MoteAODV – An AODV Implementation for TinyOS 2.0

Werner Backes and Jared Cordasco

Stevens Institute of Technology
Castle Point on Hudson, Hoboken, NJ 07030 USA
Email: {wbackes,jcordasc}@cs.stevens.edu

Abstract Reliable, inexpensive, on-the-fly networks can be established using mobile ad-hoc network (MANET) technology. Such networks can be used in a wide variety of scenarios ranging from connecting powerful computers to connecting so-called "smart dust" devices. However, the core of MANET technology, the routing protocols, were never designed for the extremely low power devices that are desired for today's applications. In this paper, we propose a new implementation of the Ad-hoc On-Demand Distance Vector routing protocol (AODV), named MoteAODV, that addresses this shortcoming. In addition, we also introduce an implementation, MoteAODV+AES, which allows for the addressing of security concerns in routing protocols on such limited devices. We also provide performance benchmarks and a comprehensive security analysis of our proposed solutions.

1 Introduction

Mobile ad-hoc networks (MANETs) allow for wireless communication without the need for a central infrastructure, using low-cost resource constrained devices in sometimes hostile environments. Even though devices in MANETs may move around considerably, they can still communicate with each other. At the root of these unique properties are the routing algorithms that make communication possible between MANET nodes that are possibly out of direct radio range. As a result, wireless ad-hoc networks are becoming increasingly popular and applications of MANETs are manifold.

Initial proposals for routing protocols in MANETs such as Dynamic Source Routing (DSR) [13], Destination-Sequenced Distance-Vector routing (DSDV) [21], and Ad-hoc On-Demand Distance Vector routing (AODV) [22] do not consider an implementation on extremely resource constrained devices such as sensor motes [6]. In fact, much of the hardware currently under consideration for wireless sensor networks (a special form of MANET) was not yet developed when these proposals were set forth. Since these mote devices were first developed, protocols like Trickle [14] and the Collection Tree Protocol (CTP) [10] appeared. While these protocols allow the motes to carry out some simple tasks in simplified scenarios, these protocols do not provide the robustness and flexibility of protocols designed for more general MANETs. In order to use motes for these more

advanced purposes, more general routing protocols are required. When deciding which routing protocol to use, there are several considerations. First, one must choose between proactive and reactive protocols. In proactive protocols, each node maintains a route to all other nodes in the network. This often involves exchanging complete routing tables during routing updates. Reactive protocols, however, only maintain routes to nodes for which there is active communication. When communication with a new node is needed, the node performs route discovery for that destination. While proactive protocols allow for faster initiation of new connections (no waiting for route discovery), they require significantly more bandwidth for route maintenance. In addition, routing table storage requirements grow in proportion to the size of the network. Therefore, for resource constrained devices it is best to use a reactive protocol.

Additional consideration has to be given to the use of source-routed or distance-vector protocols. In source-routed protocols, a node knows the complete route its data will travel to arrive at the destination. In a distance-vector protocol, however, the node only knows the next hop. With the source-routing approach, a node has more control over the path its data travels, but it also has to perform a new route discovery whenever the route breaks due to node mobility or failure. Distance-vector protocols are designed to route around failures without notifying the source if at all possible. This provides greater flexibility and will often result in fewer control messages being sent through the network. These two advantages of the distance-vector approach make it ideal for use on resource constrained devices.

Thus, research into implementing more advanced protocols should focus on reactive distance-vector protocols such as AODV. Prior work in this context includes TinyAODV [29], an implementation of AODV for version 1.x of the TinyOS operating system. TinyAODV does not support timer-based events such as `HELLO` messages or expiring routes. Our first contribution in this paper is a new implementation of AODV for TinyOS 2.0 called MoteAODV, that addresses these shortcomings. In our explanation of MoteAODV, we detail the issues that must be addressed when implementing the AODV protocol on these resource limited mote devices, specifically the Crossbow TelosB mote [27].

