

LASE: Layered Approach For Sensor Security and Efficiency

Prem Uppuluri

Department of Computer Science
University of Missouri Kansas City
email: uppulurip@umkc.edu

Samik Basu

Department of Computer Science
Iowa State University
email: sbasu@cs.iastate.edu

Abstract

In recent times, sensor security and reliability has become an important concern due to the growing applicability of sensor-based networking infrastructures. In this paper, we present here a novel multi-layer framework to protect confidentiality and integrity of data stored on sensors. The framework allows automatic and fast development of security extensions to the sensor operating system.

1. Introduction

Sensor networks consisting of several thousands of computing devices, called *sensors*, are being widely used in situations where using traditional networking infrastructures is practically infeasible. These applications include those which are based in remote physical locations or in hostile enemy territories. Their adoption in such applications is primarily due to the characteristics of the sensor – low cost, small sized, portable, battery operated devices that communicate through a wireless network using radio signals. While on the plus side these properties allow such networks to be deployed easily without much physical supervision, on the minus side, they restrict the computational power of the sensors. In fact, sensors run bare-bones software with very small foot-prints. As a consequence, such networks face several challenges which are unique to them. One such challenge is in securing them – a key problem since several sensor network applications such as military environments rely on transmitting classified data.

Most of the current state of the art in sensor network security research is focused on developing protocols for secure routing [14, 20, 23, 24] and transmission of data in an encrypted format on the network [12, 24]. However, in addition to securing data in transit on networks, security of the operating system that runs on a sensor is also critical. This is because, sensors may store security critical data such as, shared secret keys to send

and receive packets confidentially across the sensor network. Compromise of a sensor OS will lead to loss of such data, possibly compromising other sensors as well. Figure 4 illustrates some of these threats for the popular TinyOS [17] sensor operating system. However, in spite of the threats, not much work has been done in securing sensor OSes. Recent research [25] ensures security of data on a sensor by using a probabilistic model in which a limit is placed on the number of sensors sharing a common secret key, thus reducing the fallout when the key is compromised. While the probabilistic model is efficient in containing the damage due to an attack, it does not address the larger problem of protecting critical data which may not always be secret keys – hence, security mechanisms local to a sensor are required. Moreover, since sensor networks consist of thousands of sensors, we believe that enforcing confinement of shared keys within small groups of sensors is difficult – [25] does not address this issue. [14] proposes the application of anomaly/misuse based intrusion detection systems for sensor security. While this idea has merit, it is not clear how sensors can be managed to deal with (a) effects of high false alarms in anomaly based detection and (b) the need for constant updates to signatures in misuse based intrusion detection.

Salient contributions. In this paper we present a framework for TinyOS that (a) secures it against known and unknown attacks without requiring any supervision, (b) provides a software tamper-resistant lock on sensors, which ensures that in face of repeated attacks, the sensor self-destructs itself before its data can be compromised and (c) improves the efficiency of TinyOS. The key contributions of our work are:

- A behavioral control model which automatically captures the actual behavior of the TinyOS and detects attacks as deviations from the actual behavior. In this paper we discuss an algorithm to generate these models by static analysis of TinyOS binaries and a novel optimization algorithm that increases the runtime efficiency of such models.

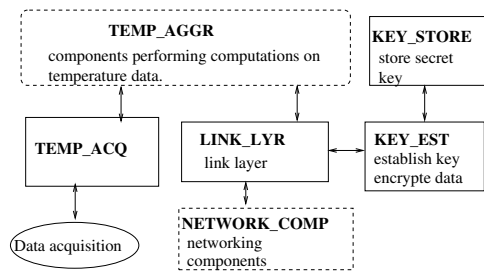


Figure 1. TinyOS Example

```

module KEY_STORE {
  provides { interface keystorage }
  uses { }
  interface keystorage {
    command int get_key();
    command void set_key();
    event int got_key(); }
  implementation {
    /* implementation of interface */
    /* function */
  }
}

```

Figure 2. Component KEY_STORE

- A framework for developing security extensions for TinyOS. The framework supports different types of security extensions. In addition, it is computationally efficient. Specifically, experiments on its performance on Linux daemons such as `ftpd` and `httpd`, demonstrated that it introduces overheads of less than 5% and moreover, the overhead was independent of the complexity of the security extension [19]. In this paper, we discuss its application to the domain of TinyOS.

