Abstract—Wireless sensor networks are composed of nodes with stringent constraints on resources. Some of these devices may have the possibility to recharge batteries (e.g., by means of solar panels); though, a reduced power consumption is anyway a key factor when recharge resources are not available (e.g., during the night for solar panels). In this paper we describe a method for security self-adaptation tailored for wireless sensor networks. This method allows devices to adapt security of applications gradually with the goal of guaranteeing the maximum possible level of security while satisfying power constraints. A case study, implemented on Sun SPOTs, is also presented to show how the method works in a real wireless sensor network.

I. INTRODUCTION AND MOTIVATIONS

Wireless Sensor Networks (WSN) are composed of a large number of nodes that are usually able to perform some measures through suitable sensors, process the gathered data, and send information to other nodes of the network. A number of more powerful devices are also be included in the network; these devices are used to collect data produced by the nodes. Nodes are placed in the environment that they have to sense and, therefore, they are resource-constrained being limited in term of area, memory, computation capabilities, and power. Power consumption is always among the most important constraints for WSN nodes [1]; power sources can, in some cases, be recharged by means of local power generators (e.g., small solar panels). Though, the batteries might require to last, even for long times, without being recharged.

An important factor that influences design and performance of WSNs is communication security. In particular, security services such as authentication, confidentiality, and availability are critical for sensor networks operating in hostile environments and, at the same time, handling sensible data. Designing a secure system in these conditions is challenging [2]: traditional security solutions are designed by using ad-hoc approaches which offer specific protection against certain attacks (e.g., countermeasures against denial of service attacks). However, they rely on the assumption that the operative environment is well-known and quite static as well. Moreover, some of these technologies have not been specifically developed for embedded systems; in many cases, their adoption in the pervasive world would be impossible due, for example, to high hardware resources requirements [3].

Typically, when designing secure systems the worst case scenario is considered: the system has to guarantee adequate protection against the strongest possible security attacks. By following this philosophy, security in WSN typically follows an “on-off” approach: either security is totally ignored or it is enabled with the strongest algorithms available, with a corresponding high consumption of resources. This is generally in contrast with the requirements of a resource-constrained devices such as mobility, flexibility, real-time configuration, open and dynamic operative environment [4], [5].

The problem of optimizing resources used for security, yet providing an adequate level of protection, is an hot topic at the moment [2]. In particular, the trade-off between energy and performance requirements of security algorithms is of utmost relevance for embedded systems [6]. As discussed above, each adopted security solution should be a good compromise between factors that are conflicting in nature such as, for example, power consumption and performances. This optimization is a complex task, especially when performed run-time [7], [8]. In this paper we concentrate on systems that are able to change their security configurations at runtime. In particular, we propose a run-time mechanism to deal with the optimization of security in accordance both with application security requirements and with system dynamic energy constraints. Our work is based on the “Adequate Protection Principle” [9] which states that security should be adequately applied to a given context. We implemented such principle by adopting a novel approach that relies on gradual adaptation of application security and of system workload: security is adapted by moving between adjacent configurations that are compatible with application requirements. System workload can be reduced by killing certain tasks. This is taken as an extreme measure that might be adopted to satisfy energy requirements.

In WSN composed of nodes that can be recharged (e.g., by using local generators such as solar cells), our solution increases WSN nodes lifetime in between recharges. Different strategies are used to favor either security or system workload. The strategy to be used and the way in which it will be applied depends on specific node energy conditions and applicative scenario. The solution proposed in this paper has been implemented on Sun SPOT nodes [10].

In the next Section, we introduce the main terms, concepts and parameters that we considered in the design of our solution; in Section II are discussed some related works. In Section IV, we present the results obtained form a case study based on a real implementation on sensor node. We discuss some security issues related to our approach in Section V.
II. RELATED WORK

The principle of gradual adaptation described in our project can recall the graceful degradation techniques used in fault tolerance; in fault tolerant systems performance may be degraded to keep the system operational even in presence of faults [11], [12]. Typically this approach does not address the optimization of security.

