

SecureCore: A Multicore-based Intrusion Detection Architecture for Real-Time Embedded Systems

Man-Ki Yoon*, Sibin Mohan†, Jaesik Choi*‡, Jung-Eun Kim*, and Lui Sha*

*Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801

†Information Trust Institute, University of Illinois at Urbana-Champaign, Urbana, IL 61801

‡Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720

Email: {mkyoon, †sibin, jekim314, lrs}@illinois.edu, ‡jaesikchoi@lbl.gov

Abstract—Security violations are becoming more common in real-time systems – an area that was considered to be invulnerable in the past – as evidenced by the recent W32.Stuxnet and Duqu worms. A failure to protect such systems from malicious entities could result in significant harm to both humans as well as the environment. The increasing use of multicore architectures in such systems exacerbates the problem since shared resources on these processors increase the risk of being compromised. In this paper, we present the *SecureCore* framework that, coupled with novel monitoring techniques, is able to improve the security of real-time embedded systems. We aim to detect malicious activities by analyzing and observing the inherent properties of the real-time system using statistical analyses of their execution profiles. With careful analysis based on these profiles, we are able to detect malicious code execution as soon as it happens and also ensure that the physical system remains safe.

I. INTRODUCTION

Many safety-critical systems such as advanced automotive/avionics systems, power plants and industrial automation systems have traditionally been considered to be invulnerable against software security breaches.¹ This was particularly the case since, in general, such systems are physically isolated from the outside world and also used specialized protocols. However, many recent successful security attacks on embedded control systems such as the W32.Stuxnet infection of Iran’s nuclear power plant [36], malicious code injection into the telematics units of modern automobiles [19] and attacks on UAVs [30] call for a rethink of the security of safety-critical embedded systems.

Another recent trend is that of multicore processing. Such processors are finding wide use in a variety of domains and embedded systems are no exception. The increase in performance, reduction in power consumption, and reduced sizes of systems using multicore processors (a single board instead of multiple boards) makes them very attractive for use in safety-critical embedded systems. A problem with the use of multicore processors in such systems is that of *shared resources* – components such as caches, buses, memory, *etc.*, are shared across

This work is supported in part by grants from Rockwell Collins RPS#6 45038, Lockheed Martin 2009-00524, NSF A17321, and ONR N00014-12-1-0046. Jaesik Choi was supported by DOE ANL 9J-30281-0015A and DE-AC02-05CH11231. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

¹A *safety-critical* or *life-critical* system is one where failure or malfunction may result in death or serious injury to humans, loss or severe damage to equipment and/or the environment.

the multiple cores and could result in security vulnerabilities [25]. For example, malicious entities could snoop on privileged information used/generated by critical code running on alternate cores, high-priority tasks could be prevented from executing by a denial-of-service attack on the shared resources (*e.g.*, keeping the bus occupied by large DMA transfers could prevent the high priority task from obtaining the memory reads it requested), *etc.* Hence, there is a need for a comprehensive solution where multicore processors could be used in safety-critical systems in a *safe* and *secure* manner. In fact, the very nature of such processors – the parallel cores and the convenience they provide – could be used to improve the overall security of the system.

In this paper, we present *SecureCore*, a secure and reliable multicore architecture solution to tackle security vulnerabilities in real-time embedded systems. We specifically pursue an approach to entrusting certain CPU cores in a multicore processor with the role of monitoring and intrusion detection.² The use of multicore processors has inherent advantages over off-chip security devices: (*i*) a CPU core is able to more closely monitor the execution behavior of software running on the other (potentially) unsecured core(s); (*ii*) the mechanisms cannot be tampered with easily or reverse-engineered. Section III provides further details about the SecureCore Architecture.

We also introduce novel techniques to observe inherent properties of the real-time code executing on the monitored core in Section IV – properties such as execution time for instance. These properties tend to be fairly deterministic in such real-time systems and hence can be used as a way of detecting anomalous behavior (indicative of malicious activity). These observations, in conjunction with the capabilities of the SecureCore architecture, significantly increase the security of the overall system by enhancing the abilities to detect intrusions. The key idea behind the proposed architecture and intrusion detection mechanism is that since real-time embedded control applications generally have regular timing behavior, an attack would inevitably alter its run-time timing signature from expected values [22].

Our architecture proposes a design so that a trusted entity, a *secure core*, can continuously monitor the run-time execution behavior of a real-time control application on an untrustworthy entity (henceforth referred to as the *monitored core*), in a *non-*

²For this paper we will focus on the use of a dual-core processor setup where one core observes the other one. In future versions, we intend to study the tradeoffs regarding how many monitoring cores are required per set of observed cores.

intrusive manner. In case malicious behavior is detected, a reliable backup control application residing on the secure core takes control away from the infected core in order to guarantee stability and loss-less control for a physical system [28]. Since there will be some inherent variability in these properties – for instance due to changes in inputs, code complexity, *etc.*, we use a statistical learning-based mechanism for profiling the *correct* execution behavior of a sanitized system.

In summary, this paper implements the following: (a) a novel architecture based on a multicore platform that provides security mechanisms for use in embedded real-time systems; (b) execution time-based intrusion detection mechanisms using a statistical learning; and (c) Simplex [28] architecture-based reliability. All of these combined provides non-intrusive, invisible monitoring capabilities and reliable, seamless control for real-time systems.

A. Assumptions

In this paper, the following assumptions are made without loss of generality: (i) We consider a CPU-based real-time control application – *i.e.*, a system consisting of periodic, independent tasks. (ii) We assume the application runs on a single monitored core. The proposed intrusion detection method does not work with multiple monitored entities in the current form. (iii) We assume that the size of the input set (to the control application under consideration) is small. This can be justified by the fact that most real-time control applications have a small footprint for input data (velocity, angle, *etc.*) within fairly narrow ranges. (iv) We assume that the execution time of the application is not unbounded. For example, the upper bounds for loops is known a priori. However, this assumption is not strictly required in this paper. It is sufficient to assume that (almost) all possible loop bounds are profiled. (v) Similarly, we assume there is no hidden execution flow path in the application – all paths are present when being profiled.

II. MOTIVATION

A. Threat Model

The W32.Stuxnet worm [36] was able to successfully subvert operator workstations and gain control of Iran’s nuclear power plants through sophisticated attacks including the first known PLC rootkits and use of multiple zero-day vulnerabilities. The worm was able to intrude into the control system by first gaining access and then downloading attack code from a remote site. The malware then gradually inflicted damage to the physical plant by substituting infected actuation commands for legitimate ones over a period of time. Despite the employment of several protection and monitoring mechanisms, the control system could not detect the intrusion and the attack until the physical damage to the plant was significant. In fact, such sophisticated systems have many entry points that are vulnerable to potential attacks and they often cannot be secured completely. Hence, there is need for failure-prone monitoring methods.

In this paper, instead of trying to prevent and/or detect intrusions at every vulnerable component, we intend to monitor and detect intrusions at the most critical component: in real-time control systems, the primary concern is the safety of the physical

plant under control. Thus, we focus on detecting an intrusion that directly targets the real-time control application. We assume that regular security process was in place to ensure the security during the application design and development phases, *i.e.*, the application is trustworthy initially. The execution timing model is obtained via profiling prior to system deployment. Also, the timing information is obtained by re-profiling the system after any updates and is supplied with the modified application (if any). We assume that the application could be compromised after the profiling stage but the stored timing profile cannot be tampered with during the updating process. We consider malicious code that can be secretly embedded in the application, either by remote attacks or during upgrades. The malicious code activates itself at some point after the system initialization and then gradually tries to change, damage, or even snoop on the physical state of the plant under control. We are not directly concerned with *how* the malicious code gained entry, but more concerned with what happens after that.

