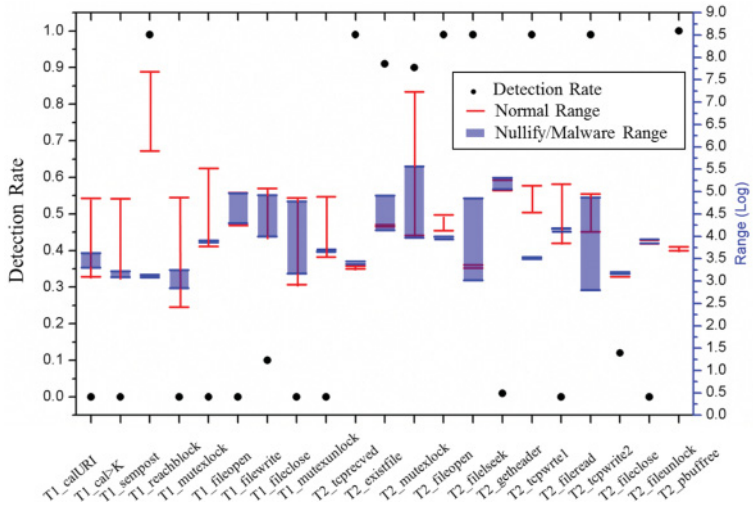


Fig. 10. Single detection rate for each event with nullified arguments in the calculation and communication tasks. The *fileopen, fileread, filewrite*, and *fileclose* events are not shown as these events are not nullified.

Figure 9(a) presents the detection rate for a nullification rate $\gamma = 0.4$. This nullification rate is the minimum nullification rate required to successfully implement the File Manipulation malware. In other words, it is the most sophisticated mimicry implementation. When the RAD hardware supports more than 15 events, all selection methods achieve a 100% detection rate. When the number of monitored events decreases, the detection rate decreases. With a constraint of 14 monitored events, only the minimum range and minimum variance selection methods achieve a detection rate greater than 88%, compared to a detection rate of less than 40% for the maximum peak probability method.

For a nullification rate $\gamma = 0.6$, the minimum variance and minimum range selection methods achieve a detection rate of 100% with 12 monitored events. As the nullification rate increases, comparing the results in Figure 9(a), (b), and (c), the RAD's detection rate increases. With a nullification rate $\gamma = 0.7$, all event selection methods achieve a detection rate greater than 99% with 11 monitored events. Overall, the minimum variance and minimum range event selection methods perform well, but neither of these methods shows a clear advantage over each other as the available number of monitored events decreases. However, the results demonstrate that maximum peak probability does not perform well. For nullification rates $\gamma = 0.4$ and 0.6, the detection rate falls below 40% when monitoring 14 or fewer events, compared to greater than 88% for the other event-selection methods. The worse performance of the peak probability metric may be caused by fluctuations in the timing distribution. For example, the event with timing distribution shown in Figure 4 has multiple close peaks with similar probability, although timing range of this event is concentrated, but the maximum peak probability is not high, so this event would not be selected.

The execution time of some events is greatly impacted by nullification, and they are potentially better events to monitor. For example, in Figure 9(a) and (c), for all three metrics, the events that lead to larger increases in the detection rate (i.e., the inflection points) are *T2_fileopen* and *T2_tcprecved*, respectively. Therefore, we further analyzed the effect of nullification on each event within the RSM. Figure 10 presents the normal execution time range, nullified/malware execution time range, and individual detection rate for events within the RSM. Note that the events for *fileopen, fileread, filewrite*, and *fileclose* are not nullified, as those events are the malicious

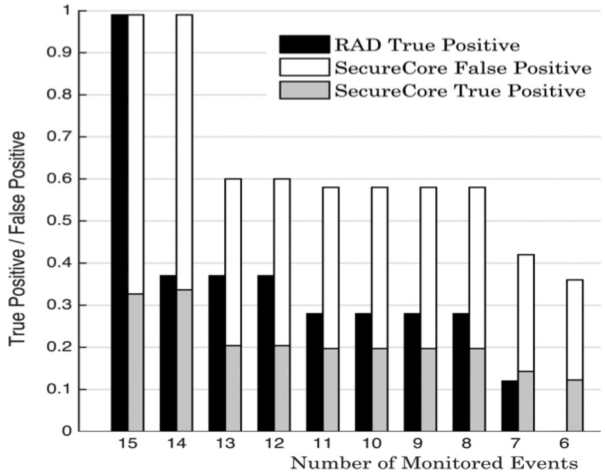


Fig. 11. True positive and false positive rates for RAD and SecureCore for different number of monitored events.
Note: For the training size considered, RAD has zero false positives.

