Highly Adaptive Cryptographic Suites for Autonomic WSNs

Dennis Bliefernicht Daniel Schreckling
Computer Science Department
University of Hamburg
Vogt-Koelln-Str. 30
D-22527 Hamburg, Germany
{1bliefer|schreckling}@informatik.uni-hamburg.de

ABSTRACT
This paper presents a novel approach to support flexible cryptographic suites on resource restricted devices. Common primitives which are used for symmetric key algorithms have been thoroughly analysed by the hardware and cryptography community to design more efficient implementations. A side product of this development and a common fact for cryptographers is the strong modularisation of block ciphers and the similarities therein. This contribution shows how an efficient combination framework can exploit these characteristics to support a large variety of cryptographic primitives on resource restricted hardware. We present a collection of basic building blocks, which can be combined dynamically to implement various primitives. With the high energy consumption of radio transmission in mind the presented work takes a different direction from existing approaches which aim at highly optimised primitives. By exploiting the modular redundancy within existing block ciphers the presented approach can adapt to a node context and choose the primitives optimal for a specific task in terms of security requirements, and resource consumption. This work additionally discusses how this modularity enables a device to efficiently update existing and install new primitives, and how it may adapt implementations to resource restricted execution environments.

Keywords
Cryptography, Sensor Networks, Adaptive Security

1. INTRODUCTION
The research area of wireless sensor networks enjoys great popularity. Technology is increasingly supporting this trend and devices are moving from the laboratory to real-life applications. Of course, this implies that in a couple of years from today we will not only be surrounded by the mobile and "bulky" devices we know today, such as Laptops, PDAs, cell phones, etc., but also by smart, small, and cheap devices. They will be able to accomplish very simple tasks providing input to the more powerful devices. With this information the latter devices will be able to sense the environment and support our daily life according to the context associated with our location.

Thus, depending on the capabilities of the nodes in this scenario, we will have different node perspectives. As an example, you may think of a simple sensor device which picks up the heart beat of a patient. This device does not apply complicated computations to the data. It does not process context information or communicate to other devices to derive information abouts its context. Instead it simply operates within a local context and reports the information to another more sophisticated device, e.g. a PDA carried by a nurse. The PDA in turn can pick up several sensor readings. According to the accumulated data and some patient profile specified in advance it is able to raise an alarm if something abnormal happens. Based on the readings which constitute the context of the PDA, this device may also be able to distinguish false alarms from real alarms. Thus, in this very simple example, a more capable node is able to adapt to the surrounding nodes, to the environment, and to its own characteristics. Moreover, the more capable nodes are able to guide or instruct restricted devices which security mechanisms to deploy. Short: nodes adapt to the context and application scenario in which they act and try to exhibit autonomous behaviour.

If we specify this observation and reconceive the roles of this scenario in the realm of security we will discover an equivalent situation. There will be devices which possess more complete information about their context. Consequently, these nodes will also be able to take better decisions when determining what kind of security mechanisms should be used in order to secure communication between nodes. Parameters which influence this decision can be node characteristics (battery status, processor power, radio technology, etc.) and the characteristics of the security mechanisms available (message overhead, cycles needed to en- or decrypt a message, etc.) to name just two examples. Thus, ideally, to always take the best decision, i.e. to use the security mechanisms, most feasible and efficient for a specific task, every device would have to support a variety of security mechanism currently available. However, currently existing approaches mainly aim at investigating the application scenario in which a resource restricted device will be used. According to this analysis separate and specific security mechanisms are determined and deployed. Most of the time this decision is a trade-off between the security guarantees provided by this particular and highly optimised mechanism and its resource requirements.

This work takes an opposite approach and describes how a resource restricted device can be equipped with a cryptographic suite able to implement a number of different security primitives. It is not based on a scenario specific and optimised cryptographic primitive which constitutes a trade-off between security and efficiency. Instead, it focuses on a modular approach which allows for combinations of small and optimised components which can be combined to primitives optimal within a certain context.

To explain our approach Section 2 first links our approach to existing work. Afterwards, Section 3 outlines the general functionality of our solution and list the problems we can solve with this approach. Section 4 will then explain our solution in respect to block ciphers. We identify the basic building blocks of block ciphers exploited in this work, present the description of these blocks and of their combination, and discuss some implementation details. Before concluding our work in Section 6, we discuss this highly dynamic approach in Section 5, regarding its efficiency, security, and its general capabilities.