The challenge of selecting the best set of cryptographic algorithms that optimizes the trade-off between resource requirements and security provided has been tackled in many works. Techniques to minimize the energy consumed by secure wireless sessions have been proposed in [13]. The authors investigated the selection of encryption algorithms and of key exchange protocols. However, they did not provide explicitly a run-time self-adaptation mechanism; instead, they shown techniques to minimize energy consumption by matching block sizes of message compression algorithms with data cache sizes. In [6] a battery power optimizer for wireless networks has been presented. Authors have performed experiments to model the relationship between power consumption and security of cryptographic algorithms. Such information have been used to formulate a knapsack problem and find the optimal level of vulnerability by changing the number of rounds in cryptographic algorithms. In contrast with our work, neither application requirements nor the possibility of managing changes in cryptographic algorithms were considered. In [7] the authors describe a self-adaptive security framework at protocol level. The mechanism proposed provides the ability to select the optimal set of security protocols with the best security/performance ratio depending on the malicious level of a node neighbors. Adaptation of cryptographic primitives is instead presented in [14], where the basic idea is to change the AES cryptography key length according to the confidentiality level required by the user.

Very few works propose and/or consider the concept of gradual adaptation for optimizing the security provided. Most of the aforementioned solutions do not include security requirements of the applications running on a system. Moreover, in contrast with these works, our framework is centered on the idea to provide the desired level of security by following a “best effort” approach: it assigns the “strongest” security algorithm and gradually degrades performances only if such configuration is not affordable for the system.

III. SECURITY ADAPTATION

In this section we describe our adaptation mechanism. As mentioned previously, security provided to different tasks is changed dynamically depending on remaining energy in the battery and on the workload of each node.

Our target system is a sensor node that may be used in multimedia applications and, therefore, is both required to take measures by means of different sensors and to communicate with the base stations frequently. Each one of these operations might be seen as a different task which has different security requirements. The node may provide the ability to use different security suites to accomplish such requirements. Supposing that any security suite can be arbitrarily assigned to applications, the possible states of our system are given by all the combinations application-security suite. For example, a node may have 3 security suites for packet encryption and may host 5 running tasks; in total it supports $3^5 = 243$ possible configurations. However, as previously mentioned, the association among tasks and security suites is subject to the constraints given by the security requirements of tasks. In general, these requirements reduce the number of alternative configurations that may be used in the system. Thus, considering the above example, if one application supports only 2 of the available security suites, the number of possible configurations are reduced to $2 \cdot 3^4 = 162$. Security requirements are detailed in Section III-B.

In our approach the security configurations can be changed runtime - by following the principle of gradual adaptation to fulfill application goals without violating system constraints. To do this in an effective way there is the need of a self-adaptation mechanism. In the system that we have considered, there is no node inter-communication; all the nodes only communicate with the base station. This greatly simplifies the management of security adaptation. If multiple nodes were involved in communications, a negotiation among multiple nodes should have taken place as described in [15].

A. Gradual Adaptation

Gradual adaptation is a process by which different contiguous security configurations are adopted at each step. All the different security options are ordered by following a proper criterion and adaptations are done by moving among different adjacent configurations, one step at a time until system and application requirements are satisfied. We have a configuration downgrade (or upgrade) with respect the considered metric, if we move the system from the current active configuration to the previous (or next) one, as shown in Figure 1. Such changes are performed gradually, that is, step by step and without jumps.

All adaptive systems are subject to constraints which may limit the goal achievements or, in some cases, even forbid their realization. In our system we considered energy consumption as the main constraint. In fact, energy consumption represents the utmost requirement for a sensor node. Our security self-adaptation process is driven by the following goals:

- Maximize the system workload: maximize the number of running tasks.
• **Maximize application security:** maximize the level of security associated to each task, according to its security requirements.

The system is subject to an **energy constraint:** the energy consumed by the active configuration should be within a certain range, delimited by an upper and a lower threshold.

