ReSecure: A Restart-Based Security Protocol for Tightly Actuated Hard Real-Time Systems

Conference Paper · November 2016

CITATION
1

READS
29

5 authors, including:

Fardin Abdi Taghi Abad
University of Illinois, Urbana-Champaign
9 PUBLICATIONS **24** CITATIONS
SEE PROFILE

Monowar Hasan
University of Illinois, Urbana-Champaign
19 PUBLICATIONS **427** CITATIONS
SEE PROFILE

Marco Caccamo
University of Illinois, Urbana-Champaign
131 PUBLICATIONS **3,461** CITATIONS
SEE PROFILE

Proceedings of

2016

the 1st Workshop on Security and Dependability of Critical Embedded Real-Time Systems

November 28th, 2016 in Porto, Portugal

in conjunction with

IEEE Real-Time Systems Symposium

29th November 2016 – 2nd December 2016

Editors:
Marcus Völp
Paulo Esteves-Veríssimo
António Casimiro
Rodolfo Pellizzoni

ReSecure: A Restart-Based Security Protocol for Tightly Actuated Hard Real-Time Systems

Fardin Abdi[†*], Monowar Hasan[†*], Sibin Mohan[†], Disha Agarwal[‡] and Marco Caccamo[†]
[†]Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, USA
[‡]Dept. of Computer Science and Engineering, PES University, India
Email: {[†]abditag2, [†]mhasan11, [†]sibin, [†]mcaccamo}@illinois.edu, [‡]agarwal.disha21@gmail.com
*These authors have made equal contribution to this work.

***Abstract*—In this paper we present ReSecure, a framework that uses the concept of *system restart* to *secure* hard real-time systems (RTS). ReSecure is used to *improve the security of RTS without violating safety or temporal constraints*. We also show how designers of systems can customize (or even optimize) system parameters to achieve the best trade-offs between security and control system performance. We demonstrate our concepts using a prototype on an ARM-based embedded platform as well as a 3 degree of freedom (3 DOF) helicopter system.**

I. Introduction

Physical systems have limitations and restrictions that need to be respected during operation. Otherwise, the system itself or the environment around them, including humans, can be damaged. These physical systems are usually controlled by embedded real-time controllers and require consistent availability and reliable execution in order to operate safely. These limitations, therefore, need to be taken into account when designing the controllers for these systems. With the growing complexity of these controllers due to the use of sophisticated operating systems (OS), vendor developed drivers and open source libraries as well as increased connectivity makes it very challenging to verify that those physical restrictions are always respected.

Despite the fact that sensors and actuators that interact with the physical world may open up composite sources of vulnerabilities, security issues in real-time systems (RTS), however, have received relatively little attention by academia or industry. The sophistication of recent attacks, limited computing resources due to the embedded architecture as well as exceptionally high reliability and availability requirements make it harder to build effective intrusion detection and prevention mechanisms in RTS.

Recent RTS are more interconnected and even controlled over unreliable mediums such as the Internet. In addition, there is more monetary and adversarial incentives for malicious activities. The Stuxnet worm that highlighted the possibility and effectiveness of an attack on critical infrastructure [1] and malicious code injection into the telematics units of modern automobiles [2], [3] are some instances of such attacks. Without considering the safety-critical and embedded nature of these systems, simply adding security mechanisms (such as encryption, message authentication, *etc.*) will not be effective to ensure both safety and security.

Hence we intend to provide designers a flexible yet unified framework that allows to customize system parameters (security policies and preference over security and performance). In this paper we present ReSecure, a runtime restart-based protection approach to ensure security in RTS. In traditional computing systems (*e.g.,* servers, smart phones, *etc.*), software problems are often resolved by restarting, either the application process or the platform [4]–[6]. We use a similar concept to improve security guarantees in RTS. In particular, we propose to *restart the platform periodically/asynchronously and load a fresh image of the applications and OS after each reboot* with the objective of wiping out the intruder or malicious entity.

Unlike conventional computing systems restart-based recovery mechanisms are not straightforward in RTS due to additional real-time constraints, as well as interactions of the control system with the physical world. Besides, most physical systems need *consistent actuation with tight timing constraints* (*e.g.,* a helicopter can quickly destabilize if the controller restarts). The ReSecure framework is specifically designed to improve security of safety-critical RTS with physical systems in need of constant actuation. In order to guarantee the physical safety requirements, we use the *Simplex architecture* [7], [8], a method that utilizes a minimal, verified controller as backup when the complex, high-performance controller is not available or malfunctioning. In particular, ReSecure is a set of software and hardware mechanisms that enable a trade-off between the security guarantees and control performance while guaranteeing the *safety*[1] of the physical system at all times. Specifically, the contributions of this work can be summarized as follows.

- We introduce ReSecure, a restart-based architectural framework to improve security in RTS. We discuss how triggering restart events through several system components can enhance security without sacrificing safety (Section IV).
- To configure system parameters based on the design requirements, we provide an analytical framework that calculates the best trade-off between security and controller performance (Section IV-C).
- We evaluate the system with a proof-of-concept implementation on an ARM-based development board and embedded real-time Linux (Section VI).

The ReSecure framework proposed in this paper is based on Simplex [7], [8] that enables the safe restart mechanisms. Specifically, we utilize a variant of Simplex, *viz., system-level Simplex* [9]. We present first a brief overview of Simplex and our assumptions on adversarial capabilities before we elaborate the design details of ReSecure.

II. Overview of Simplex Architecture

Simplex [8], [10]–[12] is a well-known architecture that uses a simple verified subsystem (Fig. 1) to ensure safety of the plant. This conservative safety subsystem is then complemented by a high-performance complex control subsystem that is concerned with mission-critical requirements. A decision module then uses the high-performance complex controller whenever possible, but will switch to the safety controller when system safety is jeopardized. Using Lyapunov stability properties from control theory, there exists a set of states that

[1]By the term 'safety' we refer to ensure that the limitations and restrictions of the physical system (*e.g.,* actuator limits, maximum pressure, maximum temperature, *etc.*) are always respected.

within those set, safety control is always able to stabilize the system and keep it safe. The goal of Simplex method it to guarantee that under any behavior of the complex subsystem, the physical system would remain safe.