B. Use of Multicore Processor in Real-Time Control Systems

Multicore processors are receiving wide attention from industries due to their ability to support generic and high-end real-time applications that traditional control hardware, *e.g.*, programmable logic controllers (PLC), are unable to provide. This trend is especially strong for instance in automotive industries [1] where CPU-based real-time control applications have a significant presence, *e.g.*, engine control, anti-lock braking systems (ABS), *etc.* As previously introduced, it was shown that automotive control applications are increasingly vulnerable to security attacks as they are more equipped with high-end and complex technologies [19]. Although we do not specifically consider automotive control applications, the use of the mechanisms presented in this paper will naturally fit into the future development processes of safety-critical automotive components as the industries are more moving toward employing more multicore-based real-time control systems. Also, the use of one or more cores for improving the security (and overall safety) of such systems is a big plus. Even though some of the resources (cores in this case) are being used up, the increase in security that is provided as a result definitely offsets any losses in performance. Hence, the use of multicore processors in secure real-time embedded systems will be beneficial to the community.

III. SECURECORE ARCHITECTURE

In this section, we present the *SecureCore* Architecture, a secure and reliable multicore architecture that aids in the detection of intrusions in embedded real-time systems and guarantees a seamless control to the physical system. We first introduce the overall structure of the architecture and then discuss the design consideration of each component in detail.

There exist several challenges in both hardware as well as software, before these techniques could be implemented in a satisfactory manner. First, a protection mechanism must be provided to the secure core so that it is tamper-resistant (especially from malicious activity on the unsecured/monitored cores). Second, the secure core should be able to closely monitor the state of the other core. However, the monitoring activity

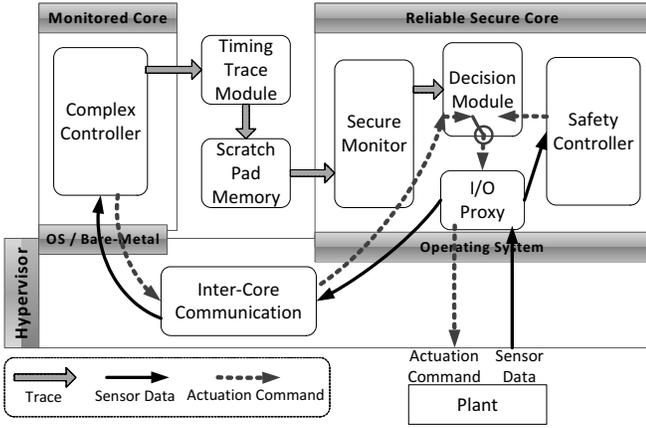


Fig. 1. SecureCore Architecture.

should be invisible as far as the observer is concerned – this is mainly so that an attacker should not be able to deceive the intrusion detection mechanisms by means of replay attacks (*i.e.*, replicating previously recorded execution behavior of an application in its correct state). Third, in a multicore environment, an application will inevitably experience a considerable variation in its execution time due to the interference caused from inter-core resource contentions [27], [39], [40]. Thus, the security invariant, *i.e.*, the execution time profile in this paper, should be accurate enough so that the intrusion detection method(s) can effectively validate the cause of any such variations. Similarly, the method should be able to take into account execution time variation caused by legitimate application contexts such as differences in input sets and execution flow. Finally, the secure core should be able to guarantee loss-less control to the physical system that it manages even if the main, monitored, or control application is compromised. How we solved these problems is elaborated in the following sections.

A. High-Level Architecture

Figure 1 shows the high-level structure of the SecureCore architecture. The system is composed of four major components – (a) the secure core, (b) the monitored core, (c) the on-chip Timing Trace Module (TTM) and (d) the hypervisor. The system is built upon the concept of the Simplex architecture [28]: a safety controller and a decision module rest on the secure core while a *complex controller* (essentially the controller that manages the physical system) runs on the monitored core. Sensor data from the physical plant is fed to the both controllers, each of which computes actuation commands using their own internal control logic. The decision module on the secure core then forwards the appropriate command to the plant depending on a pre-computed *safety envelope* for the physical system. In normal circumstances, the plant is actuated by commands from the complex controller. However, when an abnormal operation of the complex controller is detected (say, due to unreliable/erroneous logic and faults), control is transferred to the safety controller in order to maintain loss-less actuation of the physical plant. With this mechanism, the reliability of the control actions can be guaranteed by the decision module and the safety controller (that can be formally verified), provided however that all the entities are trustworthy. It is possible for the decision module or safety controller to be compromised by a security attack.

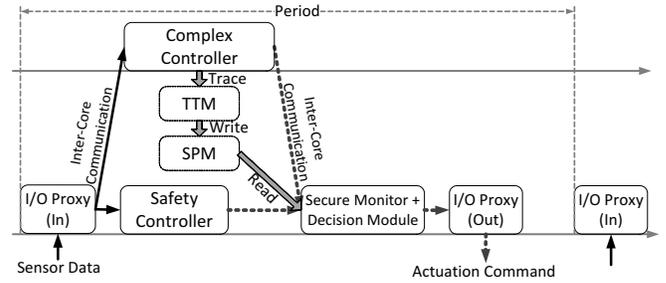


Fig. 2. Execution flow of the SecureCore components.

Furthermore, the complex controller may deceive the decision module by providing a legitimate actuation value while, for example, collecting critical system information that could be exploited during a future attack. Thus, it is important to ensure a high security level for the system. In the following sections, we describe how the security as well as the reliability of this basic Simplex mechanism can be enhanced by use of the SecureCore architecture.

B. Design Considerations

Our solution includes a hypervisor that provides a *virtualization* of hardware resources on our proposed SecureCore architecture through *partitioning* and *consolidation* [4]. In order to protect the secure core from malicious alteration by a compromised complex controller, the hypervisor provides a clean separation of memory spaces by programming the memory management unit (MMU). Also, the hypervisor itself runs in its own protected memory space. Thus, any attempts at memory access across the partitions is blocked by hypervisor.

With the help of this memory protection, we design an I/O channel between the processor and the plant. The channel is managed by an *I/O proxy* process that runs on the secure core. The I/O proxy manages all I/O to and from the physical plant. This is to prevent I/O data obfuscation that could be caused by malicious code on the monitored core. Furthermore, if the I/O channel device is directly accessible by both cores then a compromised application on the monitored core may attack the secure core indirectly by, say, causing a denial-of-service attack on the I/O channel – this will prevent the safety controller from taking over from the complex controller. This I/O device consolidation capability is also provided by the hypervisor through the *I/O MMU*. The system is configured such that the device cannot be seen from the monitored core.

Since the memory space is partitioned and the I/O device is consolidated to the secure core, data to and from the monitored core is relayed via the *inter-core communication* channel on the hypervisor level. Transferring data through a shared memory region is strictly prohibited because of a potential vulnerability [25]. As shown in Figure 2, the I/O proxy first retrieves sensor data from the plant and then transfers it to the two controllers. For the complex controller, the I/O proxy places the data on a dedicated channel between the memory space of the secure core and that of the hypervisor. The data is then copied to the buffer at the monitored core’s side. The complex controller retrieves the data by either polling or an interrupt-driven method. For the opposite direction, however, *i.e.*, if the complex controller wishes to send out actuation

commands and when the decision module wishes to retrieve such a command, it (the decision module) polls the buffer on the inter-core communication channel. The decision module also sets a watch-dog timer for this process. When the timer expires and the decision module has still not received data from the complex controller, the safety controller takes over the control. This polling-based data passing is to prevent the secure core from being unboundedly interrupted by a compromised complex controller – a vulnerability that can be exploited using an interrupt-driven method.