Original MANET routing protocols were not designed to address security issues. Subsequent protocols were introduced [3, 4, 11, 12, 15, 19, 23, 24, 30] to address these concerns. In particular, SAODV [30] suggests the use of public key cryptography to secure AODV routing. However, due to their resource constraints, mote devices are not capable of using complex public key signature schemes to secure the routing messages, as proposed in [30]. Instead, it is necessary to use hash functions and symmetric key encryption to provide message integrity and prevent eavesdropping by outside attackers. As our second contribution in this paper we describe our new implementation of the Advanced Encryption Standard (AES) on the TelosB motes with a focus on improving performance while creating an implementation that can run alongside our MoteAODV implementation within the memory and storage restrictions of the devices.

The remainder of the paper is organized as follows. Section 2 gives a brief overview of the AODV protocol. Section 3 then details our implementation for the TelosB mote. In Section 4 we describe MoteAODV+AES—a version of MoteAODV secured with pre-shared keys. We then provide protocol level results in Section 5. Section 6 gives a security analysis of the new protocol using a novel attacker model. Finally, we conclude and describe future work in Section 7.

2 AODV

AODV is an on-demand (reactive), distance-vector routing protocol for ad-hoc networks. Each node only maintains routes for nodes that it actively communicates with. When the node wishes to initiate communication with a node for which it does not currently have a valid route, it must perform route discovery.

The first step in route discovery for a particular destination D involves the source node S broadcasting a Route Request packet (`RREQ`) to its neighbors. In the `RREQ`, S specifies a destination sequence number (DSN) which indicates how "fresh" a route is desired. Neighbors rebroadcast the `RREQ` successively until it either reaches a node having a DSN for D that is fresh enough (greater than or equal to the one in the `RREQ`), or it reaches D. In order to speed up route discovery and minimize control traffic in the network, intermediate nodes must respond to `RREQ`s if they have a fresh enough route.

As the `RREQ` is forwarded, each node keeps a record of the `RREQ-ID` to avoid loop resending of the same packet. The node also prepares a reverse route to S based on the `RREQ`. This reverse route is then used for unicasting the Route Reply (`RREP`) after it is generated at either an intermediate node, or at D. This establishes the route bi-directionally between S and D[1]. A `RREP` acknowledgment (`RREP-ACK`) is optionally sent from S to D.

In addition to the `RREQ`, `RREP`, and `RREP-ACK` messages, `HELLO` messages are recommended for link-state monitoring. Nodes periodically broadcast `HELLO` messages to inform their neighbors that they are within range of direct radio broadcasts. After a specified number of `HELLO` messages from a neighbor are missed, a node sends Route Error (`RERR`) messages to all its neighbors who had established routes forwarding data through the missing node. To increase performance, the node may attempt local route repair prior to the sending of `RERR` messages. For more detailed information on the AODV protocol see [22].

3 MoteAODV

MoteAODV—our implementation of the AODV protocol—is aimed at low-power sensor devices such as the Crossbow TelosB motes [27] or the Crossbow Mica2 motes [17]. The main issues in programming or working with this kind of device are the limited amounts of main memory (RAM), programming flash, and the

[1] If an intermediate node responds, a gratuitous `RREP` sent to D is also required for a bi-directional route.

low-power CPU. The various combinations[2] thereof make it difficult to state definitively which platform is the most resource constrained. Implementation choices, e.g., ability to trade memory usage for code size, further complicate the issue. Our implementation is designed for the Crossbow TelosB motes. We have decided to base our AODV implementation on TinyOS 2.0 [29] instead of version 1.x due to the limited support in 1.x for our hardware at this point in time. In particular, the limitations of TinyAODV [29] (which is part of the TinyOS 1.x distribution) led us to re-implement AODV rather than porting TinyAODV to TinyOS 2.0. In general, TinyAODV does not support any timer-based events of AODV protocol. In particular, its limitations include never expiring routes and the lack of `HELLO` messages to observe link state changes. In contrast, the MoteAODV is optimized for sensor devices with a low-power CPU and limited memory resources and designed to support the timer-based features of AODV. In this section, we describe how the timer-based events can be implemented with little overhead, by balancing memory and computational efficiency in our data structures which are optimized for the most commonly performed operations. In addition, MoteAODV also supports the use of micro-acknowledgments to detect broken links as an alternative to `HELLO` messages.