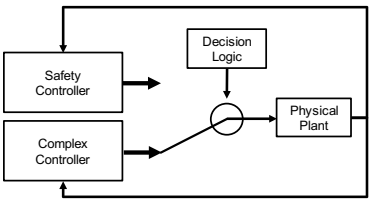


Fig. 1. Logical view of Simplex architecture. Decision module chooses the control that does not jeopardize the safety.

Our proposed approach is based on variants of Simplex such as *system-level Simplex* [9] and *reset-based recovery* [13] that have moved the safety subsystem and the decision module into a dedicated hardware unit. This isolates the trusted components from the faults and misbehavior of the complex subsystem.

III. Adversary Model

RTS face threats in various forms depending on the system and the goals of an adversary. For instance, adversaries may insert, eavesdrop on or modify messages exchanged by system components. Besides, attackers may manipulate the processing of sensor inputs and actuator commands, could try to modify the control flow of the system as well as extract sensitive information through side-channels. The adversarial capabilities we consider in this work are as follows.

i) *Integrity violation*: We assume that the adversary can compromise all the software components on the complex unit including underlying *real-time OS (RTOS)* as well as the *real-time applications*. For example, an adversary may insert a malicious task that respects the real-time guarantees of the system to avoid immediate detection, compromise one or more existing real-time tasks and/or may override parts of the underlying RTOS itself that may eventually threaten overall safety of the system.

ii) *Denial of Service (DoS)*: The attacker may take control of the real-time task(s) in the complex controller and perform *system-level* resource (*e.g.,* CPU, disk, memory, *etc.*) exhaustion. In addition, an advanced attacker may capture I/O or network ports and perform *network-level* attacks to tamper with the confidentiality and integrity (*viz.,* safety) of the system.

iii) *Information leakage through side-channels*: The adversary may also aim to learn important information by side or covert-channel attacks by simply lodging themselves in the system and extracting sensitive information. For example, the intruder may utilize side-channels to monitor the system behavior and infer certain degree of system information (*e.g.,* hardware/software architecture, user tasks and thermal profiles, *etc.*) that eventually leads to the attacker actively taking control, manipulating and/or crashing the system.

In addition to these adversarial capabilities, we also make the following assumptions.

i) *Sensor reading manipulation*: We consider that an intelligent adversary can only attack the external sensors via physical access. Since this may not be the obvious attempt in practice, we assume that the sensor readings are *not compromised*.

ii) *Safety unit integrity*: Safety unit has simple and verified software that has no interaction with the outside world except reading the complex controller command and sensor values in every cycle, and sending the final command to the actuators. Therefore, we assume that safety unit remains *uncompromised* during system operation[2].

iii) *Read-only memory unit*: We assume that the attacker *cannot* modify the content of the read-only memory unit that stores the uncompromised image of the RTOS and real-time applications. For example, this can be achieved by using an E^{2}PROM storage unit.

IV. Restartable Design: Approach and Overview

We consider a real-time control application, *viz.,* a system consisting of a set of sporadic, fixed priority[3], independent tasks executing on the complex unit. Each real-time task τ_i is characterized by (C_i, T_i, D_i), where C_i is the worst-case execution time, T_i is the minimum inter-arrival time (or period) between successive releases and D_i is the relative deadline. Although we limit ourselves to fixed-priority scheduling in this paper, the concepts of restarting to secure RTS can be extended to dynamic priority schemes without loss of generality. The schedulability of the real-time application tasks are assumed to be guaranteed by offline analysis [15]. Our design goal is to enable the complex subsystem to restart and reload an uncompromised image containing all the applications and the underlying RTOS/firmware while guaranteeing that the physical plant stays safe during restart.

The high level overview of ReSecure is illustrated in Fig. 2. As mentioned earlier, we use the system-level Simplex [9] as the basis of our design. In this architecture, safety and complex subsystems are on isolated hardware units such that faults or exploits in the complex subsystem does not affect the functionality of the safety unit. The only interaction of the safety subsystem has with the outside world is limited to reading the complex controller commands and sensor values in a controlled format and sending the control commands to actuators. Therefore, we do assume that the safety unit *always* remains trusted during system operation and there is no malicious entity embedded in it[4]. The entire security framework can be collapsed if the safety unit itself is compromised in the first place.

The decision module switches the control to the safety controller when the commands from the complex controller may jeopardize safety or when there is no command from the complex controller (*e.g.,* during a restart). The Safety controller maintains (a somewhat degraded) performance until a safe command is available from complex controller (*e.g.,* the restart completes and the original controller can resume its operation)[5].

One may argue why complex unit does not receive the same level of protection as the safety unit. The complex unit, however, is often exposed to external sources (*e.g.,* user input, I/O or network peripherals, software update, *etc.*) that opens it up to the potential sources of vulnerabilities. As mentioned earlier, the binary or control logic in the safety unit will not often change. Therefore, it makes sense to harden security mechanisms in the complex unit.

[2]This assumption is clarified further in Section IV.

[3]One approach to assign priority could be using the Rate Monotonic algorithm [14].

[4]Similar assumptions are not uncommon in literature [16], [17].

[5]For the detailed description of safety controller and decision module design procedure we refer the readers to [9] and [13].

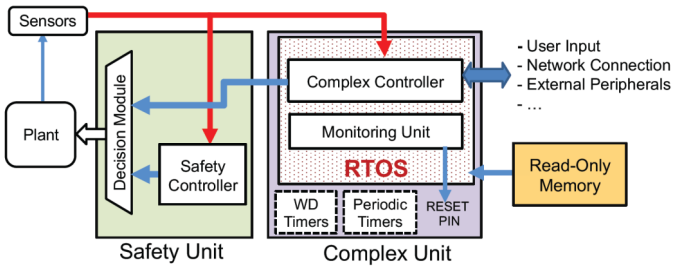


Fig. 2. The ReSecure architecture produces a verified system despite the use of an unverified complex controller. The decision module switches between the controllers to provide overall system safety.