An adaptation is triggered when the energy constraint is not satisfied and it is performed by changing both the security provided and the number of running tasks, in order to achieve the above goals.

In particular, security adaptation is performed by gradually decreasing or increasing the security provided to a given application based on its security requirements. In this case we chose to order security configurations depending on the provided security (measured as the effort required to break the cryptographic algorithms included in the considered configuration).

Initially, our framework applies a “best effort” approach to satisfy the applications requirements: it tries to guarantee to all the tasks the highest possible level of security compatible with their security requirements. This corresponds to maximizing security. If such a system configuration is not compatible with current energy constraints, the system selects a new configuration by degrading security of single applications, thus moving to a less energy hungry configuration. Further degradations are considered at the following steps if the energy constraints are not met. A security upgrade may also happen when the consumed energy is below the lower bound and a previous degradation has been performed.

Concerning workload, the adaptation is performed by gradually modifying the number of running tasks. Even in this case, a “best effort” strategy is followed: by default all tasks are allowed to be executed; the goal is, in any case, of maximizing their number. However, if the energy constraints cannot neither be satisfied with a security degradation, number of running tasks must be reduced by “killing” or suspending some of them and by denying new tasks to start. Tasks can be killed or suspended according to different strategies as explained in Section III-C. Workload configuration upgrades are possible if the system keeps track of suspended tasks in order to resume them. Upgrades may also include the re-enabling of the possibility to start new applications.

The aforementioned processes for performing adaptation require to have a selection mechanism for finding new configurations that fit within the energy constraint by considering both the security provided and the system workload. We propose to use specific policies to determine the new configuration to be enforced. A policy specifies the actions that must be enforced in order to accomplish the adaptation goals. Policies determine the adaptation space and selection mechanisms that the system can use to change its configuration. Basically, a policy contains information regarding: when it is valid and enforceable; when it should be activated; what are the adaptation parameters that it can modify. Policies are defined at design-time. They represent a flexible mechanism to tune the adaptation process according to a specific applicative scenario. Details regarding policies are presented in Section III-C.

### B. Application Security Requirements

In our framework, management of application security is delegated to the system. Each application has security requirements associated to it. These requirements are used by the gradual adaption process to provide adequate protection to the applications. The application requirements can be expressed as a tuple:

\[
\mathcal{R} = \langle S_{level}, D_{steps}; D_{policy}; \rho \rangle
\]

With \( S_{level} \) we identify the security level required by an application: high, medium, low and none; we assume that sets of cryptographic algorithms can be associated to different security levels. The security level is defined by considering the strength of each algorithm (i.e., its resistance to known attacks, measured as the number of combinations that is necessary to try to discover the cryptographic key). Additionally, algorithms are evaluated and ordered based on their energy consumptions. Energy consumption figures can be obtained by effective measurements of algorithm performances. For example, in Table I are reported the security algorithms considered in our case study grouped by their security level and ordered by the performances that they provide.

The pair \( D_{steps}; D_{policy} \) is the security degradation policy composed of degradation steps and degradation policy, respectively. The security degradation policy specifies the strategies to downgrade the security performances according with the principle of gradual adaptation. It is composed of two terms: \( D_{steps} \), the maximum number of security degradation steps that an application can accept; and, \( D_{policy} \), the policy which determines how degradation of security has to be performed. A degradation policy \( D_{policy} \) can be described as follow:

- **Non-degradable:** degradation is not allowed. The security provided have to correspond to the highest-ranking algorithm within the security level claim by an application.
- **Intra-level degradation:** degradations inside the same level are allowed, if such a level supports different alternative cryptographic algorithms. For example, an application that requires a medium security level has to maintain such a level during its lifetime but the algorithm the that provided such a medium protection may be changed.
- **Inter-level degradation:** it includes intra-level degradation and additionally the applications accepts to be downgrade to the next lower security level (e.g. from high to medium but not to low).
- **Fully-degradable:** the application accepts to be degraded as much as possible.

As stated before, upgrades of performances may occur only as a consequence of a degradation and are always allowed by an application provided that they do not exceed the security level requirements.