3.1 Timer-Based Events

In this section we introduce and detail the techniques we used to implement and emulate the timer-based mechanisms for handling expiring routes and `HELLO` messages as defined in the AODV standard [20]. These features, which are not implemented by TinyAODV, are designed to keep the routing information up-to-date and therefore allow the AODV protocol to recover from routing changes due to mobility or failure in a more timely fashion. We do not discuss the AODV protocol in detail here, but rather explain the techniques and methods used in MoteAODV to support these important features with minimal overhead. In

Parameter Name	Proposed Value	Timer Ticks
`ACTIVE_ROUTE_TIMEOUT`	3000ms	3
`MY_ROUTE_TIMEOUT`	2 · `ACTIVE_ROUTE_TIMEOUT`	2 · `ACTIVE_ROUTE_TIMEOUT`
`HELLO_INTERVAL`	1000ms	1
`NODE_TRAVERSAL_TIME`	40ms	1
`NET_TRAVERSAL_TIME`	2 · `NODE_TRAVERSAL_TIME` ·`NET_DIAMETER`	3
`PATH_DISCOVERY_TIME`	2 · `NET_TRAVERSAL_TIME`	2 · `NET_TRAVERSAL_TIME`

Table 1. Conversion of proposed configuration parameters

order to do so, we need to adapt the AODV protocol as the computational overhead caused by excessive use of timers would overload the low-power CPU of the

[2] TelosB motes have 10kb of main memory and 48kb of programming flash with a 16 bit CPU, whereas Mica2 motes have only 4kb of main memory, but 128kb of programming flash with an 8 bit CPU

TelosB motes. The solution is to replace separate timers for individual events with periodic timers. An advantage of using periodic timers is the option of extending the interval in between timer ticks in case a higher number of computations has to be performed, e.g., for an implementation of the TAODV protocol [16], or for debugging purposes. As a result, the minimum time interval that can be measured is also determined by the time in between these ticks. Therefore, all AODV timings have to be adjusted. In our setting, we are using timers recurring every 1000 ms as a basis for the conversion of the configuration parameters proposed in RFC 3561 [20]. Table 3.1 gives some examples for converting time intervals into timer ticks. It is important to note that we compute the proposed time interval for `NODE_TRAVERSAL_TIME` in our conversion to timer ticks for `NET_TRAVERSAL_TIME` using a `NET_DIAMETER` of 35. All other parameters have been converted accordingly. For reasons of modularity, we are using two periodic timers in our implementation of the AODV protocol, one timer for updates to the routing and RREQ tables and another timer for other AODV-related events, including `HELLO` messages. These recurring timers[3] are implemented as follows:

Routing table timer:

```
(1)  foreach (rt ∈ routing table) do
(2)    rt.lifetime = rt.lifetime − 1
(3)    if (rt.lifetime == 0) then
(4)      delete rt from routing table
(5)    else
(6)      if (rt.lifetime < DELETE_PERIOD) then
(7)        rt.state = ROUTE_ENTRY_INVALID
(8)  foreach (rq ∈ RREQ table) do
(9)    rq.lifetime = rq.lifetime − 1
(10)   if (rq.lifetime == 0) then
(11)     delete rq from RREQ table
```

AODV timer:

```
(1)  foreach (rp ∈ RREP wait table) do
(2)    rp.lifetime = rp.lifetime − 1
(3)    if (rp.lifetime == 0) then
(4)      delete rp from RREP wait table
(5)  foreach (nb ∈ neighbor table) do
(6)    nb.hello_miss = nb.hello_miss + 1
(7)    if (nb.hello_miss ≥
         ALLOWED_HELLO_LOSS) then
(8)      report broken link to neighbor nb
(9)      delete nb from neighbor table
```