2. BACKGROUND AND RELATED WORK

Moore's law [19] often causes people to question approaches which design methods which provide cryptographic support for resource-constrained devices. Their main argument: The advancement of technology will annul the restrictions and allow the installation of numerous security primitives in currently existing sensor platforms. However, the current development shows an inverse trend. Instead of small devices with increasing resources we observe the development of devices with constant or less resources which decrease in size, price, and to some extent also in power consumption [10, 11]. Moreover, if we look at the evolution of battery resources it becomes obvious that they currently do not keep pace with Moore's law and thus stay behind the current development [31]. As a result, the last couple of years the research in the realm of sensor networks has originated numerous security protocol suites which are feasible for highly resource restricted devices.

The most popular representatives are SPINS [22] and TinySec [16]. Both protocol suites are not dynamic in the sense that they are able to adapt the security mechanisms during deployment. Nevertheless, they give valuable insights in the design decisions to be taken when developing new security protocols for constrained devices. Furthermore, for higher flexibility they pursue a modular approach which enables the construction of more complex protocols from rather basic components. We think that this is an essential part for future adaptive mechanisms.

A more dynamic approach is proposed in [5]. Chigan et al. describe a framework which is capable of deploying combinations of security services to satisfy security needs in a specific context. The combination process in this scheme tries to find a trade-off between the level of security provided and the overall network performance of the wireless ad hoc and sensor network.

A similar approach is presented in [24]. For three different security levels this work proposes existing crypto-algorithms and security protocols based on analyses concerning their effectiveness and potential impact on low data rate devices. A security manager allows the network of heterogeneous devices to comply with the security requirements of selected services. This approach is rather limited in its adaptivity as it only offers the use of RC5 and AES for the different security levels. This work also showed how different parameters for the security protocols and cryptographic primitives can successfully safe resources of a sensor node and at the same time comply with different security levels.

Finally, the work presented in [23] aims at deploying a sensor node which offers strong asymmetric cryptography during the deployment phase. After a one-time bootstrapping procedure this crypto suite is dynamically replaced by resource friendly security protocols for the regular and secure operation of the node. The research conducted in this work does not only show how devices with very limited memory resources can benefit from public-key cryptography but it also emphasises the need for different security protocols for different application scenarios and contexts. However, the flexibility and adaptivity of this approach for real-life applications is low.

3. GENERAL FUNCTIONALITY

To understand how Highly Adaptive Cryptographic Suites (HACS) defined in this paper are going to overcome the limitations of the solutions described above we will outline the basic structure of a sensor node. This outline will focus on the logical components which enable a sensor to support HACS. Other components not relevant for our implementation are omitted. Clearly, this structure is not limited to wireless sensor nodes but can be transfered to any other network node.

Figure 1 depicts a sensor node able to run HACS. The generic communication interface allows the node to exchange arbitrary data with proximate nodes. A security policy installed on the sensor controls when, how, and to whom the sensor data is distributed. Of course, this security policy can be empty. If it is not, it can for example determine that data read by the sensor is only distributed to authenticated nodes over a secured channel. For this purpose appropriate credentials are used. The security policy is enforced by the cooperation between the policy enforcement and policy decision point.

The three software components relevant for enabling HACS are the *Security Primitive or Protocol Description* (PDC), the *Building Blocks* component (BBC), and the *Combination & Execution Unit* (CEU).

HACS accesses the building blocks component to construct the required cryptographic primitives or protocols, i.e. this component represents some memory space which contains a specific number of simple and optimised binary fragments dedicated to compute specific functions, e.g., circular left or right shifts, modulo multiplications, etc. The number