A. Triggering Restarts

In ReSecure platform, the attacker can compromise the complex unit but cannot violate the physical safety because of the decision module and safety unit. However, the system may not made progress towards the mission with only the safety controller. In order to recover the complex controller from a compromised state, we choose the following set of events to trigger a full restart of the complex unit and reload an uncompromised image of *both OS and control applications* from read-only memory: *i*) failure of critical components, *ii*) at predefined periodic times and *iii*) upon detection of an intrusion by the monitoring unit. In the following we elaborate the above choices and discuss the mechanisms to implement each one.

1) Watchdog Timer: Watchdog timers ensure that the components in the complex unit are alive and have not crashed. Hence, the complex controller and monitoring unit should update a watchdog timer at every execution loop. Thus, the platform needs to provide *at least one* watchdog timer. The watchdog must be independent of everything else in the system (except power). It is important to mention that most watchdog timers are "no way out", meaning that once activated, they cannot be deactivated. However, this must be verified on the specific platform which is being used. If the watchdog can be disabled after activation, an attacker might be able to disable the watchdog and prevent a restart.

2) Monitoring Unit: Monitoring unit can act as an attack detector and look for signs of malicious behavior in the system. Alternatively, it can look for signs of fault in the system before they occur. For instance, it can perform tests that can cover resource availability (*e.g.,* sufficient memory and file handles, reasonable CPU time, *etc.*), evidence of expected process activity (*e.g.,* system processes running, specific files being presented or updated, *etc.*), overheating, network activity, system-specific tests as well as monitoring for specific attack or security threats. For each one of the above, there is a large body of work in the literature that can be plugged in. Thus, we do not consider the specific design/functionality of the monitoring unit. Our flexible framework enables designers to implement their own monitoring methods to improve security depending on system requirements.

3) Periodic Timer: No matter how effective the monitoring strategy is, it can never detect all compromises in the system. A sophisticated attacker may even be able to disable the monitoring unit. Besides, not all the intrusions necessarily crash the complex controller and/or initiate a restart through watchdog timers. For example, an attacker may just modify the complex controller logic. The external timers trigger periodic restarts and reload uncompromised system image independent of all the components and events in the system. This allows the system to recover from the *unforeseen attacks* (*e.g.,* zero-day vulnerabilities) that all the other restart-triggering mechanisms (*e.g.,* watchdog timers and monitoring unit) fail to detect.

Note that ReSecure does *not* make any assumptions with regards to the detectability of attacks. An attack can remain undetected by monitoring unit or even may compromise the monitoring unit. However, even such sophisticated attacks will be removed when the periodic timers trigger the restart. Needless to say that a more effective monitoring mechanism leads to a faster detection and recovery and consequently a lesser loss of control performance.

B. Restartable Components

Since aforementioned triggers can occur asynchronously at any time during operation, all the components on the complex unit must be designed to be restarted at any arbitrary time. In conventional software systems that are not designed for restartability, restarting the component or the entire system at states where the restart is not expected might result in unrecoverable data-dependent failures. This can occur in two main ways: *i*) due to logic data corruption, and *ii*) due to file system corruption. The case of logic data corruption may occur if the system is being restarted while an application was updating a unit of storage (*e.g.,* a file). In this case, the semantics of the partially updated storage may be affected, leading to the permanent inability to restart an application correctly. Conversely, file system corruption can occur as the system is forcefully restarted after the execution of disk-cached I/O operations and before cached file changes are synchronized with the disk.

For this paper, we make the following considerations about the restartability of software components. First, many microcontroller-based systems as well as many embedded OS provide no support for persistent storage and thus provide no file systems. In these class of systems, a fresh image of the OS and applications is loaded into main memory from a read-only memory resource (typically a flash memory) during a restart. Second, for those platforms with support for file systems, restartability can be achieved if a read-only partition is designated for the storage of the OS and control application binaries. This will ensure that the OS and application images will not be impacted by file system corruption during restart.

C. Securing RTS by Restarting

Restarting the entire complex unit and loading the fresh image of all the framework and applications restores the system into a functional and uncompromised state. However, it does not fix the vulnerabilities that were exploited in the first place by the attacker. That is why that a single restart cannot be considered as an effective attack prevention mechanisms by itself.

The main idea of restart-based protection is that, if we restart the system *frequently enough*, it is less likely that the attacker will have time to become effective and cause meaningful damage to the system. After every restart, there will be a predictable down time (during the system reboot), some operational time (before system is compromised again) and some compromised time (until the compromise is detected or periodic timer expires). The periodic timer sets an upperbound on the compromised time of the system.

The length of each one of the above intervals depends on the type and configuration of the platform, adversary models, effectiveness of the monitoring unit and complexity of the exploits. As a general rule, the effectiveness of the restarting mechanism increases *i*) as the time to re-launch the attacks increases or *ii*) the time to detect attacks and trigger a restart decreases. In the Appendix, we provide a probabilistic

method[6] to evaluate the expected unavailability due to restarts and attacks and the expected damage from the attacks/exploits given a certain restart configuration. We also show how to find an optimal configuration to balance lack of availability and damage.

V. ReSecure and Attack Resiliency

Most of the existing security mechanisms protect the system from specific categories of attack and providing high security assurance requires a mixture of multiple mechanisms. Even then, there are always unknown vulnerabilities that can be exploited by an attacker to perform *zero-day* attacks. On the other hand, patching the OS in embedded controllers and programmable logic controllers (PLCs) against (un)known bugs and vulnerabilities is very challenging. Due to the safety-critical nature of the applications in industrial controllers, any updates to the software needs to undergo extensive testing and verification before being deployed. ReSecure on the other hand, guarantees complete safety of physical unit along with some graceful degradation and restoration of functionality in the presence of any type of exploits.

In the following, we review some examples of security threats that ReSecure can recover.