Priority level \( \rho \) is used to manage the existence of some critical applications; those applications are either necessary for the system to run, or fundamental for the system role. Non-critical applications can be denied to run if their requirements cannot be met; critical applications cannot be denied running and the system must perform all the operations necessary for feeding the resources required to satisfy their requirements.
This may include suspending or terminating non-critical applications.

Table III shows an example of security requirement specifications used in our case study.

C. Adaptation Policies

As mentioned previously, system configuration changes happen by following predefined policies: a policy \( \varphi \) is defined as a tuple containing the following items:

\[
\varphi = \{ \langle E_{ub}; E_{lb}\rangle, \langle T_{ub}; T_{lb}\rangle, \langle AD_{steps}; AD_{policy}\rangle, \langle K_{apps}; K_{policy}\rangle, \langle O_{policy}\rangle \}
\]

The first two parameters are related to the energy context: \( \langle E_{ub}; E_{lb}\rangle \) are the upper and the lower bound of the available energy in which the policy \( \varphi \) is valid and enforceable.

The second pair express the trigger conditions: \( \langle T_{ub}; T_{lb}\rangle \), are energy thresholds that trigger the adaptation process by enabling the corresponding policy. The purpose of these thresholds is to enable gradual degradation of performances when the energy consumption is greater than \( T_{ub} \) or to enable gradual upgrade when the energy consumption is lower then \( T_{lb} \).

The pair \( \langle AD_{steps}; AD_{policy}\rangle \) refers to the applications degradation policy. \( AD_{steps} \) is the default maximum number of degradation steps allowed for the applications. If an application specifies its own \( D_{steps} \), the minimum value between \( AD_{steps} \) and \( D_{steps} \) is considered. \( AD_{policy} \) indicates the rules used to select the applications to degrade. For this purpose, several strategies can be used, such as: indistinct degradation of all applications, the oldest application, the least recently started application, the most “resource greedy” application, the application with lowest priorities, randomly. In our implementation, we considered both a priority-based approach and indistinct degradation of all applications.

\( \langle K_{apps}; K_{policy}\rangle \) concerns the number (or percentage) of threads that can be terminated \( K_{apps} \) and the termination policy \( K_{policy} \). The threads can be “killed” or suspended according to different strategies. Suspended applications are resumed when enough system resources are freed. Target threads can be selected according to different strategies such as, for example, application priority or resource demand.

It is important to specify in which order \( AD_{policy} \) and \( K_{policy} \) are enforced when an adaptation is required. This is done by choosing a proper optimization policy \( O_{policy} \) which can be as follows:

1) Security degradation first: the \( AD_{policy} \) is first applied; the \( K_{policy} \) is applied as a second option. Adaptation is performed by degrading security of all applications \( AD_{policy} \): if energy constraints are not met after these adaptations, applications are killed \( K_{policy} \).

2) Killing applications first: to meet the energy constraints, the system starts to kill applications by enforcing \( K_{policy} \): if this does not allow energy consumption to go below the threshold, the system will apply security degradation \( AD_{policy} \).

3) Hybrid approach. The system applies the strategies of point (1) and (2) in turn: the system kill or suspend applications at one adaptation step and it degrades security at the following; these steps are repeated until the energy constraints are satisfied.

\( AD_{policy} \), \( K_{policy} \) and \( O_{policy} \) should be defined according with system functionalities and the applicative scenario in which the WSN is used.

The syntax described above provides the ability to define several policies that are enabled in different instants of time, depending on the battery level (i.e., the energy context). In this way different strategies can be enforced depending on the system conditions, thus customizing the adaptation behavior.

D. Self-adaptation Architecture

From the architectural point of view, our self-adaptive framework is based on the Monitor, Controller, Adapter (MCA) design paradigm as shown in Figure 2.

