Design of Adaptive Security Mechanisms for Real-Time Embedded Systems

Mehrdad Saadatmand, Antonio Cicchetti, Mikael Sjödin

Mälardalen Real-Time Research Centre (MRTC)
Mälardalen University, Västerås, Sweden
{mehrdad.saadatmand, antonio.cicchetti, mikael.sjodin}@mdh.se

Abstract. Introducing security features in a system is not free and brings along its costs and impacts. Considering this fact is essential in the design of real-time embedded systems which have limited resources. To ensure correct design of these systems, it is important to also take into account impacts of security features on other non-functional requirements, such as performance and energy consumption. Therefore, it is necessary to perform trade-off analysis among non-functional requirements to establish balance among them. In this paper, we target the timing requirements of real-time embedded systems, and introduce an approach for choosing appropriate encryption algorithms at runtime, to achieve satisfaction of timing requirements in an adaptive way, by monitoring and keeping a log of their behaviors. The approach enables the system to adopt a less or more time consuming (but presumably stronger) encryption algorithm, based on the feedback on previous executions of encryption processes. This is particularly important for systems with high degree of complexity which are hard to analyze statically.

Keywords: Security, real-time embedded systems, runtime adaptation, trade-off.

1 Introduction

Security is gaining more and more attention in the design of embedded systems. Embedded systems are nowadays everywhere. They are used in controlling systems of power plants, vehicular systems and medical devices, as well as, mobile phones and music players. The operational environment, physical accessibility, and mobility of embedded systems make them prone to certain types of attacks which might be less relevant for ordinary computer systems, such as side channel attacks, time and power analysis to determine security keys and algorithms [1]. Also, the increasing use of embedded devices as parts of networked and interconnected devices makes them prone to new types of security issues [2].

On the other hand, introducing security in embedded systems requires careful considerations, trade-off analysis, and balance with other aspects such as performance, power consumption, and so on. This is mainly due to the resource constraints and limitations that these systems have. For example, choosing an

encryption algorithm that performs heavy computations, requires lots of memory, and, as a result, consumes more energy, may impair the correct functionality of the system and violate its specified requirements. This is basically because of the fact that non-functional requirements, such as security, are not independent and cannot be considered in isolation [3]. Therefore, it is important to understand the impacts and consequences of designed security mechanisms on other aspects of the systems.

In real-time embedded systems, where timing requirements are critical, choice of security mechanisms is important in terms of satisfaction of timing requirements. One way to achieve this, is to find a security mechanism that fits and matches the timing requirements of the system (e.g., by performing timing analysis), and then implement it [4]. This method leads to a static design in the sense that a specific security mechanism, which is analyzed, and thus, known to execute within its allowed time budget, is always used in each execution. However, this method may not be practical for systems with high complexity, which are hardly analyzable or systems with unknown timing behaviors of their components. Instead, for such systems, an adaptive approach to select appropriate security mechanisms, based on the state of the system, can be used to adapt its behavior at runtime and stay within the timing constraints. In this paper, we introduce this approach, and describe its implementation for selecting appropriate encryption algorithms at runtime (in terms of their timing behaviors) in an adaptive way, using OSE real-time operating systems [5]. To this end, the timing behavior of each execution of the encryption procedures is logged, and used as feedback for selecting a more suitable encryption algorithm in the next execution.

The rest of the paper is structured as follows. Section 2 describes the motivation and background of this work. In Section 3, we discuss the approach, describe how the adaptation mechanism in the proposed approach works. Implementation and experimental results are also explained in this section. Section 4, discusses the context where the proposed approach can be more applicable and suit well. In Section 5, related work is discussed, and finally in Section 6, conclusions are drawn, and pointers to future directions of this work are provided.