The main component that enforces the security invariant in the architecture is the *secure monitor* – a process that continuously monitors the execution behavior of the complex controller. The secure monitor works in conjunction with an on-chip hardware unit called the *Timing Trace Module (TTM)*. The details of secure monitor and TTM are discussed in Section III-C. The key role of the monitor is to detect if the run-time execution time signature has deviated from what is expected/has been profiled. If any unexpected deviations are observed then the secure monitor informs decision module and control is immediately switched over to safety controller in the secure core. At the same time, the hypervisor is told to reset the monitored core and then reload a clean copy of the complex controller binary from a secure memory region. Once the reset and reload is complete, the monitored core could, potentially, resume operation and take control back. Of course, this could depend on the policy for recovery that is implemented on the actual system. It could also happen that the monitored core is completely shut down and not restarted until an engineer analyzes the issue and says it is safe for the monitored core to resume operations. This will prevent smart attackers from triggering constant back and forth switches between the complex and simple controllers – events, that could themselves, cause harm to the physical system if timed correctly. In case a rapid recovery is required, the complex controller may even be implemented to run on a bare-metal executive [4] which, however, would require a modification of the legacy code.

As described above, the architecture relies heavily on the hypervisor. Thus, the entire secure mechanism can collapse if the hypervisor itself is compromised in the first place. Hence, it is assumed in this paper that the hypervisor forms part of the trusted base; no malicious code is embedded in it. We note that while a hardware-enforced memory protection mechanism would further enhance the security of the hypervisor [38], we do not address this issue in this paper.

One may argue why the monitored core does not receive the same protection as the secure core. The secure core needs to exist for two reasons: (i) fault tolerance when the main controller fails either due to a fault or a security violation and (ii) security; the monitored core might need to have software updates/interact with external sensors, perform I/O, etc., while the data/code in the secure core will not often change. Hence, it makes sense to *harden* the secure core and not always the monitored core. Of course, even if the monitored core is hardened, it can still be susceptible to smart attackers. Another issue is that the way the secure core is hardened is by providing it with private memory and other hardware mechanisms. Doing this for all other cores would be prohibitive, especially when we want to increase the

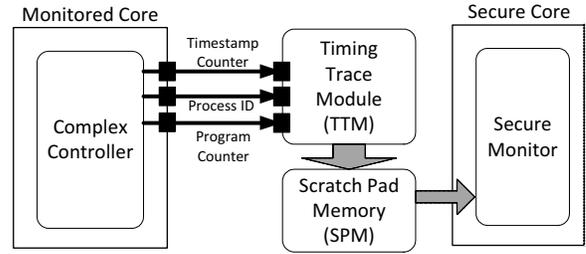


Fig. 3. Timing Trace Module.

observation to multiple cores.³

C. Timing Trace Module (TTM)

The Timing Trace Module (TTM) is a special on-chip hardware unit that traces the run-time timing information of the monitored core. The module is located between the monitored and secure cores and directly attached to the former as seen in Figure 3. When a certain event is triggered (the execution of a special instruction; explained shortly), a part of the processor state is read by the TTM. The processor's state includes the values of the *timestamp counter*, the *program counter (PC)* and the *process ID (PID)* of the current task. The trace information is then written to the scratch pad memory (SPM) that can be seen/accessed only by the secure core. The SPM is mapped to a range of the secure core's address space. A sequence of traces is collected during one single run of the complex controller (Figure 2). The secure monitor verifies the legitimacy of the execution profile obtained from the trace by comparing it with one that has been collected during implementation time when the system was in a known *good state*.⁴

We now present how TTM traces the required information from a running application. A trace operation is carried out by executing a special trace instruction in the monitored application, as described in Figure 4 (a). The special instruction also has a mode when it can register the PID of the monitored application with the TTM. Once a PID is registered, only a process that matches the PID can execute other trace instructions; this is to prevent traces from being forged by another process that might be compromised. The PID value is written at the top of the SPM and the PC value at that point is registered as the Base Address (BA) as shown in Figure 4 (b). When the trace instruction is executed while the tracing is enabled, the timestamp and the instruction address at that point execution are written at the address specified by the value of $\text{Addr}^{\text{Head}}$. Here, the address being written is a *relative* address from BA, i.e., $\text{PC}_i - \text{BA}$, that can contain positive or negative values. The reason for storing a relative address is to capture the *exact* signature of each trace, since the real addresses can change between executions – two sequences of traces may not match although they are produced at the identical places.⁵

Note: the TTM is used at *two* different points in the whole process: (i) during the development/testing phase, it

³This is part of our future research plan.

⁴The statistical learning-based profiling and monitoring methods will be explained in further detail in Section IV.

⁵We assume that no dynamically loaded libraries exist in the system and even if they do, we do not trace them.

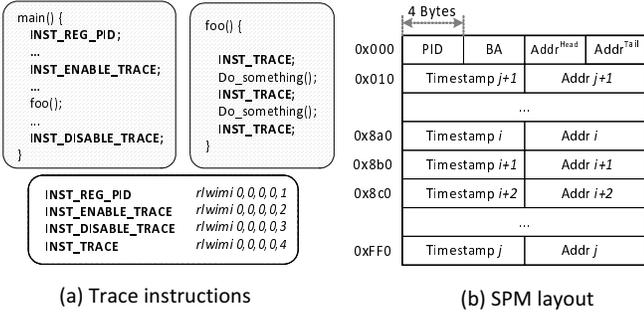


Fig. 4. Trace instructions and the layout of SPM.

is used to collect profiling information about the application processes/tasks, e.g., the real-time execution time profile described later in this paper and (ii) when the system is actually deployed in the field, the TTM is a conduit for flow of the monitoring information from the monitored core to the secure core. Meanwhile, the trace instructions are manually inserted into the code. Automated insertion of trace instructions is a future work.

The SPM is a circular buffer of traces. When a single run of the complex controller completes, the secure monitor consumes a sequence of traces specified by $\text{Addr}^{\text{Head}}$ and $\text{Addr}^{\text{Tail}}$. While it is possible for SPM buffer to overflow during execution, we note that only a small number of traces would be enough in a real-world control application due to a short span of the execution times. Also, we chose to use an SPM instead of shared memory (through a cache) as the buffer for the traces because an SPM has a lower access latency. The shared memory communication can also open up potential security breaches [25].

IV. GAUSSIAN KERNEL DENSITY ESTIMATION FOR EXECUTION TIME-BASED INTRUSION DETECTION

The intrusion detection method presented in this paper utilizes the deterministic timing properties of real-time control applications. Since any form of unwanted malicious activity consume finite time to execute, a deviation from expected regularity would likely point towards an intrusion. However, as explained in Section I, the execution time of an application can also include variations due to other, more mundane, reasons such as system effects. On a multicore processor, the sharing of hardware resources such as caches, buses, memory, etc., can result in variability in the execution times. Also, an application's own context such as different input sets and execution flows can cause deviations in timing. The main difficulty in profiling and estimating execution time comes from the fact that it is often non-parametric; e.g., monitoring only the mean, minimum, or maximum values is often not accurate enough for our purposes. Thus, in this section, we present a *statistical learning-based execution time profile and intrusion detection method* that can effectively validate the causes of any observed perturbations in execution time and account for their causes.

A. Overview

Let us first consider a simple example application consisting of three blocks of code (Figure 5 (a)).⁶ The blocks are sequen-

⁶A *Block* could refer to a sequence of instructions of arbitrary size and does not necessarily mean *Super Block* [15].