1) Monitor: the Monitor component monitors predefined system parameters and triggers proper system events in predefined situations. The trigger events can be both hardware and software and can be related to internal or external conditions of the system. There are many kinds of possible triggers and they should be decided according to the purpose and requirements of the system. Events may be detected as soon as they happen or periodically. For example, it is possible to have an “event listener” that check if the conditions of the system fit within the self-adaptation requirements or it is possible to have a periodic task that monitors the elements that can raise triggers. The periodic monitoring can be intended as a periodic task in term of timing (e.g., every 50 seconds) or in term of repeated events (e.g. a certain quantity of byte sent/received or the number of sample collected from the sensors). The choice between these two types of trigger depends on the system requirements and operating scenario.

2) Controller: the Controller is the core components of the adaptation framework. It receives the notifications from the Monitor and it computes adaptation strategies with the aim of satisfying system and application goals.

3) Adapter: the Adapter module is responsible to enforce the adaptation decisions taken by the Controller by changing the security settings of the applications or by killing/suspending proper applications.

IV. Case Study

In this section, we present a case study in which we implemented our security degradation method on the nodes of a sensor network. This network is based on the Sun SPOT technology (Sun Small Programmable Object Technology) [10]. Sun SPOTS are small wireless devices, compliant with the IEEE 802.15.4 standard and running the Squawk Java Virtual Machine (VM) without any underlying OS. The VM acts as both operating system and software layer to provide high portability and fast prototyping capabilities. Sun SPOT is designed to be a flexible development platform, capable of hosting widely differing application modules. From the hardware point of view, Sun SPOT nodes are equipped with a
32 bit ARM920T working at 180 MHz; 512K RAM/4M Flash Memory; 2.4GHz IEEE 815.4 radio with integrated antenna; 720 mAh as maximum battery capacity. By default, these devices are equipped with temperature and light sensors.

The goal of our experiments was to measure the effects of our security degradation algorithm on battery management. Power consumption and the related node lifetime is essential in the periods of times among battery recharges (supposing that nodes are equipped with local power generators).

In this implementation we have used threads for simplicity reasons. Aim of a thread is to collect data from the built-in sensors, encrypt them, prepare a packet and send it to the sink node. Two kinds of monitoring thread have been considered which exploits the sensing capability of SPOTs: the first one takes measures from the light sensor; the second one monitors the temperature. Each thread might have its own execution period (threads are periodic) and number of samples to collect (i.e., maximum number of iterations).

A special thread periodically generates new monitoring threads with random characteristics: thread type, thread period, number of samples, security requirements (as described in Section III-B).

Concerning security suites, Sun SPOT supports the TLS/SSL protocol with several cryptographic algorithms. We extended the current implementation by also enabling the AES encryption algorithm. Some measures have been performed in order to determine the consumptions and the execution times associated with each algorithm. By using this information we could order the algorithms according to their security and to their energy consumption. Such a ranking is used by the self-adaptation process to perform a gradual adaptation if the consumption is above the trigger threshold. The encryption algorithms considered in this case study are shown in Table I; the algorithms have been grouped by using the ordering discussed above. As security strength criterion, we considered the number of combinations needed to guess the cryptographic key: higher the number of combinations, stronger is the algorithm.

As described in Section III-C, several policies can be defined according to the specific scenario and desired adaptation behavior. Table IV shows the policy that we have used in our SPOT implementation. This policy is saved in the flash memory of the Sun SPOTs.

Both energy contexts and trigger thresholds have been defined as percentages of the total energy capacity. Similarly, the number of killable applications is expressed as a percentage of the number of running threads. We considered a fixed value of 0 as the application degradation policy, $D_{policy}$; this corresponds to indistinct degradation of all running applications. Termination policy $K_{policy}$ has been also fixed to 0, by meaning that the victim threads are the ones with lower priority. The optimization policy used, $S_{policy} = 0$, corresponds to a degradation of security, followed by termination of threads when the energy constraints are not satisfied.

Figure 3 shows the flowchart of the adaptation process implemented in the SPOT. This represents the logic used within the Controller component; the box in Figure 3 is the Adapter component.

A. Self-adaptation process overhead

Introducing self-adaptation in a system increases its flexibility and its reactiveness to changes in operative environment.