2 Background and Motivation

Designing security for real-time embedded systems is a challenging task. This is due to the fact that security features have impacts on other aspects of the system, such as timing, and if these impacts are not identified, analyzed, or managed properly, they can lead to violations of other non-functional requirements, and thus failure of the system. While in small and simple systems timing and schedulability analysis, covering security algorithms, can be done to ensure satisfaction of timing requirements, when it comes to very complex systems, such static and offline analyses might not be practical and feasible [6]. Even in cases where they are feasible, the results of such analyses may be invalidated at run-time, due to several factors such as transient loads, difference between the ideal execution

environment (taken into account for analysis) and the actual one, which leads to violation of the assumptions that are used to perform analysis [7].

Telecommunication systems are examples of systems with high complexity that require massive execution capacity and have security requirements. Due to the complexity of these systems, the main focus in their design is to be able to handle massive connection requests and data loads that arrive in a bursty and unpredictable fashion, than just trying to merely perform analysis of all possible conditions in the system [8]. Therefore, it is important that such systems are designed in an adaptive way, so that they can reconfigure themselves at runtime to continue providing their services, although under different Quality-of-Service (QoS) levels. Also, most of the real-time tasks in these systems have soft deadlines. This means it is acceptable, in general, for the functionality of the system, if some tasks complete their jobs within a reasonable margin after their deadlines, and the result of a single deadline miss is not catastrophic.

OSE is a Real-Time Operating System (RTOS) developed by Enea [9], which is used heavily in telecommunication systems, especially by Ericsson, from Radio Network Controllers, and Radio Based Stations (RBS) to mobile devices. In this paper, focusing on the needs of such systems that require runtime adaptation, an approach is suggested to select encryption algorithms at runtime based on how they behave in terms of their time constraints. To implement and evaluate the approach, OSE is used as the base platform. It is an example of a RTOS which is designed from scratch to provide the necessary determinism level required for fault-tolerant real-time embedded systems with high availability, particularly in telecommunication domain.

3 Approach

To design a system that can adapt itself and adjust the balance between its time constraints and security level, the approach depicted in Figure 1 is suggested.

When an application needs to encrypt some data, it sends the data along with the allowed execution time for the encryption procedure to the main encryption process. Based on the received time constraint, this process will then try to encrypt the data with an appropriate encryption algorithm, by invoking the corresponding process implementing that algorithm. This is done by first consulting the log information generated by the monitor process plus a pre-defined table for ranking of preferred encryption algorithms. An example of such a table is shown in Table 1. The table is used to capture the preferences for different encryption algorithms, but it has to be in descending order in terms of execution times. As we will see, this is important for the correct behavior of the system. Comparison of execution times for different encryption algorithms can be obtained from the result of studies such as [10], which has performed performance measurements of different encryption algorithms. This table is, therefore, filled by the user, using the result of such studies. For each algorithm in the table, there is a process that implements it (named in Figure 1 as 'process for encryption algorithm 1'...'process for encryption algorithm n').

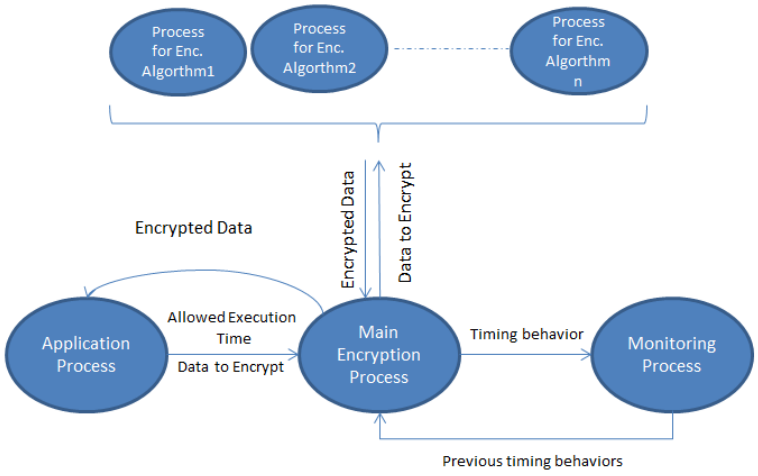

Fig. 1. Adaptive design of encryption algorithms

Rank	Encryption Algorithm
1	AES
2	3DES
3	DES
...	