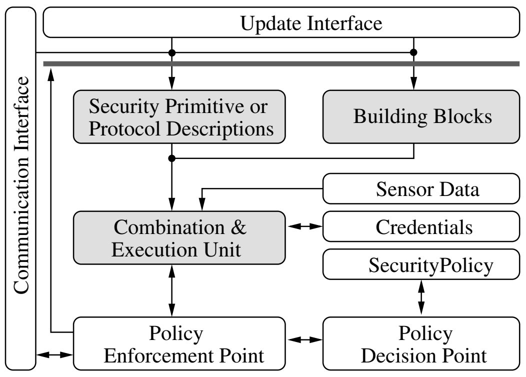


Figure 1: Logical components required on a sensor node for HACS support

and type of building blocks stored in this component are discussed in Section 4.1. The second component, Security Primitive or Protocol Description, allocates memory space for a number of high level descriptions which determine in which order the building blocks have to be executed in order to compute the output of a cryptographic primitive or to run a protocol instance. The two alternatives for these descriptions are explained in more detail in Section 4 (specifically in 4.3). Finally, the knowledge and the objects stored in the last two components have to be combined and run, i.e. the CEU reads the description from PDC derives the required building blocks to construct the corresponding security mechanism and uses the building blocks from BBC to generate and run it. As an example one may think of constructing a simple RC5 cipher and encrypt sensor data with this encryption function. This combination and execution process and its variants are also described in Section 4. The logical representation of the sensor node in Figure 1 also contains an update interface. Similarly to users which are able to connect to the sensor node via the communication interface it is possible to update building blocks and descriptions of security mechanisms using the update interface. Thus, controlled by the security policy it is possible to update either the elements of the PDC or of the BBC. In this way, it is fairly easy to install a new security mechanism which makes use of the installed building blocks. Also new mechanisms which may lack of some elements in BBC can be installed with less effort as the BBC has to undergo only slight modifications. In this way, security relevant updates on the installed mechanisms can be conducted efficiently too. Instead of providing large monolithic blocks which would represent the security mechanism itself only single small building blocks or compact descriptions are exchanged.

Apparently, it is obligatory to control this update process in order to avoid that unauthorised nodes or users are able to infiltrate the node with either building blocks or descriptions which are not providing the functionality they were designed to. Thus, the policy enforcement controls the access to both, BBC and PDC.

Currently, the decisions about which security mechanism should be employed are taken locally by the policy decision point (PDP). However, this architecture also allows input from authorised remote nodes. Hence, other nodes are able to cooperate with the local PDP in order to find, for example, a good encryption mechanism which is a good trade off between resource consumption and the required security level, respecting the context of both nodes.

With this very abstract view on our approach it is possible to see how this architecture can reduce the resource consumption during the deployment phase of "long-life" wireless sensors. Installed security primitives may become subject to vulnerabilities or may be outdated. Their update or replacement can be conducted efficiently with HACS. Additionally, we tremendously increase the flexibility of existing approaches by offering a wider range of security mechanisms. As a consequence, this flexibility may also improve the resource consumption of sensors implementing HACS.

To specify this high-level view the next section will focus on cryptographic block ciphers and cryptographic hash functions. We explain which kind of building blocks and description languages can be chosen in order to implement the architecture outlined above.

4. DYNAMIC CRYPTOGRAPHIC SUITES

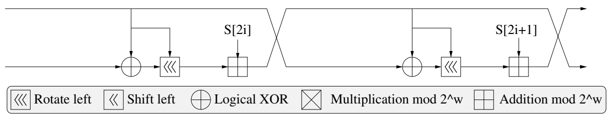
Efficient encodings of cryptographic primitives which would consume only very little space but yield very complex and flexible structure and functionality are the focus of this work. Please note that this approach does not aim at optimising single cryptographic primitives for space, performance, and low power consumption. Instead, it splits block ciphers into their building blocks and defines a mechanism to combine them in an easy and straightforward way. Obviously, this mechanism will result in less resource efficient primitives. However, at the same time it will increase flexibility and functional diversity by providing a cryptographic suite which can offer different primitives for different application scenarios. Further work will show how this approach can also effectively reduce the average power consumption.

The next questions are: Which building blocks for cryptographic primitives can be used? How do we combine them? Is the execution process for the resulting combination simple enough? How much overhead does it introduce? Sections 4.1 through 4.7 are going to address theses questions. For this purpose we are going to use the following terms: A *primitive* is a specific cryptographic algorithm (e.g. RC5, SHA-1). *Building blocks* or *code blocks* designate pieces of code, which execute a basic operation (S-Boxes, rotation of bytes, XORing, etc.) on data. The data used to steer the control flow is kept in *information blocks*, each of them containing information for the execution of a single code block. Finally algorithm-specific constant data (like a substitution matrix) are kept in *constant data blocks*, which can be used as input for code blocks.

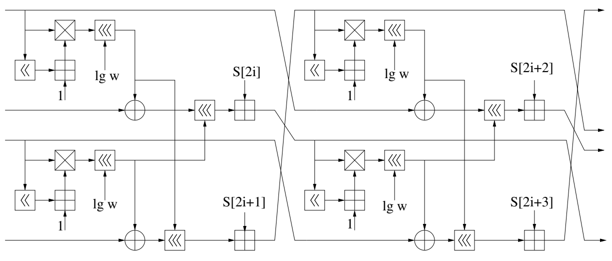
4.1 Building Blocks

At first sight, finding suitable building blocks for cryptographic primitives appears to be straightforward. Reasons for this are of historical and cryptographic nature.