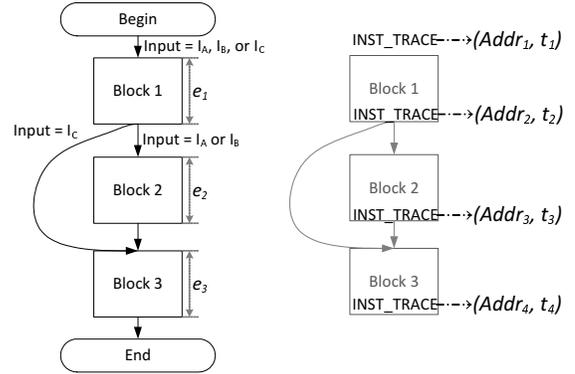


Fig. 5. Trace instructions inserted to an example application.

tially executed but depending on the input value(s) Block 2 may be skipped. Here, we do not assume a specific form of inputs – the input can be a single value, a range, or even multiple ranges of values. However, it should be assumed that the execution flow does not show deviations when presented with the same input.

The execution time profiling method (explained in Section IV-B) profiles the execution times of each *block* (measured in cycles) and generates an estimation on it. During run-time monitoring, each measured execution time of block i , e_i , is compared with the estimation, \hat{e}_i , to check how close it is to legitimate behavior. The reason we do not profile aggregated execution time is to improve the detection accuracy by narrowing the estimation domain. That is, each variation at every block gets accumulated along the execution path and this would obscure potential malicious code execution inside. For example, an attack code could redirect the execution (say using buffer overflows) during the execution of Block 2 and then return to the right address in a short amount of time. In such cases, the time taken by the extra code may fall within the interval of allowed deviations of aggregated execution time. Moreover, with block level monitoring, each block boundary can be used as a check point – the monitor can detect malicious execution along a path where a block is either skipped or never exited. Thus, an attacker would need to not only keep within fixed paths, but also complete execution in a very short amount of time – both of which significantly raise the bar against would be attackers.

B. Trace Tree

We now explain how traces generated by the TTM can be used to profile block execution times. Consider the execution flow graph in Figure 5 (a). Suppose we are interested in monitoring blocks between Begin and End. Then, we add an `INST_TRACE` instruction at the end of each block and at the top of the flow as shown in Figure 5 (b).⁷ Every time the instruction executes, a pair (Addr_i, t_i) is added as a trace (see Section III-C). This results in a sequence of traces for a single execution of the application – e.g., (Addr_1, t_1) , (Addr_2, t_2) , (Addr_4, t_4) , etc., for one input of I_C . Assuming each run of the application begins at the same entry point, we can construct a *trace tree* from a collection of such sequences as shown in Figure 6. In the tree, each edge corresponds to the address

⁷It should be noted that no `INST_TRACE` instruction must be placed *inside* a recursive function.

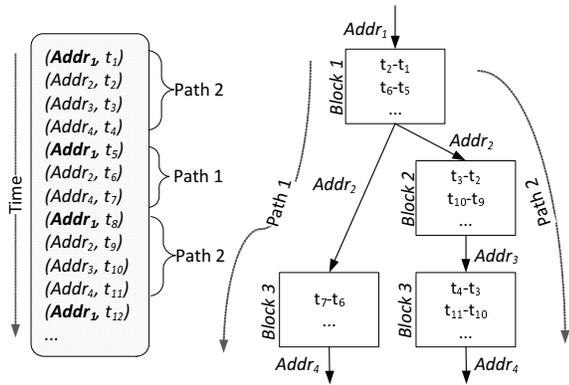


Fig. 6. Trace tree generated from a sequence of traces.

(relative from a base address) at which each `INST_TRACE` is executed. Thus, a block in the original execution flow graph can be defined as a pair $(Addr_p, Addr_c)$, where $Addr_p$ is the address of the last trace instruction that is executed before and $Addr_c$ is the address of the instruction that is executed right after the block. Accordingly, each node in the tree is a set of *time differences* between the two addresses that then are the *samples* of the block execution time.

Note, however, that the same block may have different $Addr_p$ values depending on an execution flow, for instance Block 3. Observe that such a block appears in multiple *trace paths*. Here, we define trace path P_i as a sequence of addresses $(Addr_{i,1}, Addr_{i,2}, \dots, Addr_{i,n})$, where n is the number of blocks along the execution path. Thus, two trace paths, P_i and P_j , are distinguishable if there exists a k such that $Addr_{i,k} \neq Addr_{j,k}$ (e.g., $Addr_{1,3} = Addr_4$ and $Addr_{2,3} = Addr_3$). Thus, the two Block 3's can be distinguished by the trace paths taken. Note that we extracted the trace paths from the tree without prior knowledge of input values. The tree is constructed only from a given collection of trace sequences. A higher accuracy in profiling and monitoring would be achieved by including input information when constructing the trace trees.

Now the trace tree gives us the information of how the application needs to behave in order to be considered as *legitimate execution* – i.e., in what order the traces have to be generated. In the next step, we estimate each block's execution time with samples at each node. The obtained profile will strengthen the *invariant* by enforcing what ranges of execution time each block have to fall within. The trace tree will also infer what each block's execution time should be, for individual path. However, one issue remains: a block's execution time can also vary for different inputs even along the same path (e.g., Block 3 at the right subtree in Figure 6). In what follows we address the problem of block execution time estimation in the face of varying control flow and inputs.

C. Profiling Block Execution Time Using Gaussian Kernel Density Estimation

Suppose we are given a set of samples of block execution times from a trace tree node. In this section, we show how to find a good estimation on the samples that can effectively classify the differences between legitimate and malicious execution behaviors. As previously explained, although a real-time control

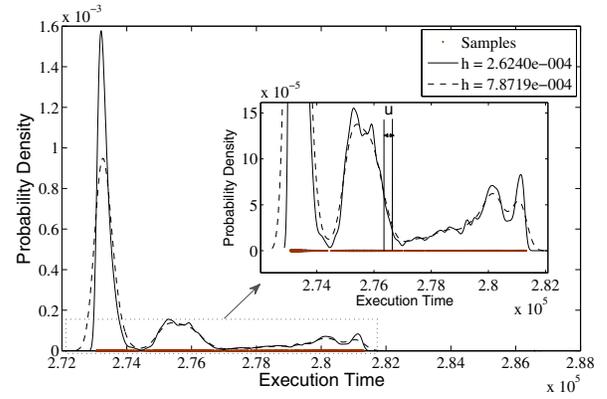


Fig. 7. Probability density estimation of an example execution block.

application has regularity in timing, noise (system effects, resource contentions, etc.), control flow variations and even input sets can cause variance in execution times. Thus, instead of trying to obtain accurate (or tight) ranges of execution times we calculate the likelihood of legitimate executions by taking into account the effects of such perturbations. For this purpose, we estimate the *probability density function* (pdf) of execution times, $f(e)$, from a set of samples, $(e^{(1)}, e^{(2)}, \dots, e^{(m)})$, by using the Kernel Density Estimation (KDE) [9], [16], [26] method. KDE is a non-parametric pdf estimation method that estimates an unknown pdf directly from sample data as follows:

$$\hat{f}_h(e|e^{(1)}, \dots, e^{(m)}) = \frac{1}{m} \sum_{i=1}^m K_h(e - e^{(i)}),$$

where K_h is a *Kernel* function and h is *Bandwidth* (also known as the *smoothing constant*). Hereafter, we simplify $\hat{f}_h(e|e^{(1)}, \dots, e^{(m)})$ as $\hat{f}_h(e)$.