Table 1. Preferred encryption algorithms in descending order in terms of execution times.

Then, when an appropriate encryption algorithm is decided (the following section describes in detail how this is done), the main encryption process passes the data to the process implementing that encryption algorithm to perform the actual encryption of the data. The time taken from the point that the main encryption process sends the plain data it has received from the application process to one of the processes implementing an encryption algorithm, until encrypted data is returned to the main encryption process by it, is sent as part of the log information to the monitoring process. This log information will be used as feedback in the next invocation of the main encryption process.

The assumption that is implicit in this design is that, when it is detected that an executing encryption algorithm is exceeding its allowed time budget, it is basically more costly to terminate it in the middle of the encryption procedure, and restart encryption of the data with another encryption algorithm, than just letting it finish its job, and instead use one with a lower execution time in the next invocation of encryption procedures. This is why the encryption algorithms in Table 1 need to be sorted according to their execution times. In this way,

the system first tries to encrypt all data with the algorithm at the top of the table. If it is observed that it cannot fulfill the specified time constraint, the second ranked algorithm (which has shorter execution time) will be chosen for encryption in the next invocation.

On the other hand, if an encryption algorithm completes its job sooner than its specified time limit, the unused portion of its time budget is used to calculate and determine whether it is feasible to go back to the previous higher ranked algorithm for the next encryption job or not. Using this approach, the system tries to adapt itself based on the feedback it receives regarding its timing behavior. Therefore, when a burst of processing loads arrive, the system adapts itself to this higher load, and when the processing load decreases, it can gradually go back to using more time-consuming (and presumably more secure) encryption algorithms. This is, for example, very useful in telecommunication systems, where lots of other services (than the one(s) using encryption) are active and need to be responsive at the same time, where task pre-emption and context switches increase dramatically and interfere with the encryption job.

3.1 Log Information and Adaptation Mechanism

The information that is logged by the monitor process has the following format:

Timestamp, Encryption algorithm, Time constraint, Actual execution time

An Example of the generated log information is shown in Table 2.

10360, AES, 50, 90
11800, 3DES, 80, 70
14500, 3DES, 60, 70
21353, DES, 60, 10
22464, 3DES, 90, 40
23112, AES, 50, 50
28374, AES, 60, 58

Table 2. Sample log information.

The table shows that the system has first performed encryption using AES algorithm, with a time constraint of 50 time units, but the actual execution time has been 90; in other words, it has violated its time constraint. Therefore, in the next execution, 3DES is automatically chosen for encryption (as it is the first lower ranked algorithm under AES in Table 1), which has finished its job in 70 time units while having 80 as its time constraint. Hence, no change in the encryption algorithm is observed and the same algorithm (3DES) is used for the third execution as well. The fifth and sixth rows in Table 2 (which represent the fifth and sixth executions) show that the system has chosen to apply a higher ranked encryption algorithm (fourth to fifth: DES->3DES, and fifth to sixth: 3DES->AES).

In the log that is kept in memory from the log information generated above by the monitoring process, if an encryption algorithm is again selected for the next invocation, its latest log record will replace the previous one. In other words, for each two consequent log records for a certain encryption algorithm, only the most recent one is kept. This is done to only keep the information that is necessary, which also leads to having only log information indicating changes of encryption algorithms being stored. Table 3, shows what is actually necessary from the generated log information shown in Table 2, for making adaptation decisions.

10360, AES, 50, 90
14500, 3DES, 60, 70
21353, DES, 60, 10
22464, 3DES, 90, 40
28374, AES, 60, 58

Table 3. Necessary portion of log information for making adaptation decisions.