It is important to note that the periodic timers only decrease the lifetime or increase the counters. Other methods like the processing of `HELLO` messages reset or initialize the counters, e.g., the missed messages counter for the neighbor for which the node received a `HELLO` message. The AODV timer checks whether the lifetime or the counter have reached a threshold. If so, it notifies the protocol about a broken link in case too many `HELLO` messages from a particular neighbor are lost. The routing table is updated and, according to the AODV protocol, appropriate `RERR` messages are issued. The extended lifetime of a route table entry is defined as the sum of `MY_ROUTE_TIMEOUT` and `DELETE_PERIOD`. This allows us to use a single value to easily determine the validity period of a route table entry and finally remove the entry after the delete period has expired (as defined in the AODV standard). The rate and the number of retries for `RREQ` messages is used as defined in the AODV standard in order to reduce the number of routing messages. The RREQ and RREP tables are introduced to emulate this behavior with recurring timers. We are using the RREQ table to track and limit the `RREQ`s a sender is allowed to issue to a destination. The RREP table tracks the issued `RREQ`s and the time a node waits for the `RREP` message to arrive. These tables are

[3] We are using the conversion table with a 1000 ms to 4000 ms increase in the timer interval in practice.

checked or initialized when a `RREQ` is to be issued and modified when a RREP message is processed.

3.2 Efficient Data Structures

Using efficient data structures plays a major role in minimizing computational overhead. To implement the aforementioned tables, used to minimize the workload within each timer tick, we introduce a simple data structure. Our new data structure allows a fast enumeration of all table entries and an efficient delete of specific table entries. The limited amount of memory on the motes forces us to limit the maximum amount of table entries and therefore makes a dynamic structure, like a double linked list, impractical. The size of each table is decided at compile time and can be adjusted by modifying the appropriate constants in the source code. Figure 1 shows an initialized, but empty table. The data

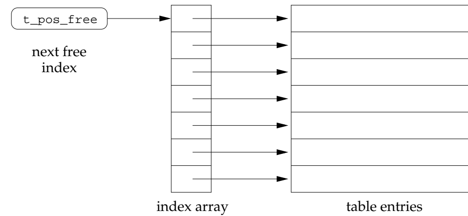

Figure 1. Data structure for tables

structure uses an index array t_array to access the actual table entries $t_entries$ and the position t_pos_free in the index array where one can find the next unused table entry. This allows us to efficiently implement the functionality needed by the periodic timers, namely, to enumerate all used table entries with little overhead and to delete a specific table entry in constant time. In addition, we are also able to append an entry to the table in constant time. These constant time table operations are implemented as follows:

Delete table entry at position d:

(1) $t_pos_free = t_pos_free - 1$
(2) $tmp = t_array[d]$
(3) $t_array[d] = t_array[t_pos_free]$
(4) $t_array[t_pos_free] = tmp$

Append table entry e:

(1) $tmp = t_array[t_pos_free]$
(2) $t_entries[tmp] = e$
(3) $t_pos_free = t_pos_free + 1$

However, focusing on these table operations in order to minimize the load for the periodic timers also has disadvantages. The delete operation, for example, changes the order of the table entries. Other operations such as inserting an entry at a specific position within the table take linear time because the order of the other table entries needs to be preserved.