In addition to security, the framework also supports extensions which make the execution of TinyOS more efficient. It is based on the observation that a substantial amount of power in TinyOS is spent in accepting an external event, identifying it and dispatching it for processing [3]. Hence, by “predicting” the set of all events that can occur next, the framework improves the efficiency.

We begin our discussion with an example of a TinyOS installation in Section 2. In Section 3 we discuss our framework in detail followed by conclusion in Section 4.

2. Example TinyOS application

TinyOS is an event-based operating system with a very small footprint (~400Kb)[17]. TinyOS applications have two entities: *components* which implement the application functionality and provide bi-directional

interfaces to access it ; and a *wiring specification* which assembles components together by defining *how* and *which* components can interact with each other using the interfaces provided by the components. The applications are written in a high-level language called *nesC*. An example of a TinyOS installation is shown in Figure 1. Its components are as follows:

- TEMP_ACQ: collects temperature information using a Berkley mote sensor [2].
- LINK_LYR: receives and injects packets from/to other sensors in the network.
- TEMP_AGGR: gets data collected by TEMP_ACQ and LINK_LYR.
- KEY_EST: used by LINK_LYR to establish a secret key during sensor bootstrapping as well as (en/de) crypting packets when injecting/accepting packets from the network.
- KEY_STORE: stores the secret key obtained after key exchange.
- NETWORK_COMP: a set of components that implement network capabilities used by LINK_LYR to inject and get packets.

Example component KEY_STORE is shown in Figure 2. It has (a) an interface containing **commands**, `key_get` and `key_set`, that other components can use to get or set the secret key; (b) **events**, such as `got_key`, which are callback functions that components using KEY_STORE’s interface need to implement; and (c) interfaces that KEY_STORE (empty in the example) can use. The `implementation` section contains the code for the commands defined in the interface. Figure 3, gives the skeleton of the interfaces, provided by the different components. For instance, LINK_LYR provides commands `inject_pkt` and `receive_pkt` whose invocation cause packets to be injected and received. Similarly SECURE_COMP, can be used to (en/de)crypt packets (`encrypt_pkt`/`decrypt_pkt`) or authenticate sensors (`auth_sensor`). Components can contain local data such as `int secret_key` which stores the secret key.

Security Issues. TinyOS does not offer any protection due to efficiency reasons. Figure 4 illustrates some of the possible threats on TinyOS. Key vulnerabilities include the compromise of the *wiring* specification and the classified/secret data stored on the TinyOS.

Efficiency Issues. Key issues which affect efficiency of TinyOS application: (a) time spent in waiting for an external event (such as input data or from another sensor) to occur and identifying that event, and (b) unused components consume power if they are run.

Security Requirements	Examples of Attacks
Confidentiality of data	Illegal access to secret/classified data stored on TinyOS such as secret keys and wiring specification can compromise not only the sensor but its peers and allow network traffic to be decrypted.
Trojan Attacks	Unauthorized addition of Trojan components and/or illegal modifications to <i>wiring</i> specification changing <i>how</i> and <i>what</i> components interact with each other
Physical Security	Attacker can physically access the sensor node and read its contents or change them by flashing the sensor memory.

Figure 4. Example attacks on sensor networks which manifest through TinyOS

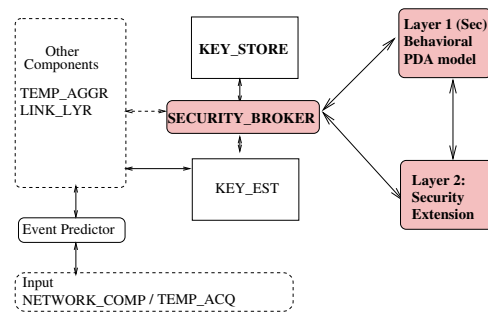


Figure 5. System Architecture

Component Name	LINK_LYR	KEY_STORE	SECURE_COMP
PROVIDES	inject_pkt receive_pkt retransmit_pkt ...	set_key get_key	validate_key kerb_auth auth_sensor ... encrypt_pkt decrypt_pkt
USES	keychange encrypt_pkt decrypt_pkt auth_sensor		set_key get_key
local-data	...	int secret_key	...