There exist several kernel functions such as Epanechnikov [12], triangular, uniform, etc. However, in this paper, we use the Gaussian kernel, $K_h(x) = \frac{1}{\sqrt{2\pi}h} e^{-x^2/2h^2}$, where $-\infty < x < \infty$.⁸ The key idea of the Gaussian KDE is to first draw a scaled Gaussian distribution (parameterized by the bandwidth h) at each sample point along the x-axis (i.e., e-axis) and then to sum up the Gaussian values at each e that results in the probability density estimate at e , i.e., $\hat{f}_h(e)$. Thus the more samples that are observed near e , the higher the density estimate $\hat{f}_h(e)$ will be. Figure 7 shows the probability density estimation derived by Gaussian KDE from a set of 6708 samples of an example block (used in the prototype implementation in Section V). As can be seen from the figure the estimated pdf is in a non-regular shape compared to what could have been obtained by a parametric distribution such as Gaussian. Also, as the bandwidth becomes wider, the resulting pdf is further smoothed out. Given this pdf, one can expect that a newly observed e^* would highly likely fall within the ranges close to 2.73×10^5 , 2.75×10^5 cycles, etc.

D. Intrusion Detection Using Execution Time Profiles

To deal with the timing variations during the execution of the code, we use the idea of probability density estimations

⁸We do not address the problem of choosing the kernels and the optimal bandwidth in this paper. Interested readers can refer to [12], [16], [26].

for monitoring and detecting intrusions. We now show how this information is used to detect intrusion at run time. In what follows we limit ourselves to a single trace node (*i.e.*, a block). However, the same method is applied to all other nodes that form a part of the code. Suppose we are given the probability density estimation \hat{f}^k of node k .⁹ Let $P^k(a \leq e \leq b)$ be the probability that an arbitrary execution time e is observed between a and b with the given pdf. **Note:** the probability that e is included within a range $[a, b]$ is $P^k(a \leq e \leq b | e^{(1)}, \dots, e^{(m)}) = \int_a^b \hat{f}^k(e | e^{(1)}, \dots, e^{(m)}) de$.

Here, the obtained pdf may not be directly usable in the continuous domain, depending on the implementation. Thus, we derive the *discrete probability distributions* (or probability mass functions) instead. Let N be the number of uniformly distributed points on the e -axis that the Gaussian KDE evaluated on. Then, there are $N - 1$ bins, each of which is characterized by $[e_{min} + i \cdot u, e_{min} + (i+1) \cdot u]$. Simply put: $[b_{min}^i, b_{max}^i]$, for $i = 0, \dots, N-1$ and $u = (e_{max} - e_{min}) / (N-1)$, where e_{max} and e_{min} are the maximum and the minimum values among the observed samples, respectively. In this setting, $P^k(e^*)$, the probability of a specific execution time e^* , be approximated by

$$P^k(b_{min}^{i^*} \leq e \leq b_{max}^{i^*}) \approx \hat{f}^k(b_{min}^{i^*}) \cdot u,$$

where $i^* = \lfloor \frac{e^* - e_{min}}{u} \rfloor$, *i.e.*, $e^* \in [b_{min}^{i^*}, b_{max}^{i^*}]$; u is the bin width and $\sum_{0 \leq i \leq N-1} P^k(e \in [b_{min}^i, b_{max}^i]) = 1$.

Note that $P^k(e^*)$ is the probability of being a legitimate execution instance assuming that the estimated pdf is the true distribution. Thus, in order to deal with errors resulting from the estimation we compare $P^k(e^*)$ with a pre-defined minimum required probability, θ (*e.g.*, $\theta = 0.05$ or $\theta = 0.01$). If $P^k(e^*)$ is below θ we consider that the execution instance to be malicious.¹⁰ Hence,

$$\begin{cases} \text{if } P(e^*) < \theta & \text{malicious,} \\ \text{if } P(e^*) \geq \theta & \text{safe.} \end{cases}$$

The value of θ affects the rate of misclassification. We define a *false positive* as a case where the secure monitor says something is malicious when it is not. Similarly, a *false negative* is defined as a case where the monitor could not detect a real attack. With a higher θ , the rate of false negatives would decrease. However, at the same time, the rate of false positives will also increase. Note that setting θ to 0 implies that any execution is considered to be legitimate.¹¹

Lastly, suppose we obtained \hat{f}^k for all nodes in a trace tree. For a given sequence of traces generated during a single execution of the monitored application, the secure monitor traverses the trace tree with the address values as explained in

⁹We drop the subscript h from \hat{f}_h to simplify the expression.

¹⁰The proposed model is related to outlier detection algorithms [18]. More specifically, the null hypothesis - e^* is legitimate - is that the sample has at least θ percent of all other points having a distance to the sample less than u , the bin width. One may regard that we reject the null hypothesis when $P^k(e^*) < \theta$.

¹¹One may perform a non-parametric hypothesis test such as Wilcoxon Signed-Ranks Test [37] or Anderson-Darling Test [5]. It should be noted, however, that the intrusion detection process needs to be performed for a set of execution time samples instead of a single sample in such non-parametric tests. Thus, the process of detection could be delayed depending on how many samples are used to perform the test.

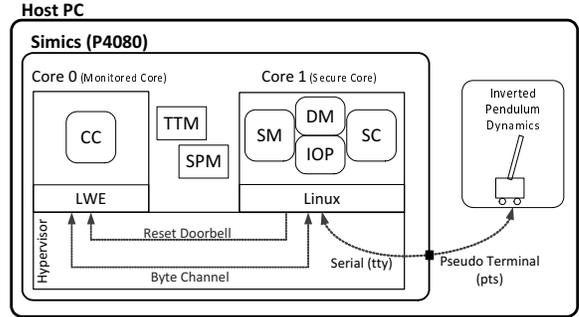


Fig. 8. SecureCore prototype implemented on Simics P4080 model.

Section IV-B. At each node k , the secure monitor calculates $P^k(t_c - t_p)$ where t_p and t_c are the timestamps when two subsequent trace instructions are executed at $Addr_p$ and $Addr_c$. If there exists at least one k such that $P^k(t_c - t_p) < \theta$, the secure monitor considers that execution to be malicious. Since we use Gaussian KDE of execution time variations for detecting intrusions, we shall henceforth refer to this technique as the *Gaussian methods for Intrusion Detection using Timing profiles* (GaIT).

V. IMPLEMENTATION

In this section, we present the implementation details for a SecureCore prototype. We first describe the hardware-level implementation and setup and then explain the software components. The latter includes a real-time control application and embedded malicious code.

A. System Implementation

We implemented SecureCore on Simics [21], a full-system simulator that can run a hardware platform including real firmware, device drivers as well as an unmodified OS and hypervisor and also allow processor architecture modifications. Figure 8 shows the system implementation overview (see Table I for the implementation parameters). We used the Freescale QorIQ P4080 Processor [3] platform that has eight e500mc cores [2]. Only two out of the eight cores were enabled - *i.e.*, cores 0 and 1 were used as the monitored and secure cores respectively. The secure core side runs Linux kernel 2.6.34. The monitored core runs on the Freescale Light Weight Executive (LWE) [4]. The choice of LWE is specific to this paper but we used it for the support of rapid reset and reload of a trusted binary it provides. The LWE could easily be replaced by any commodity or real-time OS, depending on the system requirements.