3.3 Hello Messages vs. Micro-Acknowledgments

The AODV standard [20] proposes `HELLO` messages as one method to monitor changes of the link state of neighbor nodes. In general, `HELLO` messages are the

preferred method of detecting route errors caused by the mobility or failure of network nodes. However, sending out HELLO messages on a constant basis not only increases the power usage (therefore decreasing the lifetime of the battery powered mote), but it also increases the level of exposure of the entire network. In some scenarios (mostly military) this increased risk of being detected is not acceptable. In order to address this challenge, we have developed a new transparent micro-acknowledgment mechanism which eliminates the need for HELLO messages. In order to implement the micro-acknowledgment functionality, we developed a new programming interface called PacketSend which we derived from the built-in TinyOS interface AMSend. The addition of the sendError event allows us to signal the sender of a message that the message could not be delivered to its destination. The new extended interface PacketSend provides the following methods:

```
command error_t send(am_addr_t addr, message_t* msg, uint8_t len)
command error_t sendAck(am_addr_t addr)
command error_t cancel(message_t* msg)
event void sendDone(message_t* msg, error_t error)
event void sendError(am_addr_t dest_addr)
command uint8_t maxPayloadLength()
command void* getPayload(message_t* msg)
```

The micro-acknowledgment mechanism works as follows. For each message a source sends to its destination using the new PacketSend interface, the destination sends back an acknowledgment package using the sendAck command. For simplicity we use a different channel for the acknowledgment packets than for the routing or data messages. The PacketSend interface keeps a table entry with a packet counter for each destination the source sends messages to. This counter is increased for every outgoing packet and decreased for every incoming acknowledgment packet the sender receives from the destination. The event sendError is triggered if the packet counter for a destination exceeds a predefined threshold. In some cases packets are lost, e.g., caused by a temporary high load which overflows the message buffers. In order to avoid the buildup of the packet counter over time, due to short-term problems, we use a recurring timer that decreases each packet counter. This mechanism allows us to detect link failures due to mobility or node failures and it is stable enough to handle moderate loss of packets due to short-term factors.

4 Securing MoteAODV Using Pre-Shared Keys

The AODV protocol was not designed with security in mind. MoteAODV therefore, introduces extended routing messages that contain a hash value in order to protect the integrity of the AODV packets. These routing messages (including the hash value) are then encrypted using a pre-shared key. We implemented several hash functions for MoteAODV, including MD5 [25] and SHA-256 [2]. MoteAODV uses as default AES [7, 1] with 128 bit keys in CTR mode [8, 9] to encrypt the messages. It is important to note that the MoteAODV implementation supports AES-128 in software including all block cipher modes of operation

[8, 9]. This gives us great flexibility in terms of both the key management and the actual use of AES. In the following sections we detail the optimizations and give a performance analysis for the MoteAODV implementation of AES-128 for TelosB motes. We then analyze the performance of the MD5 and SHA-256 implementations in MoteAODV.

4.1 AES-128 for TelosB Motes

In this section we present ways to optimize the AES implementation for the 16 bit processor TI MSP430 [18] of the TelosB motes. Rijndael block ciphers [7] in general, or AES in particular, are designed to work with 8 bit processors. Optimizations are known for the typical 32 bit processors used in today's PCs [26]. Unfortunately, these optimizations cannot be applied to the TelosB mote implementation due to the limited instruction set of the 16 bit TI MSP430 CPU [18]. In addition, the majority of these optimizations for AES-128 use lookup tables which would take up a significant amount of the very limited main memory (10 KB for the TelosB mote). The core of our AES-128 implementation is therefore based on the 8 bit textbook implementation [7] using a limited number of tables to save memory. The lookup tables that are unavoidable are the sbox table for encryption and the inverse sbox table for decryption, which each take up 256 bytes of memory. In order to speedup multiplication, we include another 256 byte lookup table. While traditional improvements would be to extend the lookup tables for the use of 16 bit operations, the limited amount of available memory does not allow this form of optimization. Consequently, access to the lookup tables is limited to 8 bit operations. In most cases, this prevents us from using 16 bit operations efficiently. Using simple source code transformations only, we improve on `AES_key_expansion`[4] and `AES_add_round_key` by using 16 bit operations. Two 8 bit assignment or xor operations can be replaced by one 16 bit operation. In addition, a modification of `AES_mix_columns` using one additional temporary variable gives us an additional improvement on the TelosB mote (see Appendix for details).