In this work, we consider symmetric key block ciphers. Almost all of these primitives are based on the iterated applica-



(a) RC5 block cipher



(b) RC6 block cipher

Figure 2: Comparison of block ciphers RC5 and RC6

tion of a function f. This function can be composed of simple transformation rules which are based on permutation and substitution methods. The number of iterated applications of f can increase the security of such a scheme. Depending on the amount of information used in f and the manipulation of round-data these schemes are either called Feistel networks or substitution and permutation networks (SP). The uniformity of the networks of different cryptographic algorithms can be illustrated as a graph in which nodes represent the required operations (see Figure 2). Further, new cryptographic primitives are often successors of existing ones which show better crypt-analytical characteristics. As an example, RC6, constitutes a construct which is based on two parallel RC5 algorithms [29] with an additional multiplication [28] (also see Figure 2). Likewise MD5, SHA-x, and RIPEMD-x are based on MD4 [25]. Of course, this causes a high redundancy between these algorithms. We are not the first to discover these modular and redundant characteristic. Especially in the realm of hardware design numerous similar observations have been made. Hence, we can exploit these contributions and compile a list of the common building blocks of the most popular block ciphers (see Table 1). This list is a modified and slightly enhanced version of the list provided in [14, 15].

Although there are building blocks which are solely used by one algorithm we can identify numerous elements which are common to almost all of the primitives. Consequently, especially the latter elements will assure code reduction if used with our approach. This statement cannot be generalised, yet. The effective code reduction will depend on the variety of parameters relevant for the algorithms, e.g. the word size used, the block size processed, the number of rounds of the algorithm, etc.

Whether these building blocks have best granularity for our approach is subject for future research. However, we currently claim these building blocks to be a good trade-off between fine granular operations and coarse modularisation. Obviously, in the first case we have to face higher administrative overhead due to the description and combination of the single building blocks. The next sections are going to

Table 1: Building blocks for common block ciphers and hash functions

Type	Name		Arithmetic S-Box	Pseudo-random S-Box	Fixed Permutations	Modulo Addition	Modulo Multiplication	Fixed Shift/Rotations	Variable Shift/Rotations
Block Ciphers	DES/3DES	[21]		●	●			●	
	IDEA	[17]				●	●		●
	RC5	[29]				●			●
	RC6	[28]				●	●		
	AES	[6]	●		●			●	
	MARS	[4]		●		●	●	●	●
	Twofish	[30]		●		●	●	●	
	XTEA	[20]				●		●	
Hashes	MD2	[13]						●	
	MD4	[26]						●	
	MD5	[27]						●	
	SHA-x	[8]						●	
	RIPEMD-x	[7]						●	

address both of these issue and show how this overhead can be kept to a minimum.

4.2 Primitive Description

Comparable to the commonly used representation for ciphers (see Figure 2) we are going to use a graph representation to describe the proper combination of our cryptographic primitives. Of course, the graph used has to be more expressive than the visual representation as it has to properly describe how building blocks have to be combined in order to yield correct primitives. On the other hand the graph has to avoid redundancy as much as possible to guarantee low overhead. Accordingly, its processing has to be as simple as possible.

The primitive description shall be composed in a language, that allows to describe graphs of the needed complexity, identifying *data types* and *code blocks and their respective interfaces* unambiguously, such that the system is capable of mapping them onto existing building blocks. As algorithms will most likely be designed by humans, the language should be easily *readable* and *understandable*. This excludes complex text formats and binary formats (at least at the first stage).

4.3 Interpretation vs Compilation

A straightforward implementation might execute a given description via interpretation. A simple interpreter would read the description, split it into *instructions* (here: basic building blocks), retrieving, and executing them one by one. During the runtime every statement of the description will be re-evaluated before execution. While this is a simple and easily implementable approach (no need for generation of code, often no need for low-level system actions like address space management), it has its drawbacks as interpreted solutions tend to have a larger time and memory footprint. Especially the latter is of concern in modern embedded systems as memory tends to be the most limiting resource.