events required by the malware for the intended malicious activity. It can be observed that for *T2_fileopen* and *T2_tcprecved*, the malicious nullified events lead to non-overlapping timing distributions, which are easier to detect. For other events, such as *calURI*, nullification has almost no impact on the timing behavior. Therefore, analysis of the possible methods to nullify individual events have the potential to enable better event-selection methods, although that is left as future work.

We also compare the detection rate and false positive rate of the SecureCore method [36] for the network-connected pacemaker application using the same configuration and subsets of monitored events for the File Manipulation malware. For the SecureCore approach, to ensure a uniform distribution for each bin, a bin size of 10 cycles is required. Figure 11 presents the true positive and false positive rates of SecureCore and RAD for decreasing number of monitored events for 100 executions of the application. Because SecureCore purposely marks bins with low probability (i.e., less than 1%) as potentially malicious, the detection rate achieved can be classified as true positives (i.e., detection times that do not overlap with normal execution) and false positives (i.e., detection times that overlap the normal execution). SecureCore false positive rate ranges from 13.5% to 66.33%, and the true positive rate ranges from 13.5% to 32.67%. In contrast, because the RAD approach maintains the complete timing range for events, with 8 or more monitored events, the RAD achieves a true positive rate ranging from 28% to 99%, which is an increase of 14.5% to 66.33% compared to SecureCore. Further note that the memory requirements for RAD configuration is significantly less than SecureCore, which requires storing thousands of bins for each events.

5.4.2 Information Leakage Malware. Figure 12 presents the detection rate of the *Information Leakage* mimicry malware with RAD detection for the three considered event selection methods. We observe the same trend in the detection result as that of the File Manipulation malware in Figure 9. The minimum range and minimum variance selection methods perform better than maximum peak probability selection method when monitoring less than 23 events. Overall, the minimum variance selection method performs best, achieving a 100% detection rate for 19 or more events.

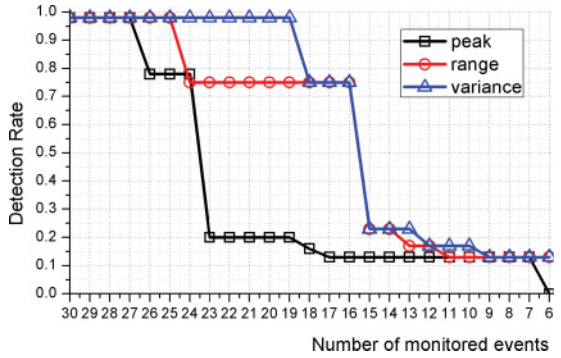


Fig. 12. Detection rate of Information Leakage malware.

5.4.3 Fuzz Attack Malware. For the Fuzz Attack malware, we implemented two different fuzz malware, namely *fuzz buffer size* and *fuzz physician configuration*, with three different fuzz variance levels, including 10%, 20%, and 100% variances. The fuzz buffer size malware fuzzes the buffer size of the cardiac log, which affects all data read from or written to the log file. The fuzz physician configuration malware fuzzes the configuration data from physician, which affects all operations related to communication, device configuration, and pacing control. The detection rates for these two fuzz malware are not as good as the detection rates for the File Manipulation and Information Leakage malware. For the fuzz buffer size malware with a fuzz variance of 100%, RAD achieves a 79% detection rate when monitoring 30 events. For the fuzz physician configuration malware with a fuzz variance of 100%, RAD achieves a detection rate of only 2% when monitoring 13 events. There are several reasons that detection rate of Fuzz Attack is not as high as the detection rate for other attacks. First, events being affected by a single fuzz malware are very limited, since the fuzz attack's complex aim is to inject very small perturbations that can disrupt the execution. In addition, some of the events that are fuzzed in the malware are not executed frequently. For example, the fuzz buffer size malware can be detected from the *T1_filewrite* event, but this event is executed infrequently. This observation indicates that selecting events using a path coverage metric as part of the selection method would yield improved detection rates, although this analysis is left as future work.