1) Malicious Software: Sophisticated adversaries can create malware that seeks to infect the external media used for transfers and updates. This opens the opportunities for possible exploits and may infect the system unknowingly to the human operator [19]. Some malicious code injections requires *time*, *resources* and sometime *physical intervention* to spread through systems. For example, the Stuxnet worm [1] was able to successfully subvert industrial control systems and infect PLCs. The worm was able to get in once an infected peripheral device was connected to the main system. After gaining access, the malware gradually inflicted damage to the physical plant by substituting legitimate actuation commands with infected ones *over a period of time*. Flame [20] is another example of malware designed specifically for espionage that spreads through USB devices and local networks.

Such malwares get into the system from removable drives that requires physical access and later slowly spread over the network by replicating themselves. In practice, however, it is unlikely that malicious entities can gain frequent physical access to the system after every reboot.

2) Stealing Sensitive Information: An adversary may want to track side-channel signals generated by the complex controller tasks with the intention of stealing sensitive information and secrets of the system. As a matter of fact, due to the *deterministic schedules* of most RTS, an attacker can extract certain system information using signals such as timing parameters of the task [21], thermal profiles [22], secret keys [23] and cache access pattern [24]. Such side channel attacks require the adversary to perform monitoring activities for a *certain period of time to improve the accuracy* of the extracted information. Restarting make it possible to prevent from *stabilizing* the attacker task before it can extract a certain degree of *useful* information.

3) Denial-of-Service (DoS) Attacks: DoS attacks is a category of attacks where the attacker exhausts the resources of the system thereby preventing the access for the legitimate user. DoS attacks can occur in two forms; *i.e.,* system-level and network-level. In system-level attacks, attacker tries to exhaust all the memory, consume all the processor utilization or fill all the disk spaces. These eventually prevent the real-time applications from functioning. The network-level DoS relies on sending an overwhelming influx of request to overflow the target's resources. Flood of TCP/SYN packets, overflowing the data buffer and reflective pings are examples of network DoS attacks.

Restarting can always claim all the stale resources back and enable the system to continue operation. In the RTS context, in order to perform a system-level DoS attack, the attacker may need to monitor the execution profile for the tasks using, say, side-channel attacks and then override the task code with malicious DoS routines. Besides, the DoS routines also need some time to saturate the system resources. It is worth noting that combining all these steps to launch successful attacks requires *considerable* time.

With a correct estimation of the capabilities of the attacker (how much traffic they generate or at what rate they can exhaust system resources), the system designer can formulate the cost function for the attack and find an optimal restart time (refer to Appendix). Thus we assert that the restart-based method can be a potential solution to prevent (as well as recover) the system from many DoS attacks.

VI. Implementation and Evaluation

In this section we evaluate ReSecure with a proof-of-concept implementation. Although our prototype is implemented on an ARM-based development board and embedded Linux, the proposed method can be ported into different RTOS and platforms without loss of generality. Source codes used in the experiments are available online[7]. Table I lists the details about the implementation and parameters used in the experiments.

TABLE I
Implementation Platform and Experimental Parameters

Artifact/Parameter	Values
Complex unit platform	ARM Cortex-A8 1GHz, 512MB RAM
Safety unit platform	Intel Core i7, 8GB RAM
Complex unit OS	Real-time Linux kernel 4.4.12-ti-rt-r30
Complex unit reboot time, T_R	13 s
Periodic restart timeout, T_{Timer}	120 s
Period of the controller tasks, T_{ct}	[10 ms, 100 ms]
Watchdog timeout	Control task period

A. Experimental Setup

Our prototype implementation is developed in C and compiled using the GNU C compiler with optimization level 2 (*e.g.,* option `-O2`). We do not use any nonstandard external libraries and the implementation has no dynamic memory allocations or recursion.

1) Physical Plant: As a physical system we consider a 3 degree of freedom (3 DOF) helicopter. We use a known model of the system obtained from prior work [25] and perform hardware-in-the-loop (HIL) simulations [26] to capture the exact behavior of the sensors and physical plant. Pitch, elevation and travel angle are reported to the safety and complex controller as sensor readings.

This plant is considered *safe* as long as the wings of the fans do not hit the horizontal surface on which the 3-DOF helicopter is fixated. The linear inequality in Eq. (1) defines the set of states in which system is *safe*.

$$elevation \pm 1/3 \times pitch > -0.3. \quad (1)$$

[6]A similar probabilistic analysis is performed in literature [18] to find the optimal restart time for software rejuvenation purposes.

[7]https://github.com/mnwrhsn/restart_n_secure_cps.

By applying the Simplex method to the linear model of the system and the safety condition, we obtain the safety controller (*i.e.,* a proportional feedback controller) and the decision logic for decision module.

2) Complex Unit: We use a BeagleBone Black (BBB) development board[8] (ARM Cortex-A8 1GHz processor, 512MB RAM, 4GB on-board eMMC storage) as our complex unit. The complex unit uses an embedded Debian GNU/Linux console image as the OS. Since the vanilla Linux kernel is poorly suited for executing real-time applications directly, we enable the real-time capabilities by applying the RT-PREEMPT patch [27] (kernel version 4.4.12-ti-rt-r30) that is known to respect real-time constraints in embedded platforms such as ARM [28].

The OS in the complex unit uses priority-based preemptive scheduling powered by RT-PREEMPT patch. The highest priority task that is eligible to run in the system is always scheduled by the OS. Each real-time controller task has a period $T_{ct} \in [10 \text{ ms}, 100 \text{ ms}]$. The periodic controller tasks are suspended after the completion of corresponding instances using the `clock_nanosleep()` system call. Also we set the program executable as a *startup cron job* (using `crontab` utility) so that it can execute as soon as the system reboots. After reboot we load the fresh (*viz.,* uncompromised) OS image from the BBB's on-board eMMC storage[9]. From our experiments we find that, it takes at most 13 seconds (on average 12.1379 seconds with 0.3509 standard deviation) to restart the BBB and make the complex controller's code operational.

In order to prevent unauthorized disk access and protect the file systems from corruption during the asynchronous restart events, we mount the file systems of the complex unit's OS as read-only. For this, we modify the `/etc/fstab` file and ensure that the root file systems will always be mounted as read-only[10] after each reboot.