Figure 3. Components in example TinyOS application

3. Overview of Our Framework

Figure 5 illustrates the architecture of our system. The salient components of this architecture are:

- **Security broker (SECURITY_BROKER):** Every component that needs to be protected (henceforth referred to as *protected* component), has a SECURITY_BROKER associated with it. It intercepts every command invocation to the protected component and determines if the invocation is valid/invalid using a layered approach.

Layer 1: Control Behavioral Model (CBM) automatically captures the actual behavior of the TinyOS components in terms of their interactions with the protected component. These models are developed by static analysis of TinyOS binaries.

Layer 2: Security Extension allows users to customize/develop security extensions to restrict access to the protected component.

In this architecture, command invocations to the protected component are intercepted by the SE-

CURITY_BROKER and then passed to the Layer 1. If Layer 1 detects an attack, the SECURITY_BROKER uses an event implemented by this layer to launch an appropriate response, for instance, one which disallows the command. When it is not an attack, the command is dispatched to Layer 2 (if present) and executes the security extensions for that command. Note that the layered approach separates the detection mechanisms that can be deployed automatically (Layer 1) from the one that requires user-defined security extensions (Layer 2). This allows, for the former to act as a filter for inputs to the latter layer – a computationally intensive layer, thereby, adding to the efficiency of the overall mechanism of SECURITY_BROKER.

- **EVENT_PREDICTOR:** To improve efficiency in terms of power usage, [3] suggests that the TinyOS should be in a passive mode for most of the time. However, when an event occurs, identifying that event and determining the action to take makes a sensor move from passive to active mode leading to increased power consumption. By using the policy specification language mentioned above, users can define the *next set* of possible events that a TinyOS is likely to see. This will allow faster identification and deployment of functions to handle the event.

We discuss the architecture in detail in the next sections.

3.1. Security Broker: Control Behavioral Model

The central theme of Layer 1 is the development of a control behavioral model (CBM) that is precise and can be employed for fast identification of incorrect observable behavior. We present in this section the existing techniques applied to identify such models followed by a novel technique for generating precise CBM which can be used efficiently.

Typically, program control behavior is represented using *control flow graphs* (CFG), a graph consisting of

states represented by program counter values and inter-state transitions labeled by system calls. Such graphs are generated by static analysis of TinyOS. Filtering proceeds by run-time monitoring of observable actions. Deviations of sequence of observable action from the control flow graph is flagged as anomalous (potentially harmful). Two important requirements of this technique, as eluded in [10], are

1. **Precision:** the model must capture all possible sequence of actions that are deemed as correct system behavior and nothing else. This stems from the underlying requirements that correct behavior must not be classified as erroneous and anomalous behavior must not go unnoticed.
2. **Efficiency:** monitoring observable behavior must incur minimal overhead in time and space usage.

Unfortunately, enhancing both precision and efficiency is a difficult, if not impossible, task owing to the fact the precision requires incorporating minute details of the system in the model which in turns slows down the monitoring phase thereby reducing efficiency. We present, in this paper, a novel approach which enhances precision without incurring loss of efficiency.

Background. Each procedure, represented by a CFG, has an entry state and can have multiple exit states (depending on the relative position of `return` statements in the procedure). Figure 6(a) shows CFGs for a program with a *main*, *line* and *end* procedures. The procedure *line* can be invoked from two different call sites, one each in *main* and *end*. Global CFG, representing the global control behavior of a program, is constructed from local CFGs by simply introducing ϵ (empty) transitions from the call site to the start state of the called procedure and from the exit points of a procedure to return locations in its potential callers. Subsequently the transitions labeled by calls to the procedures in local CFGs are discarded. Figure 6(b) presents global CFG for the procedure *main*.

Context Insensitivity in CFG. A correct program path with respect to its call-return pattern requires that when a procedure exits it returns control to the site of its most recent call [18]. A CFG model does not keep track of location to which a program control should return once a procedure exits, i.e., *context* information of a call is lost. Such context insensitivity classifies paths with unmatched calls and returns as valid execution sequences in the program. An example of such infeasible path in the global CFG is illustrated using *dotted* lines in Figure 6(b). CFG model does not to classify the transition from exit state of *line* to return location at *end* as incorrect and hence fails to detect the in-feasibility. Note that, an infeasible path results from one/more *bad* edges from

the callee's exit state to the a return location of one of its potential callers.