Because of these factors, we have chosen a different approach, which employs pre-compilation of the algorithm's description in order to reduce interpretation overhead. In this process, some code may be generated to execute on its own, reducing the interpretation of descriptions to a single run, which emits the respective code and/or data structures (see below). While this creates some additional workload when first implementing the algorithm, subsequent runs will be executed faster and with a smaller memory footprint. As algorithms don't change as often as they are executed, we believe this to be a reasonable trade-off.

4.4 Implementation via chained code blocks

This implementation is based on the idea of supplying the individual code block with enough information to guide the control flow themselves - a controlling (or interpreting) instance is not needed in this approach.

While the structure of an algorithm's description graph may contain forking (i.e. nodes with more than one output edge), execution on the machine will usually be linear. Effectively this requires us to transform the graph into a linear list of blocks to execute, probably storing some results in memory for consumption by other blocks (so while transferring the control to a block along one edge, a block along another edge processing the same results later will still have these available).

Once a linearisation (and storage for intermediate results) is found, the addresses of the code blocks (which can vary for different devices and thus cannot be part of the algorithm's description) must be resolved to actual values. This also applies to addresses of data blocks used in this algorithm, e.g. constant S-boxes in DES.

Finally all data gathered must be stored in a special form of the description, which consists of a series of *information blocks*, each corresponding to a code block. Every information block starts with the address of the code block and an offset to the next information block. These values are then followed by a series of arguments (defined by the code block), which may be addresses (for input or output of data), behaviour-influencing parameters (e.g. block sizes) or constants. The sequence of information blocks is terminated by a block only consisting of the address of a special *termination code block* which will end the execution and return to the caller. As every code block during execution is supplied with a information block according to its current role (e.g. XOR 4096 bytes starting at address 0x1000 with those starting at 0x2000 to 0x3000), it is able to execute its task as well as transferring control transfer to the next code block, supplying it with a new information block.

4.5 Pre-Compiler

As mentioned before, the graph needs to be transformed to the internal pre-compiled data structure as described above. This addresses various problems:

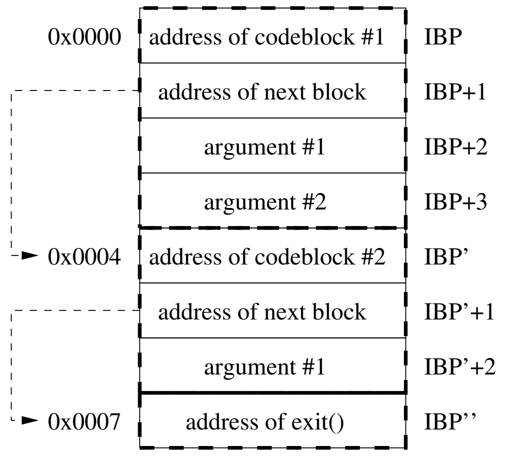


Figure 3: Example information block structure

1. *Linearisation:* The graph structure of the algorithm has to be transformed to a linear sequence of code blocks instead of a branching tree-like structure. Dependency-resolving algorithms can be used to create a linear order of execution of code blocks.

2. *Address resolution:* Names of code blocks and constant data referenced in the description have to be resolved to addresses on the target machine.

3. *Data storage allocation:* As intermediate results often consist of large blocks of data (more than a processor may hold in its internal registers), storage for this data must be found and allocated. This is a rather critical part of pre-compilation as memory seems to be one of the more critical physical resources in embedded systems.

4. *Storage:* As algorithm descriptions should be pre-compiled only once and be ready for execution in the future, the pre-compiled description should be stored in memory (possibly flash or EEPROM) in such a fashion that it can be retrieved by name or signature.

Although the implementation of the pre-compiler on the target system seems straightforward because of the easily available information (e.g. addresses) it might be more feasible to move this process to higher instances with more computational power and resources. While this may introduce additional overhead in form of communicating the necessary data (addresses, current memory layout of the device), it allows the pre-compilation to employ better optimisations techniques because the device restrictions do not apply in this environment.

4.6 Format of code blocks

The special form of the pre-compiled description imposes a special calling convention upon the code blocks that is specially crafted to efficiently use the information given in the pre-compiled description.

A register of the processor has to be selected as *information block pointer* (IBP). This register will always contain the address of the current information block. Every code block thus may retrieve the address of the next information block at the address *IBP+1* (also see Figure 3). When the IBP is set to this address (after the code block executed), the address *IBP* contains the start address of the next code block.