Considering the last row from the log as:

ts, alg, t, e
(ts: timestamp, alg: encryption algorithm, t: time constraint, e: actual execution time)

the decision that the system should adopt a lower ranked algorithm is made using the following formula:

(i) $e > t \Rightarrow$ *move down in the encryption algorithms table and select the next algorithm with a lower rank.*

Also, considering the two log records described as follows:

ts(l), alg(l), t(l), e(l) : representing the last log record
ts(h), alg(h), t(h), e(h) : representing the log record for the first encryption algorithm with a higher rank that was used before the last log record

the decision to adopt a higher ranked algorithm is made using the following formula:

(ii) $e(l) < t(l) \wedge t(l) - e(l) > abs(e(h) - t(h)) \Rightarrow$ *move up in the encryption algorithms table and select the previous higher ranked algorithm.*

For example, in Table 3, applying formula (i) on row 1 (i.e., 90>50) shows that AES has taken more time than it was allowed to; therefore, a lower ranked algorithm (3DES) is used for the next invocation. However, for row 4, and the higher ranked algorithm just before it, which is AES at row 1, formula (ii) holds (i.e., 90-40>abs(90-50)); therefore, at the next invocation (corresponding to row 5), AES algorithm was used again.

3.2 Implementation Details

The implementation of the approach is done on OSE real-time operating systems [5]. The OSE edition that is used is OSE Soft Kernel (OSE SFK), which is a simulation of the actual environment to be downloaded to the target embedded hardware, and is possible to run on a host machine (e.g., on Windows or Linux).

The execution unit in OSE corresponding to a real-time task is called process. For all the processes depicted in Figure 1, an OSE process is implemented. OSE offers two types of processes: static and dynamic. Static processes are created upon system startup, cannot be terminated, and last as long as the system is up and running. On the other hand, dynamic processes can be created and killed on the fly by another process using OSE APIs. Main application, main encryption process, and monitoring process, are created as static processes. Each encryption algorithm is implemented as dynamic processes, which are, in turn, created by the main encryption process at runtime as needed. This is just a design choice to reduce the number of active processes in the system, as encryption algorithms can also be well created as static processes, in which case, the overhead of creating them for each invocation will be reduced, while increasing the total scheduling overhead and memory usage of the system.

An interesting feature of OSE is offering the concept of *load module*. Load modules are relocatable program units that can be loaded into a running system and dynamically bound to that system. A loadable module can be uploaded, rebuilt, and quickly downloaded while the remainder of the system continues to run [5]. A load module can be considered similar to Windows .exe or .dll files. Once installed, the program (consisting of one or more processes) that they contain, can be created and started. In the case of our system, this means that security designers can add new encryption algorithms to the system dynamically or update them on the fly, while the system is active and running. Such a feature is very important in high-availability systems, where updates and upgrades (e.g., patching security issues) should affect the up-time of the system to the least degree possible.

Also, the direct and asynchronous message passing mechanism in OSE, makes this real-time operating system a great choice for use in distributed systems. This further facilitates the scalability of systems that are built on OSE. Data between processes are passed as signals using three basic OSE APIs for signal passing: *send*, *receive*, and *alloc* (to create signals containing data). The Inter-Process Communication protocol (IPC) used in OSE, makes the location of processes transparent to the user; meaning that no matter whether two processes are located on the same board or on different ones, the communication between them is done using the same set of APIs and code. Using this feature, the proposed approach is implemented without the need to use shared memory among processes. The communication between processes is implemented using the three above-mentioned APIs. This brings along the possibility to deploy the processes shown in Figure 1 on different processors and boards without affecting the generality of the approach, which is important for highly-distributed real-time embedded systems such as telecommunication systems.

Using the aforementioned features, several signals have been defined for synchronization, and to pass data between processes. For example, signal HISTORY_INFO_REQUEST is defined which is sent from the main encryption process to the monitoring process to request log information. In case of receiving this signal, the monitoring process sends last log record, and the log record for the first encryption algorithm with a higher rank, used before the last record, to the main encryption process using HISTORY_INFO_REPLY signal. Using this signaling mechanism also allows to deploy processes on different nodes if needed. This is possible since the necessary information to make adaptation decisions such as the time it takes to encrypt is passed between processes as part of the signals, making the actual location of processes in different nodes unimportant and transparent for the approach to work. The rank table for encryption algorithms is actually implemented using enumerations in C/C++ in the main encryption process.