A malicious user manipulating the executing program may use such edges in the model as an exploit – [10]

Context Sensitive Model: Push-Down Systems. Push-down system (PDS) captures, in addition to the intra-procedural control structure, the correct call-return pattern (context) of the program under normal circumstances. This is achieved by explicitly keeping track of the execution stack of the program whose behavior is being modeled by the PDS. Unlike CFG transition, which is between a pair of states, a PDS transition represents the change in the execution stack of the program: $\text{tos} \leftrightarrow \text{set}$ where `tos` is the current top-of-stack and the `set` represents the sequence of program locations pushed into the stack after the statement at `tos` is executed.

In the recent past, a number of efficient techniques has been proposed to analyze programs with (recursive) procedures using their push-down system representation [6, 8]. Following these lines, [10] proposed a runtime monitoring technique IDS using PDS. Whenever the monitored program makes a jump from one procedure to another, the return location in the caller is *pushed* in a stack (we will refer to this as *monitor-stack*). On the other hand, if the monitored program exits a procedure and goes to a state in another procedure, the execution is deemed correct only when the destination state is present in the top of the monitor-stack. In the event the transition is allowed, the top of monitor-stack is *popped* out.

Going back to the example in Figure 6(c), every inter-procedural transition is labeled by the operation on the monitor stack (`push(A)`, `pop(A)` etc). An inter-procedural transition is feasible only when the operation on the monitor-stack is feasible. Such restriction removes the presence of infeasible paths caused by unmatched calls and returns in the model.

Space Inefficiency in PDS. The PDS-based monitoring technique suffers from a major drawback of imposing a significantly large space overhead. This is attributed to the fact, the monitor-stack is potentially as large as the execution stack of the program being monitored. [10] proposes an optimization technique which makes the PDS monitoring mechanism practically usable at the cost of losing precision to certain degree. The optimization, termed as *hybrid model monitoring scheme*, requires finitizing or pre-fixing the size of the monitor-stack, thereby reducing the space usage. However, the technique involving hybrid models only allows context-sensitive monitoring for non-recursive procedure; the call-return context is not considered for recursive ones.

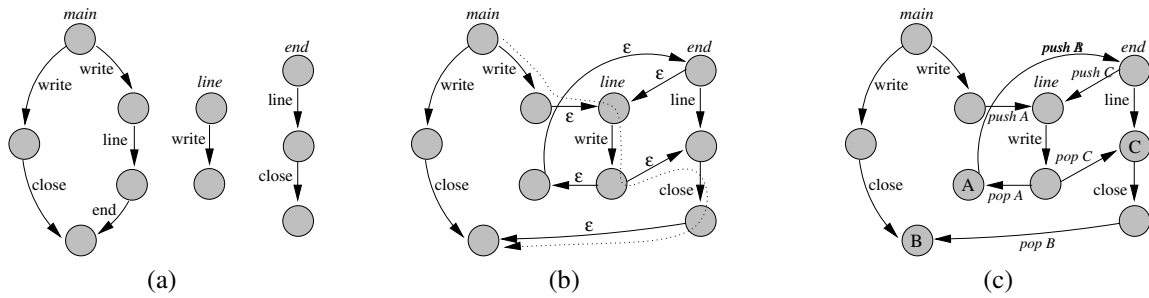


Figure 6. (a) Local CFG, (b) global CFG and (c) *push* and *pop* operations for PDS model

3.1.1 Our Solution: Annotated CFG

In this section, we propose a novel technique which is as precise as PDS-model monitoring technique and is comparable to CFG-model monitoring technique in terms of space usage, and thus caters to the basic requirements for securing TinyOS.

At its core, the technique is based on introducing a set of auxiliary integer variables one each for call transition in a CFG. We will refer to these variables as *PROcedure conteXt Indicator* (Proxi) and the corresponding CFG as *proxi-annotated control-flow graph*. At a high level, these variables record the number of times the a procedure is invoked in an execution sequence from a specific call site.