3) Safety Unit: The safety unit (*e.g.,* safety controller and decision module) is implemented on a x86 machine[11] (Intel Core i7 processor, 8GB RAM) running Ubuntu Linux 14.04. In our current implementation, the complex unit communicates with the safety unit using a fixed IP. The sensor and actuation commands (*viz.,* travel, elevation and pitch angle of the helicopter) are fed into both safety and complex controller. We implement the decision module as a multithreaded application that toggles the control between safety and complex controller according to *i)* availability of complex unit and *ii)* the decision logic obtained from applying Simplex method to the system model and the safety conditions.

4) Monitoring Unit: As we have mentioned in Section IV-A2, the flexible architecture of ReSecure allows the designer to integrate the desired security methods in order to monitor the system resources/components that they care about. For illustration purposes only, we develop a minimalistic monitoring unit. The monitoring unit is implemented as a periodic task of its own (with a period of 2 seconds). Specifically, the monitoring unit in our current implementation stores the legitimate kernel modules on startup. Later, during each of its periodic invocation at time t, it scans the currently running kernel modules at time t (*e.g.,* available through `/proc/modules` interface) and compares with the stored information. In the event when any mismatch or anomaly is detected the monitoring unit issues a restart request through `SysRq` command (*e.g.,* using `/proc/sysrq-trigger`).

5) Implementation of Restart Events: As mentioned in Section IV-A, the restart events in the complex unit are triggered by the watchdog and periodic timers. We implement the watchdog timer using `ualarm()` system call and update the timer in each periodic invocation of controller tasks. Once activated, the watchdog timer is periodically accessed. Failing to update the watchdog within the timeout duration will force the system to restart.

External periodic hardware timers can be implemented by using the BBB's general purpose input/outputs (GPIO) pins. For instance, one can interface a hardware timer with BBB's GPIO interface and generate an interrupt to trigger the restart when the timer expires. Our current prototype, however, does not use any external hardware timers. Instead, we use Linux high resolution timers (*viz.,* `hrtimers` [29]) to implement the periodic restart. In particular, we develop a loadable kernel module that activates the timer on startup and restarts the system as soon as the timer expires. In our experiments we set the timeout for periodic restart $T_{Timer} = 120$ seconds.

B. Experience and Evaluation

In the following we analyze the effectiveness of ReSecure by mimicking some of the attacks described in the preceding section. In particular we illustrate how different restarting strategies can be useful to recover from these attacks.

1) Recovery by Watchdog Timeout: As mentioned in Section III, an attacker can capture the control tasks by tracking the execution profile using side channel attacks [21] that exploit the deterministic behavior of the real-time scheduling. We perform system-level DoS by launching a *fork bomb*[12] attack on the complex controller. This DoS attack utilizes Linux `fork()` system call and recursively replicate itself. Due to the CPU overload some (or all) of the control tasks may miss their deadlines. In this experiment we assume that the attacker hijacks the highest priority real-time control task and is able to execute malicious codes with the privilege similar to those of legitimate tasks (*viz.,* `root`). Recall that we update the watchdog timer once during every invocation of controller tasks. Since the highest priority controller task is itself compromised with self-replicating malicious codes, the watchdog timer is not updated and a restart event is triggered. The system is recovered from the attack within less than approximately 14 seconds (including $T_R = 13$ seconds reboot time). Again, the safety constraints in Eq. (1) are not violated since the safety controller takes the control during restart.

2) Recovery by Monitoring Unit and Periodic Restart: Similar to previous experiment, we assume that attacker can hijack the control tasks and is able to inject malicious codes. However, unlike Section VI-B1 where the attacker overrides the highest priority task, here the attacker may take control over any arbitrary priority task (referred to as victim task). This is particularly useful for the attacker to simply lodge itself in the system and glean sensitive information [21]. To demonstrate the attack, we inject malicious code (*viz.,* a kernel-level malware [30] that intercepts every `read()` system call) inside victim task. Since the original malware was developed for the x86 architecture, we modify the source code and port it for ARM. We also slightly modify the malware logic so that instead of making any changes, it silently lodges in the

[8]https://beagleboard.org/black.

[9]Technically it is possible to override/corrupt the on-board OS image of BBB. However, this requires physical intervention to hard-flash the board by using the reset (*viz.,* USER/BOOT) switch and may not be easily doable in practice. Hence, we consider on-board eMMC as a secure storage to load the OS image after each restart.

[10]By some careful engineering it could be possible to make file system writable even if it is mounted as read-only. In our current prototype, however, we do not consider this issue. As we have mentioned in Section IV, the OS in complex unit can also be loaded from read-only physical disc that can only be modifiable by external access.

[11]We intend to port the safety unit in an embedded platform in future work.

[12]http://malware.wikia.com/wiki/Fork_bomb.

system and extracts information (*i.e.,* representing side channel attacks).

The system can recover from this attack by two means, either the monitoring unit detects the presence of unauthorized entity and/or by periodic restart. We experiment with both alternatives. In our experiments we consider that monitoring unit remains uncompromised and periodically (with period $T_{MU} = 2$ seconds) scans the currently running kernel modules and triggers a restart if any malicious entity is found. In both cases, the system recovers from the intrusion within $T_R + \delta$ seconds of attack where $\delta = T_{MU} - t$ if the restart is triggered by monitoring unit[13] or $\delta = T_{Timer} - t$ when system reboots due to periodic timeout where $t \in [0, T_{Timer}]$ represents relative point of intrusion.

VII. Discussion

Despite the fact that ReSecure provides an integrated approach to guarantee safety and security in real-time CPS, this is obviously not a silver bullet for solving all security problems. In what follows we briefly discuss the defense mechanisms of ReSecure against different threat model as well as the limitations of the current framework with possible directions of improvements.

A. Threat Model

The underlying detection algorithm used in the monitoring unit may have false positives that will trigger unwanted restarts. A few false positives will not have a significant impact on the performance. False negatives are also mitigated by the periodic timers that set an upper bound on the detection time and restart the system independent of the monitoring unit or watchdog timers. Hence, the basic safety and security premise of ReSecure does not depend on having a *perfect* monitoring unit.