Figure 13 further presents the normalized timing histogram of a single event for the two fuzz malware. Figure 13(a)–(d) compares the timing histogram of event *T1_filewrite* for the normal execution and the fuzz buffer size malware execution with fuzz variances of 10%, 20%, and 100%. Although detection rates for this single event are only 9%, 10%, and 23%, respectively, for three fuzz variances, the distributions are very different. We utilized the Kolmogorov–Smirnov test (K-S test) with $\alpha = 0.5$ to analyze the difference in the distribution for event *T1_filewrite*. The K-S test is nonparametric test used to compare two sample distributions, with input significance level parameter α and result [k, p]. For example, if $\alpha = 0.05$, [k, p] = [1,0.00001] means the null hypothesis is rejected because the p-value is less than α, indicating the two distributions are very different. The K-S test for the fuzz buffer size malware yields the result that [k, p] = [1, 3.47E-88], [1, 5.23E-108], and [1, 3.36E-234], respectively, which indicates high confidence that fuzz malware result in different timing distribution compared to the normal system execution. Figure 13(e)–(h) compares the timing histogram of event *T2_docconfig* for the normal execution and the fuzz physician configuration malware execution with fuzz variances of 10%, 20%, and 100%. Although the detection rate of this single event is under 5%, the distributions are also different. The K-S test with $\alpha = 0.05$ results in [k, p] = [1, 3.07E-19], [1, 3.18E-08], and [1, 9.94E-36], respectively. These results demonstrate the potential benefit of statistical analysis in time-based anomaly detection.