Definition 1 A *proxi-annotated CFG* of a procedure p is a tuple $PCFG_p = (S, s_0, S_E, \rightarrow, \mathcal{L}, \mathcal{V})$, where $CFG_p = (S, s_0, S_E, \rightarrow, \mathcal{L})$ and $\mathcal{V} \subseteq \mathcal{P} \times S$ with \mathcal{P} is the set of procedures called by p . CFG_p is the control flow graph for procedure p with S is the set of state, $s_0 \in S$ is the start state, $S_E \subseteq S$ is the set of exit states, $\rightarrow \subseteq S \times \mathcal{L} \times S$ is the set of transitions and \mathcal{L} is the set containing system call names and ϵ . \square

A proxi variable is denoted by $\langle q, s \rangle$ is associated with the called procedure q and the return location/state s of the caller. Following we enumerate its application in detecting correct inter-procedural execution paths while monitoring.

1. *Initialization*: Proxi variables are initialized to 0.
2. *Incrementing a proxi variable*: At the time of monitoring, an execution step corresponding to a call p with a return location at r_k , results in incrementation of all the non-zero proxi variables associated with p : $\forall i \neq k. \langle p, r_i \rangle = \langle p, r_i \rangle + 1$, where r_i are the return locations in the potential callers. It also increments $\langle p, r_k \rangle$ by 1. Monitoring proceeds from the start state of p .
3. *Decrementing a proxi variable*: An execution step corresponding to return from a procedure p must lead the control to the return location r_k

such that $\langle p, r_k \rangle$ is minimum among all the non-zero proxi variables associated to p : $\langle p, r_k \rangle = \min(\forall i. \langle p, r_i \rangle)$. All the non-zero proxi variables of called procedure p are decremented by one. Monitoring proceeds from the state r_k in the caller.

Note that, non-zero value of a proxi variable records the number of (self/mutual) recursive calls to a procedure. An execution step involving return from a procedure is deemed feasible only if the proxi variable corresponding to the return location has the minimum value among all the non-zero proxi variables associated with the called procedure. Proxi-annotated CFG model, therefore, classifies correct call-return pattern of programs without using monitor-stack required for PDS model monitoring technique, i.e., space inefficiency caused due to the monitor-stack is nullified without any loss in precision.

Theorem 1 An execution sequence is classified by a push-down model as feasible ,if and only if, it is classified as feasible by proxi-annotated control flow graph model. \square

Figure 7 shows the updates to the proxi variables for monitor-stack operations in PDS corresponding to the example in Figure 6(c). The path shown in Figure 6(b) is classified as infeasible by PDS model and also the proxi-annotated CFG. The transition from *main* to *line* increments $\langle line, A \rangle (= 1)$. The subsequent observable transition from *line* to *end* is not allowed as $\langle line, C \rangle$ is equal to zero and is not minimum proxi variable associated with called procedure *line*.

3.2. Security Broker: Facilitating Security Extensions

Layer 2 allows a user to implement security extensions. In particular, it supports extensions based on specifying and enforcing security policies using a high-level specification language. Examples of such extensions include access control policies, [7, 9], intrusion detection systems [19, 13] and program confinement [15].

Monitor-stack operation	Proxi-variable constraint	Proxi-variable operations
Push A	-	$\langle line, A \rangle ++$ $\langle line, C \rangle ++$ (if $\langle line, C \rangle \neq 0$)
Push C	-	$\langle line, C \rangle ++$ $\langle line, A \rangle ++$ (if $\langle line, A \rangle \neq 0$)
Push B	-	$\langle end, B \rangle ++$
Pop A	$\langle line, A \rangle$ non-zero min	$\langle line, A \rangle --$ $\langle line, C \rangle --$ (if $\langle line, C \rangle \neq 0$)
Pop A	$\langle line, C \rangle$ is non-zero min	$\langle line, C \rangle --$ $\langle line, A \rangle --$ (if $\langle line, A \rangle \neq 0$)
Pop B	$\langle end, B \rangle$ non-zero min	$\langle end, B \rangle --$

Figure 7. PDS Vs. Proxi-Annotated CFG

In order to support different types of extensions, the language must be able to express history sensitive sequences. For instance, in the TinyOS example mentioned earlier, consider the following security policy: component KEY_EST can only invoke the command get_key implemented by component KEY_STORE, when LINK_LYR has asked KEY_EST to encrypt a packet. Capturing this policy requires ability to express events such as commands (*get_key*), keep track of the sequence of command invocations and relationships between their arguments over time interval.