3.3 Evaluation

To test the behavior of the system, a tool called CPU Killer [11] was used to create desired percentage of CPU loads at desired times. Moreover, Optima, which is a debugging, profiling, and monitoring tool developed by Enea for OSE, was also used to monitor and observe, in the form of graphs and tables, CPU usage levels at different system ticks from the startup of OSE. The system was run two times: once without having adaptation and the second time using adaptation. At each time, CPU loads of 10%, 50%, 70%, and then back to 50%, and 10% were applied. The results are shown in Figure 2.

The columns for each log record identify: system time (ticks), encryption algorithm, time constraint (ticks), and actual execution time (ticks). As mentioned in the previous section, an enumeration in the form of *"enum algorithms { AES=1, THREEDES=2, DES=3 };"* was used to represent the information of Table 1 in the code. The logs are decorated here with additional marks to facilitate explanation and understanding of the results.

In case I, where no adaptation was applied, AES (as number 1 in the second column) algorithm is constantly used for encryption. This is because the goal is to provide the maximum level of security, and therefore, the system is designed to prefer and choose the topmost encryption algorithm (whenever possible) from the table, which represents the strongest one. The system was started while applying 10% CPU load, and as can be seen from the figure, encryption is done within its time constraint of 300 and no violation is observed. However, when CPU load is increased to 50%, encryption starts violating its time constraint. Violations are marked with * mark in the figure. In case of the first violation, it can be seen that encryption was completed in 305 ticks while the time constraint is 300 ticks. Violations get worse (with more time margins) at 70% CPU load. It is only after going back to 10% CPU load, that encryption can meet its constraint again.

In the second case (II), where adaptation was used, it can be easily understood at the first sight that the number of violations have decreased. The first

(I) No Adaptation		(II) With Adaptation	
10%:	580,1,300,258	10:%	584,1,300,276
	859,1,300,269		868,1,300,273
	1145,1,300,276		1155,1,300,277
	1429,1,300,274		1441,1,300,276
	1711,1,300,272		1719,1,300,268
50%:	2027,1,300,305*		2111,1,300,382A
	2474,1,300,429*		2331,2,300,203
	2918,1,300,430*	50%:	2770,1,300,428*
	3364,1,300,432*		2996,2,300,211
	3813,1,300,435*		3229,2,300,214
70%:	4482,1,300,658*		3452,2,300,203
	5399,1,300,900*		3706,2,300,243
	6301,1,300,881*	70%:	4105,2,300,381*
	7199,1,300,880*		4500,3,300,380*
	8115,1,300,898*		4880,3,300,361*
50%:	8640,1,300,505*		5257,3,300,360*
	9086,1,300,429*		5639,3,300,362*
	9529,1,300,428*		5950,3,300,294A
	9974,1,300,430*	50%:	6162,3,300,194
10%:	10285,1,300,296		6380,2,300,197
	10556,1,300,261		6594,2,300,199
	10819,1,300,251		6810,2,300,200
	11083,1,300,255		7023,2,300,200
	11349,1,300,256		7236,2,300,199
			7450,2,300,199
			7641,2,300,177
		10%:	7754,2,300,103
			8026,1,300,262
			8307,1,300,270
			8572,1,300,255
			8846,1,300,264

Fig. 2. Results of running the system with and without adaptation.

violation occurs when the CPU load is set to 50%, however, the system adapts itself to this new load and uses 3DES (as number 2 in the second column), which helps the system to perform within its time constraint again. When CPU load is set to 70%, violations are again observed. Therefore, the system adapts itself by using DES (as number 3 in the second column) instead of 3DES, to stay within the time constraint. Even using DES, the system still fails to meet its constraint, however, within a smaller time margin. In spite of violations, since no lower rank algorithm than DES was defined in this experiment, the system keeps using it as the last possible choice. When the CPU load is reduced back to 50%, using the two formulas described at the beginning of the section, the system realizes that it can go back to using a higher ranked algorithm (3DES in this case; hence number 2 is again observed in the second column as the used algorithm) without causing any violations. Finally, by reducing the CPU load further to 10%, the systems goes back to using AES again.