Further optimization is possible by using assembler code to re-implement parts of the AES-128 functions. We are able to achieve an additional optimization by making extensive use of the 16 bit capabilities of the TI MSP430 CPU in the TelosB motes. The MSP430 has a very simple but limited instruction set. The majority of register operations take up 1 processor cycle, but accessing memory is expensive and usually takes 3 cycles per memory operation. It is important to note that the number of cycles does not depend on whether it is an 8 bit or a 16 bit operation. The special operation `swpb [register]` which swaps the upper and lower byte of a register can be used to efficiently convert two 8 bit values into a 16 bit value. Some operations can be improved by rearranging the instructions and computations within the function in order to take advantage of the processors capability. In, e.g., `AES_sub_bytes_shift_rows`, the results of the

[4] Function names are chosen similar to the textbook implementation [7].

byte-wise memory access to the sbox table are combined into a 16 bit register that can then be written more efficiently (see Appendix for details).

Minimizing the number of cycles and the amount of registers used, including minor assembler improvements to the key expansion, increases the performance of our implementation of AES-128 significantly. Table 2 shows the average running time for our different AES implementations. The original message used for encryption/decryption in CTR mode was 20 bytes in length. The time was measured using the TinyOS timer module and each operation was performed 5000 times. Our assembler implementation gives us more than 30% reduction

operation	text book	improved	assembler
encrypt block	1.994 ms	1.638 ms	1.300 ms
decrypt block	2.365 ms	1.996 ms	1.686 ms
encrypt/decrypt (CTR)	3.541 ms	2.900 ms	2.377 ms
main memory	1 kb	1 kb	1 kb
program flash	4.5 kb	4.2 kb	4.7 kb

Table 2. Timings and memory usage for AES-128 implementation on TelosB motes

in the running time for the encryption/decryption of a message in CTR mode. Compared to the improved (non-assembler) version, our assembler implementation still results in approximately a 20% reduction in running time. Further improvements for the MSP430 CPU are difficult because, as mentioned above, the CPU's limited instruction set makes a conversion from 8 bit to 16 operations and vice versa difficult. The main memory usage could be reduced by hard-coding the lookup tables using a switch-case statement, but the programming flash usage would increase significantly. This technique could therefore be used for Mica2 motes that have less main memory, but almost three times the programming flash of the TelosB motes.

4.2 Hash Functions MD5 and SHA-256

We implemented the hash functions MD5 [25] and SHA-256 [2] for the TelosB motes in order to integrity protect routing messages. Unlike our AES-128 implementation for the TelosB motes, we only implemented SHA-256 according to the standard [2]. For MD5 we implemented the textbook version [26] and developed a slower, but more compact version, which consumes less programming flash by using a small portion of the main memory. Table 3 shows the main memory and programming flash usage of our MD5 and SHA-256 implementations. The running times in Table 3 are the average time for hashing one block including the time needed for the padding of a message. The performance drop of our compact MD5 implementation can be explained by the increased number of memory access operations. On a TI MSP430 CPU, these operations are significantly slower than register based operations. On the other hand, the faster textbook implementation of MD5 takes up about 20% of the TelosB programming flash.

	MD5 (text)	MD5 (compact)	SHA-256
hash a message	1.582ms	4.703ms	9.163ms
main memory	–	272 byte	256 byte
program flash	9kb	4kb	2 kb

Table 3. Timings and memory usage for the MD5 and SHA-256 implementation on TelosB motes

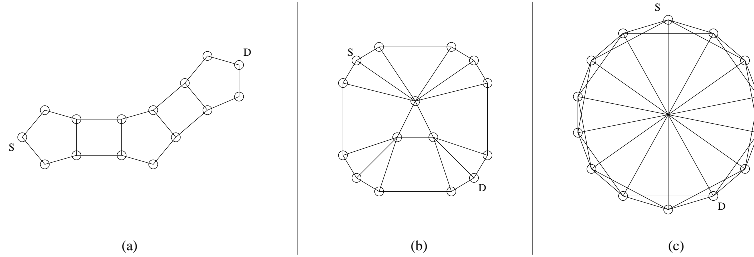

Figure 2. Test networks with (a) low mobility, low range (b) low mobility, high range and (c) high mobility, high range.