This rules out regular expressions (regexs) which are widely used in specifying and enforcing policies, e.g., in tcpdump [1] and the BSD packet filter [16]. In particular [5] has proved that regular expressions can only be used to specify safety policies. On the other hand, though general purpose languages such as nesC (similar to C/C++ but for TinyOS) can easily express such policies, the policies are not recursively enumerable (re). This is because nesC has the same expressive power as a Turing machine and hence policy enforcement is an undecidable problem.

We developed a language as part of our earlier work on system security [19, 22, 21], called behavioral modeling specification language (BMSL). BMSL extends the familiar pattern matching constructs of regular expressions to the domain of events with arguments. Security policies specified using BMSL are enforced using an efficient enforcement mechanism called extended finite state automaton (EFSA) [21]. The expressiveness of BMSL policies and the efficiency of pattern matching using EFSA's makes them suitable for developing security extensions for TinyOS. Specifically, BMSL policies have the same expressive power as extended finite state machines, a formalism commonly used in formal methods literature [11]. In addition, experimental results demonstrate that policy enforcement using EFSA is efficient causing overheads of less than 5%, for each

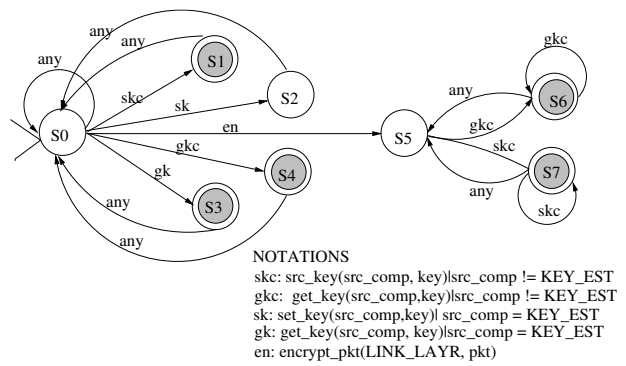


Figure 9. EFSA for KEY_STORE security policies

command invocation [22].

In the rest of the section, we discuss the policy specification language and its use in our framework.

BMSL Policy Specification Language. The most primitive pattern in BMSL consists of a single command/event with a reaction, such as, $a(x) | x > 3 \rightarrow term()$. Here, the pattern is characterized by a command/event name (*a*), arguments (*x*) and a boolean valued condition containing simple comparison/arithmetic operators ($x > 3$) and/or external functions in general purpose languages such as C/C++. The meaning (semantics) of pattern is given in terms of the system call histories that it captures. For instance, the above primitive pattern captures all commands/events *a* whose argument values are greater than 3. Rules have reactions, such as *term()* written in C/C++, which are invoked when the patterns are matched.

Complex patterns are formed from primitive patterns using: sequencing (" \cdot "), alternation (" $|$ ") and closure (" $*$ ") operators. An example of a complex pattern is: $a(x) | (x > 3) \cdot b \cdot c(y) | (y < 5)$, which consists of the primitive patterns $a(x) | (x > 3)$, *b* and $c(y) | (y < 5)$. It captures the set of all histories which start with an command/event *a* whose argument is greater than 3, followed by a sequence of zero or more *b*'s, and ending with command/event *c* whose argument is less than the value 5. BMSL's have very efficient enforcement models called *extended finite state automaton* (EFSA) which extend finite state automaton to remember context information across states.

Using BMSL for TinyOS Extension Development.

The observable behavior of two interacting TinyOS components is the set of sequences of commands/events invoked by them. Security policies can thus be specified in terms of these sequences. Using BMSL, we specify such policies over commands and events. While enforc-

```

/* Restriction on getting/setting secret keys */
(get_key(src_comp) || set_key(src_comp, key)) | src_comp ≠ KEY_EST → { destroy_key() }
/* Complex Rule: KEY_EST can only get key when LINK_LAYR wants to encrypt packet */
!(encrypt_pkt(LINK_LAYR, pkt)) * get_key(KEY_EST) → { term() }