Two rows are marked with A (at time 2111 and 5950) to show anomaly in the results. The first one which shows a violation for AES algorithm under 10% load, while previous rows show that it can meet its constraint under this load. This can be due to some background services doing some work in the system at that point, which has affected AES to perform encryption as before, and thus,

resulted in a violation. Or, it can be because of a relatively slow movement of the slider in the CPU Killer application, which is used to raise the CPU load to 50% manually, and thus resulting in the system working in a CPU load between 10% and 50% for a short while at that moment. This can also be the reason for the anomaly observed in the next row marked with A (at time 5950). In this case, this row shows that DES has managed to complete within its time constraint under 70% CPU load. While, it could not perform so in the previous rows related to this algorithm (having number 3 in the second column). This can again be because of the relatively gradual decrease of the CPU load from 70% to 50%, causing the system to work at some CPU load in between.

Also another interesting observation from this result is that, as the consequence of using adaption, more encryption jobs have been performed in the second case (II), under a shorter period of time.

4 Discussion

Our suggested approach and the way we implemented it, gives this flexibility to have different time constraints for each invocation of encryption procedures. This may not, however, be needed in all systems, and only a fixed value (e.g., originating from a system level requirement) might seem to be enough for many situations, but other systems can well benefit from this flexibility.

The whole adaptation mechanism can also be used as an option; in the sense that, if a system detects certain patterns in CPU load variations and violation of timing constraints in the applied encryption algorithm, it can *turn on* adaptation mechanism and let the system decide which encryption algorithm to use. Moreover, use of the suggested adaptation mechanism may be most beneficial when there are many requests for encryption and frequency of CPU load changes are such that they make the overhead of adaptation mechanism acceptable. On the other hand, if there are only a few encryption requests or there are not big changes in CPU load (or the range of changes is very small and known beforehand), using a fixed encryption algorithm may be more desirable (to remove the overhead cost of adaptation).

The security level of the system, originating from the choice of encryption algorithms, is actually determined by the list of encryption algorithms that designers choose to include in the described encryption algorithms table. So, for example, if for a system only AES and 3DES are acceptable, the table can be constructed using only these two algorithms. This also defines what is the range of strongest and weakest encryption algorithms that the system may be using at any moment. Moreover, while we only focused on the algorithm itself, and did not discuss key length or block length explicitly, these factors (even the number of rounds), where applicable, can easily be taken into account using the table. For example, instead of just having AES, we can put AES256 and AES128 as items in the table, to bring into picture the role of key length, and the system will choose each when decided appropriate.

Providing adaptations on encryption algorithms, also automatically leads to some sort of security through obscurity (note the famous quote of "security through obscurity is not security") [12,13]. One interesting topic that we leave as a future direction of this work to be investigated, is that whether adaptive mechanisms, as the one described here, can lead to weaknesses in the system and facilitate the job of attackers. For instance, issues such as this can be analyzed more throughly that if attackers get to know the details of adaptation mechanisms, they might force the system into adopting the lowest ranked (weakest) encryption algorithm by creating CPU loads, and making it easier for themselves to break into the system.

5 Related Work

Designing security features for embedded systems has its unique challenges and requires specific engineering methods and considerations. These issues have been the subject of many studies such as [14,1,2]. [14] and [1], focus on these unique challenges of security in embedded systems in general, and discuss them under the *processing gap*, *battery gap*, *flexibility*, *tamper resistance*, *assurance gap*, and *cost* titles. [14] also provides workload analysis of SSL protocol, and examples for energy consumption of different ciphers, to discuss and illustrate the impacts of security features in embedded systems. The vision for security engineering of embedded systems in the scope of a project is described in [2].

[15] and [10] are examples of works that study the impacts of security mechanisms on specific aspects of a system. Measurement and comparison of memory usage and energy consumption of different encryption algorithms on two different sensor nodes (MicaZ and TelosB motes) are performed and discussed in [15], and [10] offers performance and timing comparisons of encryption algorithms on two Pentium machines. The approach we proposed in this paper, relies on the results from the performance and timing comparisons of encryption algorithms as provided in the aforementioned study.