There may be cases where an advanced attacker performs a series of attacks with a view to destabilizing the system. Whether the attack is being detected by the monitoring unit and/or by watchdog timeout, this will cause the system to be restarted *frequently*. Despite the fact that frequent restart may affect (or degrade) system performance, ReSecure ensures that the system will still remain safe during the sporadic unavailability of the complex unit. This kind of attack, however, will require the adversary to intrude into the system and launch series of attacks as soon as the fresh OS image is loaded. Again, we can argue that this will be *difficult* and *unlikely* in practice.

B. Limitations and Improvement

ReSecure requires a dedicated computing module as a safety unit due to periodic/asynchronous unavailability of complex unit. However, this overhead and performance loss comes with increased security guarantees – that might be acceptable for many security-critical RTS.

Although ReSecure can recover from certain types of DoS, code injection and side channel attacks, the restart mechanism does *not* prevent intrusion. As we have illustrated in the Appendix, the frequency of periodic restarts is calculated based on the prior knowledge of exploits. Therefore, ReSecure may *not* work as expected for *zero-day vulnerabilities*. For instance, the attacker may launch successful attacks between two consecutive restart events. However, compared to non-restart based mechanisms, the proposed framework ensures a clean (*e.g.,* uncompromised) system state every once in a while with *guaranteed* safety at the cost of restarts.

Given that certain side channel attacks are less effective if we are utilizing the restart mechanism, ReSecure in its current form, however, does not ensure *zero* information leakage from such attacks. This is because, there is always the possibility that the attacker can extract sensitive information between consecutive restarts. This issue can be mitigated by *randomizing* execution patterns and system configurations after every restart. For example, researchers have proposed schedule obfuscation method aiming at randomizing the real-time task schedule [31], cache random-eviction and cache random permutation [32] as well random key distribution [33]. Such techniques can be utilized along with ReSecure to randomize system parameters after each restart and further reduces the chance of information leakage. We intend to incorporate such techniques on top of the restart-based recovery mechanism in future work.

VIII. Related Work

The Simplex architecture [10], [34] has been used extensively to provide verified design with unverified logic. Variants of Simplex have been proposed to account for physical system failures [35], as well as faults in OS or processor [9]. These works have focused on fault-tolerance and are not concerned with the security issues in RTS.

The notion of restarting has been used in some studies to recover from faults in safety-critical systems. System-level Simplex is the first architecture that is able to restart the complex controller [9] with formal safety guarantees. Reset-based recovery for safety-critical systems [13] has further analyzed restarting as a recovery mechanism. Both of these approaches utilize extra hardware to isolate safety subsystem from faults in complex subsystem. The objective is to recover the system safely from *non-adversarial software faults* in the controller and underlying layers (OS and firmware). To our knowledge, ReSecure is the first work that extensively considers enabling *systematic restart-based recovery with guaranteed safety* to ***secure*** the RTS.

Although sophisticated adversaries and malware developers are able to overcome air-gaps in recent years, security issues considering hard real-time constraints have not been studied comprehensively in literature. Recent work [16], [17], [36] on dual-core based hardware/software architectural frameworks aim to protect RTS against security vulnerabilities. A similar line of work exists [37] where authors leverage the deterministic execution behavior of RTS and use Simplex architecture to detect intrusion while guaranteeing the safety. However the solution proposed by that work is limited to time-based intrusion detection methods only. In contrast, ReSecure utilizes the concept of restarting on top of Simplex and able to prevent and/or recover from large classes of attacks.

IX. Conclusion

The evidence from recent successful attacks on UAVs [38], automobiles [39] and industrial control systems [1] indicate that security violations are becoming more common in RTS. In this work, we are moving towards developing an integrated security-aware RTS and illustrate restart as a viable mechanism to ensure security in such safety-critical systems. Designers of such systems can now evaluate the necessary trade-offs between control system performance degradation and increased security guarantees – thus improving the overall design of RTS in the future.

[13]To be specific, the actual recovery time in this case is $T_R + T_{Detect}$ where $T_{Detect} < T_{MU}$ refers the time to detect the attack by the monitoring unit. However, in our experiments we do not profile exact detection time and provide an upper bound instead.

REFERENCES

[1] N. Falliere, L. O. Murchu, and E. Chien, "W32. stuxnet dossier," *White paper, Symantec Corp., Security Response*, vol. 5, p. 6, 2011.

[2] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno *et al.*, "Comprehensive experimental analyses of automotive attack surfaces." in *USENIX Security Symposium*. San Francisco, 2011.

[3] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham *et al.*, "Experimental security analysis of a modern automobile," in *IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 447–462.

[4] G. Candea and A. Fox, "Crash-only software," 2003.

[5] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox, "Jagr: An autonomous self-recovering application server," in *Proceedings of the Autonomic Computing Workshop*. IEEE, 2003, pp. 168–177.

[6] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot-a technique for cheap recovery." in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, ser. OSDI, vol. 4, 2004, pp. 31–44.

[7] L. Sha, "Using simplicity to control complexity," *IEEE Software*, vol. 18, no. 4, pp. 20–28, 2001.

[8] L. Sha, R. Rajkumar, and M. Gagliardi, "Evolving dependable real-time systems," in *IEEE Aerospace Applications Conference*, vol. 1. IEEE, 1996, pp. 335–346.

[9] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha, "The system-level simplex architecture for improved real-time embedded system safety," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2009, pp. 99–107.

[10] L. Sha, "Dependable system upgrade," in *IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 1998, pp. 440–448.

[11] ——, "Using simplicity to control complexity." IEEE Software, 2001, pp. 20–28.

[12] T. L. Crenshaw, E. Gunter, C. L. Robinson, L. Sha, and P. Kumar, "The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures," in *IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2007, pp. 400–412.

[13] F. Abdi, R. Mancuso, S. Bak, O. Dantsker, and M. Caccamo, "Reset-based recovery for real-time cyber-physical systems with temporal safety constraints," in *IEEE Conference on Emerging Technologies Factory Automation (ETFA)*, 2016.

[14] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, Jan. 1973.

[15] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

[16] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha, "Securecore: A multicore-based intrusion detection architecture for real-time embedded systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013, pp. 21–32.