The hypervisor is configured such that the memory spaces between the cores are cleanly separated and the monitored core is set to be a *managed partition* under the secure core (core 1 can reset core 0 via a unidirectional *reset doorbell*). A byte channel (16 bytes-wide) was established to be the inter-core communication channel between the cores. We set the clock speed of each e500mc core to be 1000Mhz. In addition, we attached caches to the cores (not shown in the figure) for a more realistic environment. Each core has L1 instruction and data caches, each of size 16KB. The cores share a unified L2 cache of size 128KB. We note that without the caches, every instruction execution and data fetch would take 1 cycle on Simics.

TABLE I
IMPLEMENTATION AND EXPERIMENTAL PARAMETERS.

Component	Description
Clock speed	1000MHz
L1 Inst. and Data cache	16KB, 8 ways, latency: 2 cycles
L2 Unified cache	128KB, 32 ways, latency: 10 cycles
Exec. time of complex contr.	$[0.85^6, 1.2^6]$ cycles
Exec. times of malicious loops	440, 720, 1000 cycles (1,2,3 loops, resp.)
Min. required probability θ	0.01 or 0.05

The Timing Trace Module (TTM) was implemented by extending the Simics `sample-user-decoder` that is attached to core 0. When the decoder encounters the trace instructions, the relevant information is written to the SPM. We modified the ISA of the e500mc core [2] so that the execution of the `rlwimi` instruction will trigger an event to the TTM.¹² As shown at the bottom of Figure 4, there are four types of trace events differentiated by the last parameter: (a) `INST_REG_PID` registers the process ID of the calling application with the TTM, (b) `INST_ENABLE/DISABLE_TRACE` enables/disables the trace operations of TTM and (c) `INST_TRACE` writes a trace to the SPM. As mentioned in Section III-C, when tracing is turned on and the `INST_TRACE` instruction is executed, the timestamp and the instruction address at the point of the execution are written at the address specified by the value of `AddrHead`. The SPM has a size of $4KB$ and is mapped to a region of core 1’s address space by the hypervisor.

Lastly, all processes including secure monitor (SM), decision module (DM), I/O proxy (IOP) and the safety controller (SC) run in user-space. Sending/receiving data through the byte channel is done via a kernel module that requests a hypervisor call. The processes (Figure 2) execute with a period of 10 ms.

B. Application Model

As our physical control system, we used an Inverted Pendulum (IP). However, since the simulation speed of Simics is slower by an order of magnitude than the dynamics of a real IP, we used control code and related dynamics generated from a Simulink [32] IP model. These were then encapsulated into software processes. The dynamics process runs on the host PC and is synchronized with the control system managing it in Simics through a pseudo terminal (**Note:** Simics sees it as a real serial connection). The physical state of IP is defined as the cart position and the rod’s angle, perpendicular to the ground. This state is sent to the controllers executing on Simics and they, in turn, compute an actuation command that is then sent back to the dynamics process that then emulates the action of the IP. For realistic dynamics, we embedded a *Gaussian noise generator* at the output of the rod angle in the dynamics.

As mentioned above, the system runs on Simics and monitors the IP control application. We use the same control code for the complex and simple controllers for evaluation purpose only. However, since the code is too simple with very little variance in execution time, we inserted a fast fourier transform (FFT) benchmark from the EEMBC AutoBench suite [11], `aiff_t`, to the complex controller as shown in Figure 9 (a). The benchmark consists of three phases after initialization. We modified it so

¹²`rlwimi` is the *Rotate Left Word Immediate Then Mask Insert* instruction. Execution of `rlwimi 0,0,0,0,i` for $0 \leq i \leq 31$ is equivalent to `nop`.

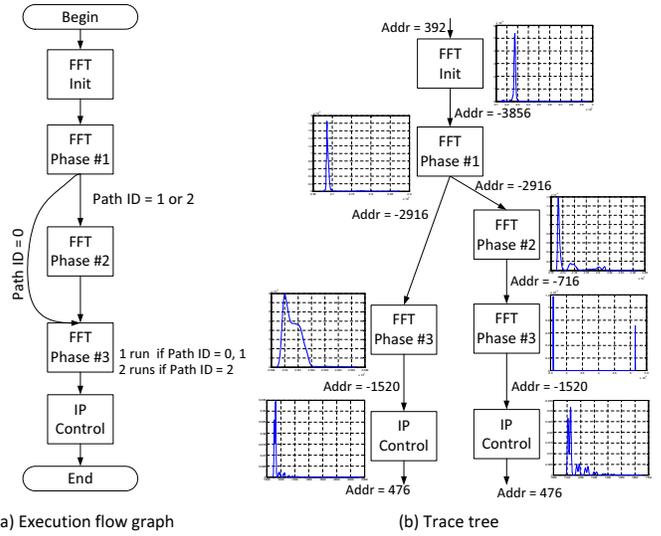


Fig. 9. The execution flow and the corresponding trace tree of IP+FFT.

that after initialization it randomly selects a path ID. If the ID is ‘0’, FFT Phase 2 is skipped, and Phase 3 is executed twice if the ID is ‘2’. From this structure, we wish to observe how well our detection methods can deal with execution time variances caused by inputs and flows. After the FFT phases complete, the IP control logic is executed. The logic controls the IP so that it is kept stabilized at position ‘+1’ meter from the origin.

We inserted *malicious code* at the end of FFT Phase 3. It is a small loop in which some arrays used in previous FFT phases are copied. The average execution time of the malicious code is 440, 720 and 1000 cycles for 1, 3 and 5 loops respectively. The code becomes activated when the cart position of IP received from IOP becomes $+0.7$ meter. Thereafter the code is executed randomly and the complex controller discards the actuation command calculated by the IP logic and sends out one duplicated from the previous execution. This will result in two effects – variances in execution time that differ from expected values and also sends wrong actuation information to the control system. Both of these effects should trigger our detection systems.

To profile the execution times of the complex controller, we inserted `INST_TRACE` instructions at the end of each block and one at the top of each flow (*i.e.*, before `FFT_init()`) as explained in Section IV-B. We executed the system in a normal condition (*i.e.*, no malicious code activation) for 10,000 runs until we obtained a collection of traces. From these traces, a trace tree is constructed as shown in Figure 9 (b). We then used the `ksdensity` function in Matlab to derive the pdf estimation \hat{f}^k of the samples at each block k .¹³

VI. RESULT AND DISCUSSION

In this section, we evaluate our SecureCore architecture through experiments on the prototype presented in Section V. We then discuss some limitations and possible improvements.

¹³We set the number of bins, N , to 1000. The kernel smoothing bandwidth h is then automatically selected by the function.

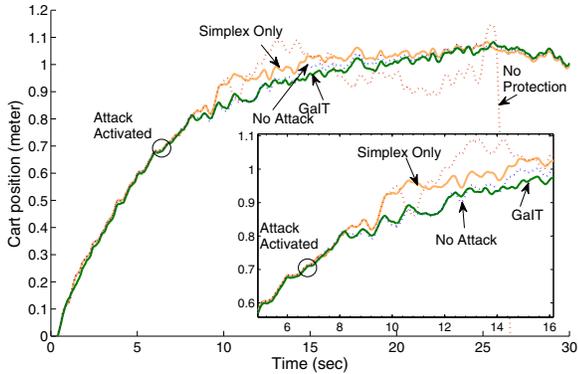


Fig. 10. Trajectory of cart with different protection approaches.