```

Figure 8. Simple policies for securing KEY_STORE component

```

module Layer2 {
provides { interface patternmatch_efsa }
...
interface patternmatch_efsa
  command match(command, command_arg_structure);
  event int response_match(reactionlaunched);
  implementation {
  /* EFSA in nesC */
  }
}

```

Figure 10. Layer 2 component using EFSA

ing policies over interactions with a protected component, we add an extra argument, source id (*src_comp*), to the commands/events that defines which component is interacting with the protected component. We developed a security extension which prevents invalid accesses to the KEY_STORE component in the example TinyOS application. Valid accesses to the component occur when the LINK_LAYR component on each sensor initiates a secure key exchange. Since our research does not focus on the protocols, we assume that it is one of the different protocols currently being used in sensor networks [25, 23]. Key exchange is initiated only when the sensor network is deployed or when a new sensor is added to an existing network. Hence, the key is set in any sensor only once in the KEY_STORE component. Once the key is set, it can be used by the LINK_LAYR to (en/de)crypt packets which it injects/accepts from the network. Based on this behavior we develop two security policies which are illustrated in Figure 8 and explained below:

- No component other than KEY_EST can get or set the secret key.
- If component KEY_EST wants to access the key, it can only do so, provided the LINK_LAYR has initiated a request to encrypt a packet (using *encrypt_pkt* command)

These policies are translated into an EFSA. A schematic illustration of the example EFSA for the policies is shown in Figure 9. The dark circles representing states *S1*, *S4*, *S6* and *S7* are final states which trigger the response *destroy_key*, which erases the secret key and *S3* triggers the response *term*. EFSA is generated as a nesC program, and implemented as part of a com-

```

/* A and B are sensors */
1. A and B agree on a prime number P and integer G.
/* A and B exchange secret key */
2. A sends B computation involving P and G.
3. B sends A computation involving P and G.
4. Both compute secret key s
5. A and B use s to send encrypted packets to each other

```

Figure 11. Abstraction of Diffie Hellman Protocol

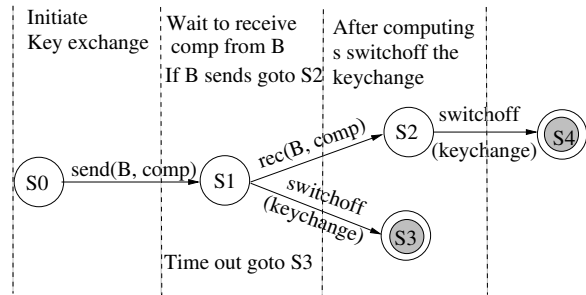


Figure 12. EFSA of the Event Predictor when using Diffie Hellman Protocol

ponent. Figure 10 illustrates this component. It provides a function *match*, which takes as an argument the intercepted command/event (intercepted by the SECURITY_BROKER component) and its arguments. It then uses the EFSA to match the command. The SECURITY_BROKER can invoke responses to the match using the event *response_match*.

Responses to violations of security policies. In the above example, functions *destroy_key* and *term* are used to respond to policy violations. Function *destroy_key*, simply erases the secret key. This ensures that even if the attack succeeds, the sensor is unusable. Hence, the response creates a software *tamper resistant* mechanism. The reaction *term* – is more passive, it simply terminates the command/event.

3.3. Event Predictor

The purpose of an event predictor is to compute the next possible set of events that can be received by the

sensor. In our approach this can be based on knowledge of either the protocol being used or the interaction between components. Once such commands are determined, the event predictors, store pointers to the set of all next “possible” commands (called cached commands). When an external event occurs, such as, receiving a data packet, event predictor first checks if the event can be handled by one of the cached commands of the component. Since the number of cached commands is a subset of the total number of commands, in an average case it improves efficiency in handling the event. This concept is very similar to the associative caches used by operating systems in computer architecture. The event predictor is analogous to the cache replacement algorithm. We use the EFSAs discussed earlier as such predictors. Specifically, BMSL security policies are used to capture the protocol information or the interaction between components. The policies specify the current event and the next possible set of events. The generated EFSAs corresponding these specifications are the event predictors. When an event pertaining to the protocol occurs, EFSAs trigger transitions to the next possible set of expected events. For instance, consider an abstract version of the Diffie-Hellman key exchange protocol [4] between two sensors *A* and *B* as illustrated in Figure 11. The EFSAs for the event predictor is shown in Figure 12. Key exchange is being performed by the KEY_EST component. When LINK_LAYER of *A* invokes the command `keychange` which is part of KEY_EST component, `keychange` sends some computation to *B*. The event predictor then expects either a reply from *B* or a timeout. It switch-offs of all components. When it receives any event, it first wakes up `keychange` and sends that message to it. If `keychange` cannot handle the message, i.e., the message is of the wrong type, the event predictor then goes through the entire process of determining the event type. We note that this is a preliminary idea and needs further experimentation.