![Fig. 2. Self-adaptive schema](image-url)
Though, to support these capabilities, an additional quantity of energy and of execution time are required. In order to understand the feasibility of our self-adaptation framework, such overheads have been estimated. Figures 4 and 5 show the measures obtained for different number of running applications. Self-adaptation related overheads grows with the number of applications involved; though, the amount of this growth depends on types of adaptation involved and on the number of degradation steps considered. In this case study applications of the same type have been used and the measurements have been done by forcing all the applications to degrade their performances during the self-adaptation process. In fact, in sensor nodes, usually, there is a limited number of security algorithms that can be used, due to their constrained resources. Thus, the growth on the number of possible system configurations mainly depends on the number of running applications. Similar overhead considerations are valid in case of upgrade of performances. Though, as shown by the experimental results, despite this additional costs, we can observe an improvement of system performances by using adaptation.

B. Experimental results

In order to evaluate the effectiveness of our approach we compared the behavior of a system with no self-adaptivity with an equivalent system in which security adaptation is used. In particular, we analyzed the effects of security adaptation on system workload, energy consumption, and execution time. Moreover, we performed a detailed analysis of system adaptation behavior. The results reported in this section refer to three different sets of experiments that are related to different working conditions for the system. The three sets are listed below:

1) Fixed energy constraint.
2) Fixed system workload.
3) Complete scenario.

The first two sets are used for performing comparisons with a system that does not support self-adaptivity; the third one has been used to show and evaluate how gradual adaptations are performed. A description of the three experiments and of the results obtained follows.

1) Fixed energy constraint: Aim of this experiment is to show the behavior of our self-adaptive framework when an energy constraint is set and to compare it with the behavior of a system with no self-adaptation. We set a threshold of 0.19% of the total battery charge (i.e., 720mAh) both for the self-adaptive system and for the normal one. The experiment is done by introducing in both systems an increasing number of periodic threads. All of them have a period of 4s and a new thread is introduced every 25s. The monitoring period is 60s. Both the experiments with the non self-adaptive system and the ones with the self-adaptive system are terminated when the power consumption reaches the aforementioned threshold in a monitoring period (i.e., when the maximum allowed power consumption is reached).

For the case with self-adaptivity, we adopted the policy $P_{\text{a}}$ of Table IV. It is valid on the entire battery level range and it provides the highest number of degradation steps (i.e., 4) for all threads. The upper bound trigger threshold has been set to $T_{\text{up}} = 0.18\%$. Instead, no lower bound trigger threshold has been set, since it is not relevant for the scope of this...
experiment; percentage of killable applications is set to 0, to avoid threads to be killed.

In order to avoid fluctuations in results, each experiment has been repeated five times and the mean of the obtained results have been considered.

Figure 6 reports the comparison results, where the system without self-adaptivity is considered as a reference. Our self-adaptive framework allows the system to execute threads for a longer time (roughly 31% more) before reaching the threshold on power consumption, by providing the ability to execute, at the same time, more threads.

Our framework brings advantages by extending system lifetime, despite the added overhead. Moreover, an increment in performances is obtained despite the lower energy consumption per time, as shown in Table II.

2) Fixed system workload: In this set of experiments a system with security degradation and another without it are compared over the same period of time while they are executing a fixed number of threads. The number of threads used in each experiment is 6; every one of them will be periodically executed for a total of 40 times with a period of 4s. All threads have different security requirements: two threads have high security requirements, other two of them have medium security requirements, and the last two of them have low security requirements, as shown in Table III.

Degradation steps have been limited to one step for high and medium security threads; the low security threads, instead, cannot decrease their security any further. Threads all initialized at the beginning of the execution, and stopped when all of them performed 40 iterations. Also in this case we collected results of five different runs, and reported the average values.

Concerning the self-adaptive system, also in this experiment, we considered an unique policy valid in the entire range of battery levels, $\varphi_b$, of Table IV. The upper bound for triggering the adaptation has been fixed to the average value of the energy consumption obtained from the case without self-adaptation; no lower bound trigger threshold has been set; any of the threads can be killed (i.e., $K_a$ and $K_{policy}$ equal to zero).