This allows us to create a generic template for code blocks (Memory[Addr] denotes the value in memory at address *Addr*):

```
begin_codeblock:
   ; parameters may be accessed using the values at
   ; addresses IBP+2, IBP+3...

   ...

   IBP = Memory[IBP+1]
   jump to Memory[IBP]
```

Depending on the code block parameter values may or may not be cached in registers, but this is up to the code block implementation. Also this may be used as a mere wrapper to a function using another calling convention or as a template for an assembly-implemented function.

This template establishes a clear interface for code blocks, which utilises the pre-compiled description structure to execute an algorithm without a supervising (and resource-consuming) interpreter instance.

4.7 Calling convention transition

While the special calling convention used in this implementation is acceptable within the execution of an algorithm, normally the surrounding system dictates some other calling convention. To create a usable implementation some transition code has to be written in order to allow external code to call algorithms and use their results. This requires two special code blocks for entry and exit.

4.7.1 Entry code

The entry function has to take parameters in the system's calling convention (a pointer to the pre-compiled algorithm description; lookup, fetching and possibly address substitution for input and output parameters can be handled by regular code), which will be loaded into the *IBP*. It may also be necessary to save some register values on the stack for later restoration if the surrounding system requires this. The entry function then transfers control to the address in `Memory[IBP]`, the first code block.

4.7.2 Exit code

The exit code has to implement the other half of the surrounding calling conventions. This primarily consists of possibly restoring stored registers off the stack and supplying a return value if necessary (although data mostly will be exchanged via memory).

4.7.3 Execution

These code blocks for entry and exit encapsulate the special calling convention used in HACS. The entry function can be called by the surrounding system like any other function. At this point the special calling convention used here is used, every code block calls its successor until the exit block is reached. This block returns to the system's calling convention. All of this is transparent to the surrounding environment, which may treat the algorithm like any normal function.

5. DISCUSSION

Before we conclude our paper this section discusses some of the characteristics of the presented approach. We especially consider security issues, the capabilities and limitations of our mechanisms, and start with arguing about the efficiency of our solution.

5.1 Efficiency considerations

Algorithms expressed in the given description, executed by code blocks will not reach the efficiency of monolithic, optimised code, but the flexibility we gain using our approach is much greater than the apparent overhead.

- *Code execution time overhead* depends strongly on the code blocks. As many code blocks will operate on large chunks of data, the block runtime will dominate the overall runtime, while the instructions used to maintain the control flow will use little time. Because of this, the actual time overhead should be minimal.
- *Memory consumption overhead* depends on the number of algorithms currently needed. The more algorithms have to be available, the more memory is conserved as the pre-compiled descriptions naturally tend to be smaller than the corresponding code. While initially there is some overhead for the maintenance structures (look up tables etc.) this impact is reduced when more and more algorithms are added.
- *Communication overhead* happens primarily whenever a new algorithm is added. As this should not be as frequent as the execution of an algorithm, we believe this is of little concern.

The execution of the algorithm could further be optimised by imposing more rules on the code blocks. Registers or specially assigned data areas (local on-chip memory for example) might be used to pass data between code blocks more efficiently. Of course, these characteristics have to be known and used by the pre-compiler. These data would be part of the memory management information supplied by the resource constrained device during pre-compilation. Further research has to show if this is a viable attempt to increase performance and reduce power usage.

5.2 Security considerations

As this implementation allows dynamically defined code execution, security has to be a consideration. First of all, new code blocks might be necessary or desirable. These should *only be supplied and accepted from the same authorised sources, which are also capable of code updates.* As the code blocks basically execute unsupervised, they have to be as authenticated as any other code to be executed on the machine.

While entering new algorithm descriptions into the system could be restricted in the same way as code updates, it may be eligible to allow unauthorised (in respect to code updates) clients to access data from the node while still employing cryptographic algorithms (e.g. a normal user of the system who may process the data but not change the node's code base). As stated before the needed cryptographic algorithms may depend on the current environment and are possibly not yet known to the resource restricted device. This scenario poses an interesting question, should the client be rejected or could it supply an algorithm description although it is not eligible for code updates?

An unchecked pre-compiled description may lead to various security breaches, as addresses of data and code blocks are part of the description. Still, it is possible to allow entry of new, unauthenticated algorithm descriptions, in this scenario though, the description has to be subject to consistency checking:

- *Code block addresses* may only point to the start of known code blocks.
- *Constant data block addresses* may only point to the start of known constant data blocks, additionally a possible length parameter may have to be checked to avoid reading past the end of the constant data block and potentially leaking data.
- *All data storage addresses* have to be checked in the same way to secure the system against buffer-overflow-alike vulnerabilities and data disclosure.