A. Early Detection of an Intrusion

We first evaluate our timing-based intrusion detection method by measuring how quickly it can detect malicious code execution compared to vanilla Simplex-only approach. As explained in Section V-B, the malicious code embedded in the complex controller is activated when the cart passes through the point at $+0.7$ m. In this evaluation, we set the minimum required probability θ to 0.01 and the loop count of the malicious code to 3. The cart positions were traced from the IP dynamics process for the cases where (a) there is no attack, (b) attack + no protection, (c) attack + Simplex only and finally (d) attack + Simplex + GaIT (our detection method). Additionally, we set an event that is triggered when Simplex or our method detect anomalies. However, for evaluation purposes, we intentionally disabled our method until the cart passes over $+0.5$ meter; if we enable it from the beginning, a false positive would activate the safety controller before an attack takes place.

Figure 10 shows the different trajectories of the cart for the four cases. The cart is stabilized at the position near $+1$ m when there is no attack or if the control logic is protected (either by GaIT or vanilla Simplex). When there is no such protection mechanism, however, the cart becomes destabilized finally after time 25 seconds. When the protection mechanisms are active (SecureCore + GaIT) and the malicious code was activated (at around 6.9 seconds), it was almost instantly detected by GaIT. We can see this from the magnified section of the plot showing the trajectory of the cart along with the normal case (*i.e.*, no attack). On the other hand, although it is not clear in the figure, the Simplex-only method detected the abnormal behavior of the complex controller at around 9.5 seconds. In this case, we see that the cart has deviated from its normal trajectory for a moment; it was later returned to the normal trajectory. Even though the experiment was performed with a restrictive setup for a simple application, the result shows that our timing-profile based intrusion detection method (GaIT) can supplement Simplex through early detection. Even though vanilla Simplex can detect malicious activity, it does so much later than GaIT and only because the compromised controller tried to actuate the physical system into an unsafe state. Many times, attackers may not send wrong actuation commands – they may snoop on the operation of the system and collect privileged information. SecureCore and GaIT will be able to detect such activity almost

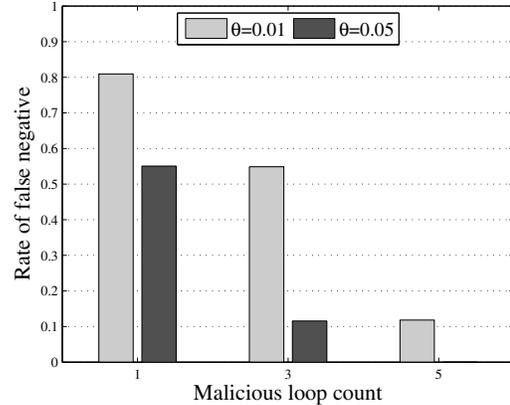


Fig. 11. False negative rates for different θ and malicious loop counts.

TABLE II
FALSE NEGATIVE RATES (# ATTACKS MISSED / # ATTACKS TRIED).

	1 loop	3 loops	5 loops
$\theta = 0.01$	827/1022(81%)	574/1046(55%)	130/1098(12%)
$\theta = 0.05$	578/1050(55%)	117/1011(12%)	0/1045(0%)

instantaneously, as evidenced here, while Simplex will fail to detect it. Also, attackers could increase the wear and tear on the physical system under vanilla Simplex – by causing the system to operate, albeit briefly, in an unsafe state. This can also be avoided by use of our techniques.

B. Intrusion Detection Accuracy

The early detection capability, however, can be effective only when a higher classification accuracy is possible. Thus, we evaluate the accuracy of our intrusion detection method by measuring the false positive and false negative rates. In this experiment, we disabled the reset mechanism of secure core to correctly count the number of attacks and misclassifications. However, the functionality exists for future work. As mentioned before, a false positive occurs when the monitor classifies an execution to be malicious when it was not and a false negative is when a malicious attack goes undetected. The evaluation was performed with the minimum required probability θ set to 0.01 or 0.05. For each case, the loop count of the malicious code was set to 0, 1, 3 and 5. Then, we sampled decisions made by the secure monitor until we collected at least 1000 samples.

To measure the rate of false positives, we ran the system without activating the malicious code. For $\theta = 0.01$, only *one* false positive out of 1024 samples was found. With $\theta = 0.05$, the monitor classified 7 samples out of 1015 legitimate executions as attacks. We then activated the malicious code to measure the false negative rates. Table II shows how many attacks the monitor missed for each θ and loop count. For example, for $\theta = 0.05$ and the loop count of 3, the monitor could not detect 117 out of a total 1011 malicious code executions. As can be seen from Figure 11, the false negative rate decreased when the malicious code executed for longer time frames. For the same execution, a higher value of θ also showed reductions in the rate of false negatives. However, as previously mentioned, there is a tradeoff between a higher θ and a lower one. That is, while setting θ higher can reduce the chances that the monitor

will not miss malicious code execution, it can also increase the rate of false alarms. In such cases, the control would be frequently switched to the safety controller even if the complex controller is not compromised. This could degrade the overall control performance for the physical system. Thus, a balanced θ should be obtained, either through extensive analysis or through empirical methods.

C. Limitations and Possible Improvements

The main cause of misclassifications comes from noise during execution time profiling. A legitimate execution time might not appear in the samples but might be observed during the actual monitoring phase. Moreover, the malicious execution time might also fall within a legitimate interval. This is especially possible when an attacker exploits the system by using a short and steady malicious code execution such as an example using the Return-Oriented Programming (ROP) attack [29]. An ROP-based attack is a sequence of short code blocks each of which would last for less than 100 cycles (or even 10 cycles) before returning to the original execution path. Thus, an attacker may deceive the proposed detection method by executing such short code because legitimate timing variations would likely last longer than the ones caused by such malicious code. As the result in the previous subsection shows, the proposed method would not perform well when such attacks are employed.

Thus, it is the key that we narrow the range of execution time variances as much as possible so that the above situations, *i.e.*, the probability that a legitimate execution instance can fall within the range and that even a short length of malicious execution can deviate from the range is maximized. One way to achieve this is to run the final system on a real-time operating system that inherently has more deterministic execution times. Disabling interrupts during execution (if possible) [22] or locking frequently used data or instructions on cache [6] can help increase the predictability of such executions. In addition, using a real-time multicore processor [20], [24], [39] can further improve the accuracy by reducing or eliminating unpredictable variations in execution times caused from contentions on shared resources such as cache, bus, memory, *etc.* We are also investigating methods to improve our analysis techniques to detect such small, constant, variations.

VII. RELATED WORK

The work that is closest to that presented here is the Secure System Simplex Architecture (S3A) proposed by Mohan *et al.* [22]. That architecture employs an FPGA-based trusted hardware component, that monitors the execution behavior of a real-time control application running on a untrustworthy main system. They use the execution time and the period of the application as a side-channel monitored by the trusted hardware. The main difference of our work from S3A is that, in our work, finer executions units are profiled and monitored by a statistical learning method taking application contexts into account as well. Also, we target the use of multicore architectures for secure embedded real-time systems. An earlier work was proposed by Zimmer *et al.* [41], in which the absolute worst-case execution time (WCET) was used as a security invariant.

There exists some work in which a multicore processor (or a coprocessor) is employed as security measure in different aspects. One example is the Dynamic Information Flow Tracking (DIFT) mechanism by Suh *et al.* [35]. Shi *et al.* [31] proposed INDRA, an Integrated Framework for Dependable and Revivable Architecture, in which logs of application executions on monitored cores are verified by a monitoring core through buffering of logs on a special on-chip memory. While this is similar to the work proposed here, the difficulties arise due to the real-time nature of the systems we consider and problems with contention in shared resources that could result in security violations. Also, the security measure in their work is functional behavior such as function calls and return (*e.g.*, the monitoring core verifies if each function always returns to the right address). We focus on the execution profiles of the tasks (*e.g.*, timing) as security invariants – something that is very feasible in real-time systems but not in general purpose ones. Similar work can be found in by Chen *et al.* [7]. The work also employs a logging hardware that captures the information, *e.g.*, program counter, input and output operands and memory access addresses of any instruction that the monitored application executes. The captured traces are delivered through a cache to another core for inspection. The work was extended where a hardware accelerator was proposed to reduce high overheads in instruction-grain monitoring [8]. There have also been coprocessor-based approaches. Kannan *et al.* [17] addressed the high overheads in the multicore-based DIFT by proposing the DIFT Co-processor, in which application instructions and memory access addresses, *etc.*, are checked with a pre-defined security policy. A similar approach was taken by Deng *et al.* [10] where reconfigurable logic attached to the main CPU checks for software error as well as DIFT from execution traces.