In this paper, an adaptive way to deal with the timing costs and requirements on security mechanisms was introduced. It should, however, be mentioned that there are other ways for taking into account these timing costs, which might suit very well other types of systems than discussed here. In systems with less complexity which are analyzable, a static and non-adaptive structure can be designed (i.e., a fixed set of security features will be used all the time e.g., to encrypt data). The idea we proposed in [4] is basically an example of such approach and systems.

The use of adaptive approaches and feedback mechanism for better CPU utilization and task scheduling in dynamic systems, where execution times of tasks can change a lot at runtime, has also been the topic of many studies in the real-time systems domain, such as [16]. One of the interesting works in the area of security for real-time embedded systems which uses an adaptive method is the study done in [17]. One difference between our work and [17] is that, there, the focus is on a set of periodic tasks with known real-time parameters,

whereas, our main target is complex systems consisting of periodic, sporadic and aperiodic tasks. Therefore, the analysis and formulas they offered in that work may not be applicable or need to be extended to support the type of systems we discussed here. Also, they consider the security level of the system as a QoS value explicitly, while in this paper, it is considered implicitly and left to the user through the use of a sorted table for encryption algorithms. Moreover, the main adaptation component of the system in that work is key length, while in our work it is the encryption algorithms that are adaptively replaced, and can easily include key length or any other relevant parameters as well. One of the interesting and close studies to our work is [18]. They basically use a similar type of adaptation mechanism as ours. However, the main focus in this work is on client-server scenarios using a database, and to manage performance of transactions. Also, security level adjustment in this work is done periodically using a security manager component, while in our method, adaptation mechanism executes per request and is not active when application has no request for encryption. Moreover, in that work, while security level switch is occurring, it can lead to use of an inappropriate encryption method by a client, which is solved by rejecting it, through passing several acknowledgment messages and repeating the process. Therefore, synchronization and message loss due of out of order arrival of messages are problematic for the security manager, which is handled by re-sending of data and applying other means.

As another approach for managing security in real-time systems, Tao Xie and Xiao Qin, has basically incorporated timing management of security mechanisms as part of the scheduling policy and developed a security-aware scheduler in [19].

6 Conclusion and Future Work

In this paper, we discussed security, as a non-functional requirement, in the design of real-time embedded systems, and particularly, how the choice of encryption algorithms, can affect timing requirements in these systems. An adaptive approach for selection of encryption algorithms was suggested for systems which need to balance their services at runtime in order to achieve their time constraints. It was shown how the approach can help the system to react to different processing loads and perform its encryption procedures within the defined time constraints. While, OSE RTOS was used as the base platform for implementation of the approach, there is nothing that stops it from being implemented on other platforms.

In the suggested approach here, a gradual increase or decrease of the rank of encryption algorithms was used. In other words, in each adaptation step, the system chooses either the next higher or lower ranked algorithm. As a future work, it can be evaluated how the approach would perform, if in each adaptation, the lowest or highest ranked algorithm was selected instead. For example, if it is observed that the system completes its encryption job earlier than its time constraint, it jumps to the top of the rank table and chooses the highest ranked algorithm for the next invocation. It would be interesting to study for which

systems/situations, each of these methods work better and categorize systems accordingly. Calculating the overhead of adaptation mechanisms and optimizing them is also left as a future study.

Moreover, in this work we focused on complex systems with not much information about timing properties of each individual task in the system to perform analysis. This situation was observed in the design of a telecommunication subsystem during our work in the CHESS project [20]. Accordingly, the approach that is suggested in this paper tries to improve satisfaction of timing constraints of the system by keeping a history of the timing behavior of the system. There is room to improve the suggested adaptation mechanism by taking into account more information about the system than was used here, and also more knowledge about the task that requires encryption when available.