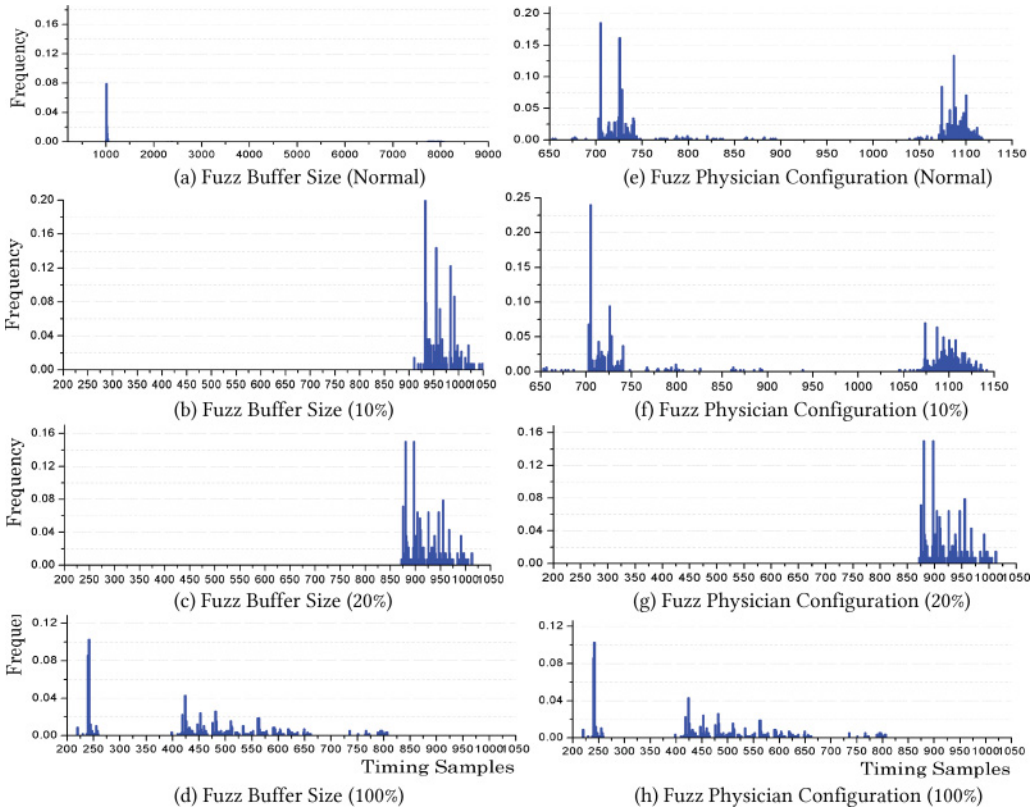


Fig. 13. Normalized histogram comparison between normal timing and fuzz timing. (a)-(d) Timing histogram of fuzz buffer size of *T1_filewrite* (e)-(h) Timing histogram of fuzz configuration of *T2_doc_config*.

5.4.4 Control Loop Delay and Denial of Service Malware. The *Control Loop Delay* malware delays the pulse time of the pacer, which threatens the safety of the patient. This malware can be detected by monitoring the timing requirement Tpulse specified in Table 2. For a pulse delay of 2ms, 5ms, and 10ms, the detection rate is 2%, 93%, and 100%, respectively.

The DoS attack is executed by sending 800 packets/s to port 80, which is responsible for communicating with the physician. 800 packets/s is the minimum packet rate required to deny the physician's access to retrieve the patient's cardiac log using the 100MHz pacemaker prototype. The number of monitoring events needed for maximum peak method, minimum range method, and minimum variance method to detect DoS is 7, 6, and 6, respectively.

5.4.5 Interrupt Disable Malware. The *Interrupt Disable* malware disables single or multiple interrupts in the application, thereby disabling the pacemaker's critical functionality. If the malware disables ISR3 for the AVI timer, the ventricular pacer would be disabled, and the patient cannot receive a life-saving ventricular pulse. This malware can be detected by monitoring system-level sequence and timing requirements, specified in Section 5.1. For example, disabling ISR1 would break the sequence requirement (1) and (3), and break timing requirement (5). Because all these requirements are monitored in default for RAD, the Interrupt Disable malware can be easily detected. To evaluate the mean time-to-detection (i.e., time from when the interrupt is disabled to the moment RAD detects the anomalous execution), we simulated a heart with three behavioral

Table 4. Mean Time to Detection (MTTD) (ms) for the *Interrupt Disable* Malware for Multiple Heart Patterns

Interrupt Disabled	Detection Scenarios					
	Healthy		Unhealthy1		Unhealthy2	
	MTTD (ms)	*[min,max]*	*MTTD (ms)*	*[min,max]*	*MTTD (ms)*	*[min,max]*
ISR 1	817.42	[591,991]	660.419	[1,1002]	691.039	[1,992]
ISR 2	Inactive	Inactive	Inactive	Inactive	634.059	[1,992]
ISR 3	Inactive	Inactive	248.785	[1,411]	Inactive	Inactive
ISR 4	310.10	[151,411]	217.165	[1,411]	954.364	[1,992]

ISR2 (VAI timer) is inactive when heart pattern is Healthy and Unhealthy1; ISR3 (AVI timer) is inactive when heart pattern is Healthy and Unhealthy2.

patterns. The first is a healthy heart pattern that won't activate any pacer interrupts, the second is an unhealthy heart pattern that activates the AVI interrupt (ISR3), and the third is an unhealthy heart pattern that activates the VAI interrupt (ISR2).

Table 4 presents mean time-to-detection in ms of disabling one interrupt for 50 random insertions of the Interrupt Disable malware for each pattern. [min, max] is theoretical detection time bound based on the timing requirement. The RAD can detect violation of timing requirement one cycle after internal timer expires, thus minimizes time-to-detection. Table 4 demonstrates the ability of RAD to detect Interrupt Disable malware, in which there is enough slack on average to mitigate the timing violates. Disabling more than one interrupt shortens the time-to-detection because more system requirements are violated.

6 CONCLUSIONS AND FUTURE WORKS

We presented a timing-based runtime anomaly detection methodology RAD for non-intrusively detecting mimicry malware within embedded systems. Using a network-connected pacemaker prototype, we demonstrated that the RAD approach achieves a 100% detection rate for mimicry malware using as few as 14 monitored events. The RAD approach can also detect other malware types, including fuzz, control loop delay, and interrupt disable malware, as well as indirectly detecting DoS attacks. Compared with SecureCore, which also uses timing to detect malware, RAD achieves up to a 66.33% greater true positive rate, while using significantly less memory for the configuration data.