[17] M.-K. Yoon, S. Mohan, J. Choi, and L. Sha, "Memory heat map: anomaly detection in real-time embedded systems using memory behavior," in *ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.

[18] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *International Symposium on Fault-Tolerant Computing (FTCS-25)*. IEEE, 1995, pp. 381–390.

[19] J. Viega and H. Thompson, "The state of embedded-device security (spoiler alert: It's bad)," *IEEE Security & Privacy*, vol. 5, no. 10, pp. 68–70, 2012.

[20] B. Bencsáth, G. Pék, L. Buttyán, and M. Felegyhazi, "The cousins of stuxnet: Duqu, flame, and gauss," *Future Internet*, vol. 4, no. 4, pp. 971–1003, 2012.

[21] C.-Y. Chen, R. B. Bobba, and S. Mohan, "Schedule-based side-channel attack in fixed-priority real-time systems," University of Illinois at Urbana Champaign, http://hdl.handle.net/2142/88344, Tech. Rep., 2015, [Online].

[22] C. Bao and A. Srivastava, "A secure algorithm for task scheduling against side-channel attacks," in *International Workshop on Trustworthy Embedded Devices*. ACM, 2014, pp. 3–12.

[23] K. Jiang, L. Batina, P. Eles, and Z. Peng, "Robustness analysis of real-time scheduling against differential power analysis attacks," in *IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2014, pp. 450–455.

[24] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the Annual International Symposium on Computer Architecture*, ser. ISCA. New York, NY, USA: ACM, 2007, pp. 494–505. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250723

[25] S. Rajappa, A. Chriette, R. Chandra, and W. Khalil, "Modelling and dynamic identification of 3 DOF Quanser helicopter," in *International Conference on Advanced Robotics (ICAR)*, 2013, pp. 1–6.

[26] J. A. Ledin, "Hardware-in-the-loop simulation," *Embedded Systems Programming*, vol. 12, pp. 42–62, 1999.

[27] L. Fu and R. Schwebel, "Real-time linux wiki," https://rt.wiki.kernel.org/index.php/rt_preempt_howto, [Online].

[28] J. Altenberg, "Using the realtime preemption patch on ARM CPUs," in *Real-Time Linux Workshop*, 2009, pp. 28–30.

[29] J. Corbet, "The high-resolution timer (API)," Article on LWN. net, http://lwn.net/Articles/167897, 2006, [Online].

[30] M. Phillips, "Attack via kernel module," https://github.com/mrrrgn/simple-rootkit, [Online].

[31] M.-K. Yoon, S. Mohan, C.-Y. Chen, and L. Sha, "TaskShuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016, pp. 1–12.

[32] T. Zhang and R. B. Lee, "New models of cache architectures characterizing information leakage from cache side channels," in *Proceedings of the 30th Annual Computer Security Applications Conference*. ACM, 2014, pp. 96–105.

[33] H. Chan, A. Perrig, and D. Song, "Random key predistribution schemes for sensor networks," in *Security and Privacy, 2003. Proceedings. 2003 Symposium on*. IEEE, 2003, pp. 197–213.

[34] D. Seto and L. Sha, "A case study on analytical analysis of the inverted pendulum real-time control system," DTIC Document, Tech. Rep., 1999.

[35] X. Wang, N. Hovakimyan, and L. Sha, "L1simplex: fault-tolerant control of cyber-physical systems," in *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*. ACM, 2013, pp. 41–50.

[36] D. Lo, M. Ismail, T. Chen, and G. E. Suh, "Slack-aware opportunistic monitoring for real-time systems," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014, pp. 203–214.

[37] 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 *ACM international conference on High confidence networked systems*. ACM, 2013, pp. 65–74.

[38] D. P. Shepard, J. A. Bhatti, T. E. Humphreys, and A. A. Fansler, "Evaluation of smart grid and civilian uav vulnerability to gps spoofing attacks," in *Proceedings of the ION GNSS Meeting*, vol. 3, 2012.

[39] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham *et al.*, "Experimental security analysis of a modern automobile," in *IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 447–462.

APPENDIX
ANALYZING THE IMPACT OF RESTART

For a given system we represent the set of possible exploits with $\mathcal{E} = \{e_1, e_2, ..., e_N\}$. Each, e_j is associated with a probability of occurrence $P(e_j)$ which specifies how common the exploit is. From definition we have $\Sigma_j P(e_j) = 1$. Here, T_R is the time for a full complex unit restart, T_{Com} is the time the attacker needs to re-launch the attack (During this time system is not compromised yet), T_{Detect} is the time it takes for the monitoring unit to detect the attack and initiate a restart; T_{Timer} is the fixed time for the periodic timer to restart the system.

For each exploit, T^j_{Com} and T^j_{Detect} are defined as random variables associated with a probability density function $P^j_{Com}(t)$ and $P^j_{Detect}(t)$. The shape of $P^j_{Com}(t)$ depends on vulnerability level of the platform and the effectiveness of e_j. In a similar fashion, the shape of the $P^j_{Detect}(t)$ depends on the effectiveness of the intrusion detection techniques used in the monitoring unit for detecting e_j. Furthermore, a monotonically increasing function called $Damage^j(t)$, is associated with each exploit that defines how much damage e_j causes within t seconds after being effective. The probability density functions and damage function are assumed to be provided by the system designer using the analytic and experimental data and from the knowledge of the system, exploits and detection mechanisms.

In this setting, the only configurable variable is the restart period of the periodic timer, *i.e.,* T_{Timer}. Very short restart periods can limit the attackers; however, it also reduces the time that the complex unit is available. On the other hand, a long T_{Timer} can increase the risk of damage to the system by the exploits. In what follows, we show how to find the optimal T_{Timer} for a given system.