4. Conclusions

We discussed a framework which supports: (a) basic security based on identifying deviations on TinyOS component interactions from their actual behavior and (b) a powerful security extension development mechanism based on specifying and enforcing security policies. The efficiency of this layered framework is critical in the setting of sensor security. We presented an optimization algorithm to make deviation detection (a) efficient and precise. In addition, our prior experiments, on policy-based security assurance for traditional operating system, provide strong testimony of its efficient applicability to TinyOS. As future work, we would like to

deploy our methodology in sensor network and measure its efficiency in terms of battery consumption.

References

- [1] Tcpdump. In *UNIX man pages*.
- [2] CROSSBOW. <http://www.xbow.com/>.
- [3] D. Culler, E. Brewer, and D. Wagner. Slap. In <http://webs.cs.berkeley.edu/SLAP-proposal.doc>.
- [4] W. Diffie and M. E. Hellman. New directions in cryptography. In *IEEE Trans on Info Th.* 22(1976), 644-654.
- [5] U. Erlingsson and F. B. Schneider. Sasi enforcement of security policies. In *NSPW*, 1999.
- [6] A. B. et al. Reachability analysis of pushdown automata. In *Concurrency Theory (CONCURR)*, 1997.
- [7] C. C. et al. SubDomain: Parsimonious server security. In *LISA 2000*.
- [8] J. E. et al. Efficient algorithms for model checking pushdown systems. In *Computer-Aided Verification (CAV)*, pages 232–247. Springer-Verlag, 2000.
- [9] L. B. et al. Domain and type enforcement unix prototype. In *USENIX Sec 95*.
- [10] J. T. Griffin and S. Jha. Detecting manipulated remote call streams. In *USENIX Security Symposium*, 2002.
- [11] G. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 1997.
- [12] C. Karlof, N. Sastry, and D. Wagner. Tinysec: Link layer encryption for tiny devices. In <http://www.cs.berkeley.edu/nks/tinysec>.
- [13] C. Karlof and D. A. Wagner. Secure routing in wireless sensor networks. In *First IEEE International Workshop on Sensor Network Protocols, May 2003*.
- [14] C. M. Krishna, I. Koren, A. Ganz, and C. A. Moritz. Security tradeoffs in nest. In *Presentation in: http://arts.ecs.umass.edu/hkumar/presentations.htm*.
- [15] Z. Liang, V. N. Venkatakrishnan, and R. Sekar. An application transparent approach for executing untrusted programs. In *ACSAC*, 2003.
- [16] S. McCanne and V. Jacobson. The BSD packet filter. In *USENIX 93*.
- [17] Misc. Tinyos: A component-based os for the networked sensor regime. In <http://webs.cs.berkeley.edu/tos/>.
- [18] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995.
- [19] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *USENIX Security Symposium*, 99.
- [20] R. Szewczyk, V. Wen, D. Culler, and D. Tygar. Spins: Security protocols for sensor networks. In *Journal (WINE), September 2002*.
- [21] P. Uppuluri. *Intrusion Prevention Using Behavior Specifications*. PhD thesis, SUNYSB, 03.
- [22] P. Uppuluri and R. Sekar. Experiences with specification-based intrusion detection. In *RAID*, 01.
- [23] D. A. Wagner. Janus: Confinement of untrusted applications. In *TR CSD-99-1056*.
- [24] Q. Xue and A. Ganz. Runtime security composition for sensor network (securesense). In *Multimedia Networks Lab, ECE Dept, Univ. of Massachusetts Amherst, 2003*.
- [25] S. Zhu, S. Setia, and S. Jajodia. Leap. In *10th ACMCCS*, pages 62–72, 2003.