Future work includes determining to what extent even small malicious changes can be detected by observing changes in execution time or execution time distribution. We plan to develop automated tools to explore different methods to nullify individual events, analyzing the changes in time distributions, and selecting events that are empirically harder to nullify. We anticipate this work may yield better detection of Fuzz Attacks. We will also explore subcomponent timing-based detection methods that analyzes different components of the software execution timing, including intrinsic timing of the instructions, cache misses, memory accesses, pipeline stalls, context switches, and so on. It is expected that analyzing the subcomponent timing will improve detection accuracy, as mimicking the execution time at such a fine-grained level will require significantly greater sophistication for mimicry malware. To properly determine the energy overheads of integrating the RAD hardware within an SOC, future work includes evaluating the energy consumption of the RAD hardware using an ASIC or full-custom implementation.

REFERENCES

[1] ARM. 2011. Embedded Trace Macrocell ETMv1.0 to ETMv3.5 Architecture Specification.

[2] D. Arora, S. Ravi, A. Raghunathan, and N. K. Jha. 2005. Secure embedded processing through hardware-assisted runtime monitoring. design. In *Automation and Test in Europe Conference*, (March 2005), 178–183.

[3] D. Arora, A. Raghunathan, S. Ravi, and N. K. Jha. 2006. Architectural support for safe software execution on embedded processors. In *Conference on Hardware Software Co-design and System Synthesis*, (Oct. 2006), 106–111.

[4] S. Bhatkar, A. Chaturvedi, and R. Sekar. 2006. Dataflow anomaly detection. In *Symposium on Security and Privacy*, (May 2006), 15–62.

[5] M. Bond, V. K. Srivastava, K. McKinley, and V. Shmatikov. 2010. Efficient, context-sensitive detection of real-world semantic attacks. *Programming Languages and Analysis for Security*, (June 2010), 1–10.

[6] Z. I. Botev, J. F. Grotowski, and D. P. Kroese. 2010. Kernel density estimation via diffusion. *Annals of Statistics*. 38, 5 (2010), 2916–2957.

[7] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. Iyer. 2005. Non-control-data attacks are realistic threats. In *USENIX Security Symposium*, (July 2005), 177–192.

[8] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. 2010. Flexible and efficient instruction-grained run-rime monitoring using on-chip reconfigurable fabric. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. (Dec. 2010), 137–148.

[9] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull. 2002. Graphviz – Open source graph drawing tools. In *Graph Drawing*. Springer, 2002, 483–484.

[10] Federal Financial Institutions Examination Council (FFEIC). Cyberattacks on Financial Institutions' ATM and Card Authorization Systems. https://www.ffiec.gov, 2014.

[11] A. Frossi, F. Maggi, G. Rizzo, and S. Zanero. 2009. Selecting and improving system call models for anomaly detection. In *Conference on Detection of Intrusions and Malware, and Vulnerability*, (July 2009), 206–223.

[12] D. Gao, M. Reiter, and D. Song. 2003. Gray-box extraction of execution graphs for anomaly detection. In *ACM Conference on Computer and Communications Security*, (Oct. 2003), 318–329.

[13] N. Idikaand A. P. Mathur. 2007. *A Survey of Malware Detection Techniques*. Technical Report, Purdue University, (2007).

[14] Z. Jiang, M. Pajic, S. Moarref, R. Alur, and R. Mangharam. 2012. Modeling and verification of a dual chamber implantable pacemaker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, (March 2012), 188–203.

[15] C. Liu, C. Yang, and Y. Shen. 2014. Leveraging microarchitectural side channel information to efficiently enhance program control flow integrity. In *Hardware/Software Codesign and System Synthesis Conference* (Oct. 2014). 1–9.

[16] S. Lu, M. Seo, and R. Lysecky. 2015. Timing-based anomaly detection in embedded systems. In *Asia South Pacific Design Automation Conference* (Jan. 2015). 809–814.