All of these techniques, while applicable for general purpose systems, could also be repurposed for embedded real-time systems. Hence, the combination of these techniques with our SecureCore and GaIT approaches will only make the overall system more secure and hence, safer.

Hofmeyr *et al.* [13] developed an intrusion detection method based on a profile of normal behavior by using traces of system calls (however, no machine learning technique was employed). The profile is a database of unique sequence of legitimate system calls of a specific length. During monitoring each sequence of system calls is tracked and the method checks if a significant deviation from the legitimate trace has occurred. Muckamala *et al.* [23] developed an intrusion detection method based on users' behavior pattern. Using a neural network and a support vector machine (SVM) mechanism, a classifier is trained to learn a set of important behavior features and to recognize abnormal behavior patterns. In fact, most of the work for intrusion detection systems using machine learning have focused on network activity monitoring. Sinclair *et al.* [33] used decision trees along with genetic algorithms to generate rules for differentiating malicious traffic from normal network activities. Similar work used robust support vector machines (RSVMs) to detect network activity anomalies [14]. A good summary of machine learning for network security is provided by Sommer *et al.* [34].

VIII. CONCLUSION

In this paper, we proposed SecureCore, a novel application of a multicore processor for creating a secure and reliable real-time control system. We used a statistical learning method for profiling and monitoring the execution behavior of a control application (GaT). Through the architectural and the theoretical support, our intrusion detection mechanism implemented could detect violations earlier than just a pure safety-driven method, Simplex. This helps in achieving reliable control for physical systems. The isolation achieved by SecureCore and the monitoring mechanisms presented by GaT also prevents attackers from causing harm to the physical systems, even if they gain total control of the main controller. Evaluation results showed that with careful analysis and design of certain parameters, one can achieve a low misclassification rate and higher intrusion detection rates. As future work, we plan to extend the presented approach to support concurrent monitoring of multiple applications on multiple cores. Also, we intend to address many of the limitations presented in Section VI-C.

REFERENCES

- [1] Autosar 4.0. <http://www.autosar.org/index.php?p=3&up=1&uup=0>.
- [2] e500mc Core Reference Manual. http://cache.freescale.com/files/32bit/doc/ref_manual/E500MCRM.pdf.
- [3] Freescale QorIQ P4080 Processor. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4080.
- [4] Freescale's Embedded Hypervisor for QorIQ P4 Series Communications Platform. http://cache.freescale.com/files/32bit/doc/white_paper/EMBEDDED_HYPERVISOR.pdf?srch=1&sr=2.
- [5] T. W. Anderson and D. A. Darling. Asymptotic theory of certain "goodness of fit" criteria based on stochastic processes. *The Annals of Mathematical Statistics*, 23(2):193–212, 1952.
- [6] A. Arnaud and I. Puaut. Dynamic instruction cache locking in hard real-time systems. In *Proc. of the International Conference on Real-Time and Network Systems*, May 2006.
- [7] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Proc. of the workshop on Architectural and system support for improving software dependability*, pages 63–65, 2006.
- [8] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proc. of the International Symposium on Computer Architecture*, pages 377–388, 2008.
- [9] J. Choi and E. Amir. Lifted relational variational inference. In *Proc. of the conference on Uncertainty in Artificial Intelligence*, pages 196–206, 2012.
- [10] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *Proc. of the IEEE/ACM International Symposium on Microarchitecture*, pages 137–148, 2010.
- [11] EEMBC AutoBench Suite. <http://www.eembc.org>.
- [12] V. A. Epanechnikov. Non-parametric estimation of a multivariate probability density. *Theory Probab. Appl.*, 14(1):153–158, 1969.
- [13] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, 1998.
- [14] W. Hu, Y. Liao, and V. R. Vemuri. Robust anomaly detection using support vector machines. In *Proc. of the International Conference on Machine Learning*, 2003.
- [15] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for vliw and superscalar compilation. *J. Supercomput.*, 7:229–248, 1993.
- [16] M. C. Jones, J. S. Marron, and S. J. Sheather. A brief survey of bandwidth selection for density estimation. *Journal of the American Statistical Association*, 91(433):401–407, 1996.
- [17] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *Proc. of the IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 105–114, 2009.
- [18] E. M. Knorr and R. T. Ng. A unified notion of outliers: Properties and computation. In *Proc. of the International Conference on Knowledge Discovery and Data Mining*, pages 219–222, 1997.
- [19] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 447–462, 2010.
- [20] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *Proc. of International Conference on Compilers, Architecture, and Synthesis from Embedded Systems*, 2008.
- [21] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hillberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.
- [22] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo. S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *Proc. of the ACM International Conference on High Confidence Networked Systems*, 2013, to be published.
- [23] S. Mukkamala, G. Janoski, and A. Sung. Intrusion detection using neural networks and support vector machines. In *Proc. of the International Joint Conference on Neural Networks*, pages 1702–1707, 2002.
- [24] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *Proc. of IEEE/ACM International Symposium on Computer Architecture*, pages 57–68, 2009.
- [25] P. Parkinson. Safety, security and multicore. In *Proc. of 19th Safety-Critical Systems Symposium*, pages 215–232, 2011.
- [26] E. Parzen. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076, 1962.
- [27] M. Schoeberl and P. Puschner. Is chip-multiprocessing the end of real-time scheduling? In *Proc. of International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.
- [28] L. Sha. Using simplicity to control complexity. *IEEE Softw.*, 18(4):20–28, 2001.
- [29] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proc. of the ACM Conference on Computer and Communications Security*, pages 552–561, 2007.
- [30] D. Shepard, J. Bhatti, and T. Humphreys. Drone hack: Spoofing attack demonstration on a civilian unmanned aerial vehicle. *GPS World*, Aug 2012.
- [31] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. In *Proc. of the International Symposium on Computer Architecture*, pages 102–113, 2006.
- [32] Simulink. <http://www.mathworks.com/products/simulink>.
- [33] C. Sinclair, L. Pierce, and S. Matzner. An application of machine learning to network intrusion detection. In *Proc. of the Computer Security Applications Conference*, pages 371–377, 1999.
- [34] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 305–316, 2010.
- [35] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–96, 2004.
- [36] US-CERT. ICSA-10-272-01: Primary stuxnet indicators. Aug. 2010.
- [37] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.
- [38] P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves. Implementing embedded security on dual-virtual-cpu systems. *IEEE Des. Test*, 24(6):582–591, Nov. 2007.
- [39] M.-K. Yoon, J.-E. Kim, and L. Sha. Optimizing tunable wcet with shared resource allocation and arbitration in hard real-time multicore systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 227–238, 2011.
- [40] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 129–142, 2010.
- [41] C. Zimmer, B. Bhatt, F. Mueller, and S. Mohan. Time-based intrusion detection in cyber-physical systems. In *Proc. of the ACM/IEEE International Conference on Cyber-Physical Systems*, pages 109–118, 2010.