Another direction of this work is to introduce other factors besides time for performing adaptations. These factors may include energy consumption, memory usage, and even situations where a system is under attack. Moreover, it was discussed whether and how adaptation mechanisms might actually help attackers to break more easily into a system. This issue can serve as an interesting topic for more careful investigations.

References

1. Kocher, P., Lee, R., McGraw, G., Raghunathan, A.: Security as a new dimension in embedded system design. In: Proceedings of the 41st annual Design Automation Conference. DAC '04 (2004) 753–760 Moderator-Ravi, Srivaths.
2. Gürgens, S., Rudolph, C., Maña, A., Nadjm-Tehrani, S.: Security engineering for embedded systems: the secfutur vision. In: Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems. S&D4RCES '10, New York, NY, USA, ACM (2010) 7:1–7:6
3. Cysneiros, L.M., do Prado Leite, J.C.S.: Non-functional requirements: From elicitation to conceptual models. In: IEEE Transactions on Software Engineering. Volume 30. (2004) 328–350
4. Saadatmand, M., Cicchetti, A., Sjödin, M.: On generating security implementations from models of embedded systems. In: The Sixth International Conference on Software Engineering Advances (ICSEA 2011). (2011)
5. Enea: The architectural advantages of enea ose in telecom applications. `http://www.enea.com/Templates/Landing.aspx?id=27011` (Last Accessed: September 2011)
6. Wall, A., Andersson, J., Neander, J., Norström, C., Lembke, M.: Introducing temporal analyzability late in the lifecycle of complex real-time systems. In: Proceedings of RTCSA 03. (2003)
7. Chodrow, S., Jahanian, F., Donner, M.: Run-time monitoring of real-time systems. In: Real-Time Systems Symposium, 1991. Proceedings., Twelfth. (1991) 74 –83
8. Saadatmand, M., Cicchetti, A., Sjödin, M.: Uml-based modeling of non-functional requirements in telecommunication systems. In: The Sixth International Conference on Software Engineering Advances (ICSEA 2011). (2011)
9. Enea. `http://www.enea.com` (Last Accessed: September 2011)

10. Nadeem, A., Javed, M.: A performance comparison of data encryption algorithms. In: First International Conference on Information and Communication Technologies, ICICT 2005. (2005) 84 – 89
11. CPU Killer. http://www.cpukiller.com/ (Last Accessed: September 2011)
12. Mercuri, R.T., Neumann, P.G.: Security by obscurity. In: Commun. ACM. Volume 46., New York, NY, USA, ACM (2003)
13. Hissam S, Weinstock C., Plakosh D., Jayatirtha A: Perspectives on open source. Software Engineering Institute, Carnegie Mellong http://www.sei.cmu.edu/library/abstracts/reports/01tr019.cfm (Published: November 2001, Last Accessed: September 2011)
14. Ravi, S., Raghunathan, A., Kocher, P., Hattangady, S.: Security in embedded systems: Design challenges. ACM Transactions on Embedded Computing Systems (TECS) **3** (2004) 461–491
15. Lee, J., Kapitanova, K., Son, S.H.: The price of security in wireless sensor networks. In: Journal of Computer Networks. Volume 54., New York, NY, USA, Elsevier North-Holland, Inc. (2010) 2967–2978
16. Khalilzad, N.M., Nolte, T., Behnam, M., Åsberg, M.: Towards adaptive hierarchical scheduling of real-time systems. In: 16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'11). (2011)
17. Kang, K.D., Son, S.H.: Towards security and qos optimization in real-time embedded systems. In: SIGBED Rev. Volume 3., New York, NY, USA, ACM (2006) 29–34
18. Son, S.H., Zimmerman, R., Hansson, J.: An adaptable security manager for real-time transactions. In: Euromicro Conference on Real-Time Systems. (2000) 63–70
19. Xie, T., Qin, X.: Scheduling security-critical real-time applications on clusters. In: IEEE Transactions on Computers. Volume 55. (2006) 864 – 879
20. CHESS Project: Composition with Guarantees for High-integrity Embedded Software Components Assembly. http://chess-project.ning.com/ (Last Accessed: August 2011)