Figure 6 shows the percentage variations with respect the case without self-adaptivity. The increment in the execution time is due to the delay introduced by the adaptation process for selecting and enforcing the new optimal system configuration. On the other side, the adaptation of security provides the ability to save energy by degrading the security level according to thread security requirements. Due to fixed workload, in Figure 6 we do not have changes in the number of running threads and operations performed.

During the tests with self-adaptation only one degradation of performance is registered due to the degradation policies of each thread; however, the ratio between the energy spent and the total execution time is lower than the case without adaptation, as shown in the second row of Table II.

3) Complete scenario: in this experiment we demonstrate a complete scenario in which gradual adaptation is performed. We show how different policies can be applied in order to optimize the trade-off between system workload and security while respecting different constraints on energy consumption.

For these experiments, three policies have been defined as reported in Table IV (run 3a). The number of degradation steps is set to the maximum value (4) for all threads, unless a...
In Figures 7, 8, 9 we can observe three types of adaptation that occur in different instant of time (i.e., iterations) according to the corresponding energy discharge, are at iteration 78 (from system energy context. Changes in active policy, because of 10 seconds. The other thread properties and requirements are execution is composed of 300 monitoring intervals. The monitoring period has been set to 90 seconds and the global execution is composed of 300 monitoring intervals.

Thread are created every 120 sec with a sample period of 10 seconds. The other thread properties and requirements are randomly selected: light or temperature type of monitoring thread; a random number of iterations from 50 to 550; random security requirements from high, medium and low, with random degradation policy and degradation step.

The results obtained are shown in Figures 7, 8, 9, where the three polices (℘\text{1}, ℘\text{2} and ℘\text{3}, respectively) are enabled in different instant of time (i.e., iterations) according to the system energy context. Changes in active policy, because of corresponding energy discharge, are at iteration 78 (from ℘\text{1} to ℘\text{2}, Figure 8) and iteration 206 (from ℘\text{2} to ℘\text{3}, Figure 9).

In Figures 7, 8, 9 we can observe three types of adaptation behavior, depending on the current energy constraint of the active policy: a) no adaptation required; b) adaptation not possible; c) adaptation performed. In the case b), adaptation is not allowed either because the performances of the system are already at maximum level or because all the available degradations (both security and thread termination) have already been applied.

Initially, when ℘\text{1} is enforced (Figure 7), the energy consumption is below the policy lower bound threshold. Thus, according to the best effort approach, adaptation is not performed because all threads requirements are already satisfied with the higher security level they required (the above case b)). Basically, until the energy consumption is within the policy thresholds, no adaptation is required. Instead, in Figure 8, the system gradually tries to degrade/upgrade the performances to fit within ℘\text{2} thresholds.

In Figure 9, when policy ℘\text{3} is active, the system tries to reconfigure itself several times according to the policy specifications; however, it is not able to stay within the policy thresholds. This is due to the high system workload
conditions, thus providing to applications, at any given time, upgraded or downgraded (depending on system energy requirements) algorithms. The strongest security algorithm available; downgrades (and upgrades) are performed depending on system energy conditions, thus providing to applications, at any given time, the highest level of security that is compatible with system conditions. Furthermore, by using adaptation policies it is possible to tune the adaptation mechanism and customize it according to specific application scenarios.

In this paper we not only proposed the gradual adaptation approach, but we also demonstrated its feasibility by implementing it in resource-constrained devices.

Future work includes the extension of our framework to support more adaptation parameters (such as processor and memory usage) as well as implementing smarter controllers that might be able to deal with concurrent policies.

VII. ACKNOWLEDGMENTS

This work was partially supported and funded by the Hasler Foundation under the Project AETHER+ (Grant No. 09083-2043-021). The paper reflects only the view of the authors; the Hasler Foundation is not liable for any use that may be made of the information contained herein.

REFERENCES