Meta-data for this consistency checking has to be supplied by the code block directory as the meaning of parameters and their usage as data fields solely depend on the implementation of the code blocks.

During the pre-compilation phase, the resource constrained device has to disclose some internal data, namely the addresses of various data and code blocks as well as some parts of the memory layout. While this in itself is benign information, past experience has shown that this information may be valuable to an attacker. Although *security by obscurity* itself has been shown to be infeasible, this at least has to be taken into consideration.

While allowing limited access to unauthorised sources most likely will add some complexity to the system, in some scenarios it may be feasible in order to restrict code access to rarely used machines and/or persons, while opening the flexibility of dynamic algorithm selection to all users of the system.

5.3 Capabilities and Limitations

With this approach we aim at *increasing the flexibility* of resource constrained devices to enable them to choose the best cryptographic primitive to save resources and to securely communicate with a wide range of other devices.

Due to the high level description of the cryptographic primitives we also *simplify the process* of *distributing and employing new primitives.* Instead of providing each platform with a new implementation we simply have to distribute the high level description and potentially a small number of additional building blocks.

Directly linked to this issue is the *simplification* of the *update and patching* process which will only require either the exchange of existing building blocks or graph representations of a primitive.

We also plan to use our graph model to *account for other resource constraints* such as small RAM or register sizes. We organise blocks in a tree structure. If we are able to specify in which order these cells are executed (e.g. by using work flow graphs) we can also compute intermediate results or key schedules and store them on different memory until needed for further processing. Similarly, we can split up word sizes and tailor the processing to the available word size of the registers of the hardware platform.

Additionally, we also imagine to *exploit* the *dependencies* between primitives. As explained above many primitives are modifications of already existing algorithms. Thus, to express advanced or evolved primitives which are based on others we think of using linear combinations. The vectors of these combinations would be graph representations of the underlying cryptographic primitive. Their scalars would correspond to parameters, such as, permutation matrices, round parameters, or building blocks combining the graphs.

The presented solution will require a dynamic loading process which is not supported in widely employed operating systems for resource constrained architectures such as TinyOS [12]. However, recently developed operating systems and extensions can account for this lack of functionality [1, 2, 3, 9, 18]. Research has to show what implications the dynamic loading process will have on resource consumption (in terms of battery, memory, processor cycles, etc.).

Our approach is not able to also effectively address asymmetric cryptographic primitives. The underlying principles for these algorithms are of mathematical nature. Therefore, within our research we will have to find out whether we can also identify simple building blocks which can be used to also yield asymmetric cryptographic primitives.

6. CONCLUSIONS AND OUTLOOK

This contribution presented a new approach which allows the provisioning of sensor nodes or similarly resource-restricted devices with a large variety of cryptographic primitives. To achieve this goal we explained how the characteristics of block ciphers can be exploited and assembled a list of primitive building blocks. We further showed two possible ways - pre-compilation and interpretation - to efficiently combine these building blocks to construct existing block ciphers. The presented approach also enables a device to use a large variety of cryptographic primitives. This may not be optimal in terms of resource consumption. However, we aim at increasing the flexibility and variety of use of the device. Due to the ability to choose a cryptographic primitive which offers the best trade-off between security and resource consumption in a specific context we assume that our approach will decrease the average power consumption of a device.

In order to implement a working system as described in this paper, several steps have yet to be taken, which will be part of our future research. At first, a specification for the basic building blocks and their interfaces is required. Based on that an implementation of these building blocks on different target architectures will be provided. With a formal specification of our description language we will then be able to combine the building blocks to valid cryptographic primitives. Of course, this pre-compilation process has to be developed and compared with a naive interpretation process. A thorough specification of the communication protocols between nodes (especially during pre-compilation) will be required in order to avoid possible new attack vectors which are based on our solution. To finally validate, compare, and assess our approach with existing solutions appropriate test-cases will be designed and simulated.

7. ACKNOWLEDGEMENTS

This work has been supported by the European Union, through the FET Project BIONETS.

8. REFERENCES

[1] R. Balani, C.-C. Han, R. K. Rengaswamy, I. Tsigkogiannis, and M. B. Srivastava. Multi-level software reconfiguration for sensor networks. In *EMSOFT*, pages 112–121. ACM, 2006.