[17] S. Lu. and R. Lysecky. 2015. Analysis of control flow events for timing-based runtime anomaly detection. In *Workshop on Embedded Systems Security* (Oct. 2015).

[18] S. Mao and T. Wolf. 2010. Hardware support for secure processing in embedded systems. *IEEE Transactions on Computers*, 59, 6, 847–854.

[19] E. Marin, D. Singeleé, B. Yang, I. Verbauwhede, and B. Preneel. 2016. On the feasibility of cryptography for a wireless insulin pump system. In *ACM Conference on Data and Application Security and Privacy* (March, 2016). 113–120.

[20] R. A. Maxion and K. M. C. Tan. 2002. Anomaly detection in embedded systems. *IEEE Transactions on Computers*. 51, 2, 108–120.

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

[22] C. McCarthy, K. Harnett, and A. Carter. 2014. *Characterization of Potential Security Threats in Modern Automobiles: A Composite Modeling Approach*. National Highway Traffic Safety Administration, Washington Tech. Rep, (Oct. 2014).

[23] MicroBlaze. 2009. Microblaze processor reference guide embedded development kit EDK 11.4. 102–104.

[24] J. Mu, K. Shankar, and R. Lysecky. 2013. Profiling and online system-level performance and power estimation for dynamically adaptable embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)* 12, 3, Article 85, 1–20, 2013.

[25] K. Patel and S. Parameswaran. 2008. SHIELD: A software hardware design methodology for security and reliability of MPSOCs. In *Design Automation Conference* (June 2008), 858–861.

[26] K. Patel, S. Parameswaran, and R. Ragel. 2010. Architectural frameworks for security and reliability of mpsocs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 99, 1–14.

[27] M. Prates, V. H. Lachos, and C. R. B. Cabral. 2011. mixsmsn: Fitting finite mixture of scale mixture of skew normal distributions. R package version 0. 2-9

[28] M. Rahmatian, H. Kooti, I. Harris, and E. Bozorgzadeh. 2012. Hardware-assisted detection of malicious software in embedded systems. *IEEE Embedded Systems Letters (ESL)*, 4, 4, 94–97.

[29] M. Ramilli and M. Prandini. 2012. Always the same, never the same. *IEEE Security & Privacy*, 8, 2, 73–75.
[30] M. I. Sharif, K. Singh, J. T. Giffin, and W. Lee. 2007. Understanding precision in host based intrusion detection. In *International Symposium on Research in Attacks, Intrusions and Defenses*. 4637, 21–41.
[31] N. K. Singh, A. J. Wellings, and A. L. C. Cavalcanti. 2012. The cardiac pacemaker case study and its implementation in safety-critical java and ravenscar ada. In *International Workshop on Java Technologies for Real-time and Embedded Systems* (Oct. 2012). 62–71.
[32] Slowloris HTTP DoS. http://Ha.ckers.org/slowloris/, 2014.
[33] N. Stollon. 2011. *On-Chip Instrumentation: Design and Debug for Systems on Chip*. Springer US, 2011.
[34] D. Wagner and P. Soto. 2002. Mimicry attacks on host based intrusion detection systems. In *ACM Conference on Computer and Communications Security* (Nov. 2002). 255–264.
[35] M. K. Yoon, S. Mohan, J. Choi, and L. Sha. 2015. Memory heat map: Anomaly detection in real-time embedded systems using memory behavior. In *Design Automation Conference* (June 2015), 1–6.
[36] M. K. Yoon, S. Mohan, J. Choi, and L. Sha. 2013. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *Real-Time and Embedded Technology and Applications Symposium* (April 2013), 21–32.
[37] T. Zhang, X. Zhuang, S. Pande, and W. Lee. 2005. Anomalous path detection with hardware support. In *Conference on Compilers. Architectures and Synthesis for Embedded Systems* (Sep. 2005), 43–54.
[38] C. Zimmer, B. Bhat, F. Mueller, and S. Mohan. 2010. Time-based intrusion detection in cyber-physical systems. In *ACM/IEEE International Conference on Cyber-Physical Systems* (April 2010), 109–118.

Received November 2016; revised April 2017; accepted July 2017