The expected time that the complex unit is available and not compromised can be computed from the following:

$$Exp(T^j_{Com}) = \int_{t=0}^{T_{Timer}} t \cdot P^j_{Com}(t) \cdot dt + T_{Timer} \cdot \int_{t=T_{Timer}}^{\infty} P^j_{Com}(t) \cdot dt. \quad (2)$$

Besides, the expected time that it takes for the monitoring unit to detect an exploit is given by

$$Exp(T^j_{Detect}) = \int_{t=0}^{T_{Timer}} \Big(\int_{\tau=0}^{T_{Timer}-t} \tau \cdot P^j_{Detect}(\tau) \cdot d\tau + (T_{Timer} - t) \cdot \int_{\tau=T_{Timer}-t}^{\infty} P^j_{Detect}(\tau) \cdot d\tau \Big) \cdot P^j_{Com}(t) \cdot dt. \quad (3)$$

The expected restart period for an exploit can be obtained as follows

$$Exp(T^j_{Period}) = T_R + Exp(T^j_{Com}) + Exp(T^j_{Detect}). \quad (4)$$

Restarting the system, increases the unavailable time of the system. The expected percent of the time that the system is not available can be computed as follows

$$Exp(\text{Unavailability}) = \sum_{e_j \in E} P(e_j) \Big(1 - \frac{Exp(T^j_{Com})}{Exp(T^j_{Period})}\Big).$$

For a fixed T_{Timer} value, the expected damage of an exploit can be computed using the following probabilistic distribution of the time to compromise and the time to detect the exploit.

$$Exp(Damage^j) = \int_{t=0}^{T_{Timer}} \Big(\int_{\tau=0}^{T_{Timer}-t} Damage^j(\tau) \cdot P^j_{Detect}(\tau) \cdot d\tau + Damage^j(T_{Timer} - t) \int_{\tau=T_{Timer}-t}^{\infty} P^j_{Detect}(\tau) \cdot d\tau \Big) P^j_{\text{Com}}(t) \cdot dt \quad (5)$$

Therefore, total expected damage from all the exploits on the system is given by

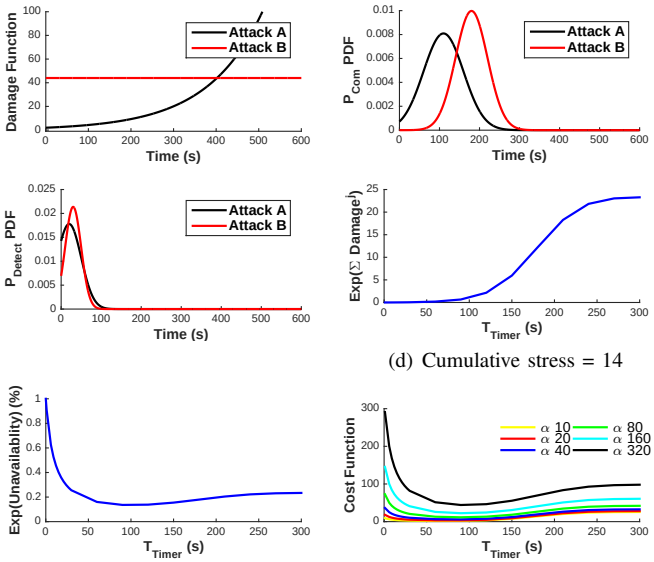
$$Exp(\text{Total Damage}) = \sum_{e_j \in E} P(e_j) \cdot Exp(Damage^j).$$

The expected total damage of exploits and the unavailable time of the system are inversely related. Restarting the system more frequently would decrease the expected damage from security exploits and at the same time will reduce the availability of complex controller. We formulate this as a minimization problem to find a T_{Timer} that minimizes the following $Cost$ function:

$$Cost(T_{Timer}) = Exp(\text{Total Damage}) + \alpha \cdot Exp(\text{Unavailability}).$$

In the above, parameter α determines the importance of availability versus the of risk being compromised. This is a design choice based on the applications of the physical systems under control. Note that, more available time for the complex controller can increase system performance. For systems in which security is far more important than performance, a small value for α is desirable. This leads to a smaller value for T_{Timer}, thus leading to a higher security but less available time and control performance.

In Figs. 3 and 4, we illustrate the above analysis for two examples. In both examples two possible exploits are considered with $P(e_1) = P(e_2) = 0.5$. Both exploits in Example 1 have a longer compromise time and shorter detection time than the exploits in the Example 2 (Based on P_{Com} and P_{Detect} functions).



(d) Cumulative stress = 14

Fig. 3. Example 1, $T_R = 10s$, P(A) = P(B) = 0.5

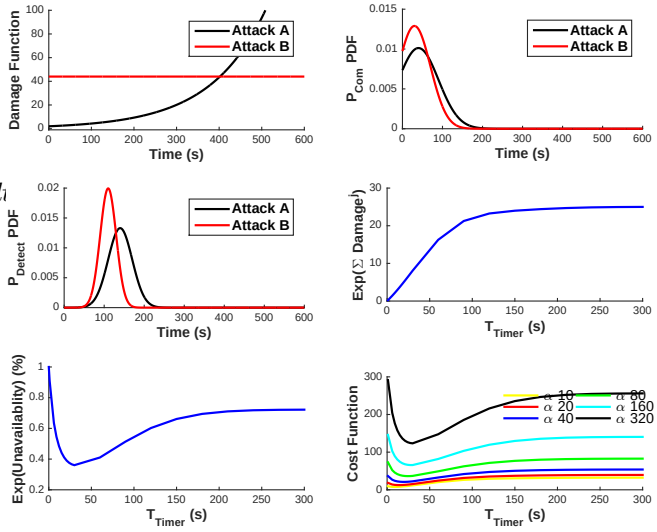


Fig. 4. Example 2, $T_R = 10s$, , P(A) = P(B) = 0.5

As seen in the plot of the cost functions for two examples, the optimal T_{Timer} (the value of T_{Timer} that minimizes the $Cost$ function) for Example 1 is smaller than the Example 2. Intuitively, the system in Example 1 needs less frequent restarts. Moreover, the different patterns in the compromise and detection times in the two examples, leads to a larger cost for the system in the second example for the same values of α. For instance, in case of $\alpha = 160$ (plotted with cyan color), minimum cost is 11.2 for $T_{Timer} = 103$ seconds in Example 1 and 65.7 for $T_{Timer} = 30$ seconds in Example 2.