[2] S. Beyer, R. Taylor, and K. Mayes. Operating system support for dynamic code loading in sensor networks. In *PerCom Workshops*, pages 311–315, 2006.

[3] J. Blumenthal and D. Timmermann. Resource-aware service architecture for mobile services in wireless sensor networks. In *ICWMC*, page 34, 2006.

[4] C. Burwick, D. Coppersmith, E. D'Avignon, R. Gennaro, S. Halevi, C. Jutla, S. Matyas, L. O'Connor, M. Peyravian, D. Safford, and N. Zunic. MARS — A Candidate Cipher for AES, June 1998.

[5] C. Chigan, Y. Ye, and L. Li. Balancing Security Against Performance in Wireless Ad Hoc and Sensor Networks. In *Vehicular Technology Conference, 2004*, volume 7, pages 4735–4739, September 2004.

[6] J. Daemen and V. Rijmen. The Block Cipher Rijndael. In *CARDIS*, pages 277–284, 1998.

[7] H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In *Fast Software Encryption*, pages 71–82, 1996.

[8] D. Eastlake 3rd and P. Jones. US Secure Hash Algorithm 1 (SHA1). RFC 3174 (Informational), Sept. 2001.

[9] H. A. et al. MANTIS: system support for MultimodAl NeTworks of In-situ Sensors. In *WSNA'03*, pages 50–59, New York, NY, USA, 2003. ACM Press.

[10] J. Gehrke and S. Madden. Query processing in sensor networks. *IEEE Pervasive Computing*, 03(1):46–55, 2004.

[11] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy. The platforms enabling wireless sensor networks. *Commun. ACM*, 47(6):41–46, 2004.

[12] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[13] B. Kaliski. The MD2 Message-Digest Algorithm. RFC 1319 (Informational), Apr. 1992.

[14] J.-P. Kaps. *Cryptography for Ultra-Low Power Devices*. PhD thesis, ECE Department, Worcester Polytechnic Institute, Massachusetts, USA, May 2006.

[15] J.-P. Kaps, G. Gaubatz, and B. Sunar. Cryptography on a speck of dust. *Computer*, 40(2):38–44, Feb 2007.

[16] C. Karlof, N. Sastry, and D. Wagner. TinySec: A Link Layer Security Architecture for Wireless Sensor Networks. In *SenSys'04*, pages 162–175, Baltimore, November 2004.

[17] X. Lai, J. L. Massey, and S. Murphy. Markov ciphers and differential cryptanalysis. *Lecture Notes in Computer Science*, 547:17–38, 1991.

[18] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.

[19] G. E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*, 38(8), April 1965.

[20] R. Needham and D. Wheeler. TEA extensions. Technical report, University of Cambridge, Cambridge, UK, 1997.

[21] N. I. of Standards Technology (NIST). Data Encryption Standard (DES), Oct. 1999.

[22] A. Perrig, R. Szewczyk, and V. Wen. SPINS: Security Suite for Sensor Networks. In *Proc. of ACM MobiCom*, 2001.

[23] A. Poschmann, D. Westhoff, and A. Weimerskirch. Dynamic Code Update for the Efficient Usage of Security Components in WSNs. In *KiVS*, pages 445 – 455, Berlin, Mar. 2007. VDE Verlag.

[24] N. R. Prasad and M. Ruggieri. Adaptive Security for Low Data Rate Networks. *Wireless Personal Communication*, 29(3-4):323–350, 2004.

[25] B. Preneel. *Analysis and design of cryptographic hash functions*. PhD thesis, Katholieke Universiteit Leuven, 1993.

[26] R. Rivest. The MD4 Message-Digest Algorithm. RFC 1320 (Informational), Apr. 1992.

[27] R. Rivest. The MD5 Message-Digest Algorithm . RFC 1321 (Informational), Apr. 1992.

[28] R. Rivest, M. J. B. Robshaw, R. Sidney, and Y. L. Yin. The RC6 TM Block Cipher. In *First Advanced Encryption Standard (AES) Conference*, 1998.

[29] R. L. Rivest. The RC5 encryption algorithm. In *Proceedings of the Second International Workshop on Fast Software Encryption (FSE)*, pages 86–96, 1994.

[30] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson. *The Twofish encryption algorithm: a 128-bit block cipher*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[31] D. Steel. Smart Dust. Technical report, University of Houston, Information Systems Research Center, Houston, March 2005.