5 Overall Protocol Performance

In this section we detail the protocol level performance results we have obtained in our testing. These tests were performed in three different network scenarios. Our first scenario represents the connectivity of nodes in a network with low mobility and short radio range (Figure 2a). The second scenario represents a network of nodes with low mobility and an increased radio range (Figure 2b). Our third testing scenario represents the topology of a network with nodes having a high rate of mobility and a long radio range (Figure 2c). For each of these networks, we test both MoteAODV and MoteAODV+AES. For nodes using MoteAODV+AES to send a packet, a SHA-256 hash value for the packet was computed and appended to the message. This new, larger packet is then encrypted using AES-128 in CTR mode. Upon receiving such a message, a node decrypts the packet and checks the hash value. Packets that do not verify in this way are dropped.

5.1 Route Discovery

The results for successful route discovery are shown in Table 4. These results and all that follow are the results of averaging ten separate measurements. As can be seen in the timings, the addition of the AES and hash computations, along with the increased message size, causes a 2.9-fold increase in discovery time for Scenarios (a) and (b), and a 3.7-fold increase in Scenario (c). The increase in Scenario (c) is caused by the high number of connections for each of the nodes which allows for a high number of possible routes to the destination. The additional computations in MoteAODV+AES, in particular the computation

of the SHA-256 hash value, cause delays within the processing of the routing messages. While in all scenarios there is a significant increase in route discovery time, it still takes less than one second and there is an increase in security as discussed later in Section 6.

Scenario	MoteAODV	MoteAODV+AES
(a)	220.1 ms	642.3 ms
(b)	112.9 ms	333.0 ms
(c)	59.4 ms	223.1 ms

Table 4. Timings for successful route discovery.

5.2 Link Failure Recovery

Earlier we described the use of micro-acknowledgments for the silent detection of link failure. While this may be preferred in certain applications, the AODV standard recommends HELLO messages and as such they are the preferred mechanism for general networks. Therefore, our tests focus on the use of HELLO messages for link failure detection. As mentioned above, if a node fails to receive the required number of HELLO messages it assumes the link is dead. The standard recommends a missed HELLO threshold of five HELLO_INTERVAL periods [20]. However, since our implementation sends HELLO messages less often due to the limited CPU power of the motes, we also tested with a threshold of three periods. The results for detection of link failure are shown in Table 5. These tests were not performed in the three separate scenarios because the overall network topology does not affect the detection of link failure between two individual nodes. As the results show, there is little overhead for MoteAODV+AES with regard to link failure detection. Also, the detection times decrease in direct relation to the size of the threshold which shows that there are no underlying components that would prevent proportionally faster detection when the threshold is decreased.

After link failure is detected, a node must discover an alternate route. Due to

Threshold	Time to detection
5 * HELLO_INTERVAL (MoteAODV)	22.688 sec
5 * HELLO_INTERVAL (MoteAODV+AES)	23.425 sec
3 * HELLO_INTERVAL (MoteAODV)	12.024 sec

Table 5. Times for detection of link failure.

the impact of the topology on the route discovery, specifically the length of the alternate route, we performed different tests for each of the network scenarios presented above. The results are summarized in Table 6. Once again, we see a significant increase in overhead for MoteAODV+AES as in the original route discovery results. Additionally, we see that taking out nodes downstream (i.e., a node that is closer to the destination) results in longer times for alternate route discovery. Taking out a central node in Scenario (b) makes the alternate route significantly longer and therefore increases the time necessary for discovery compared to the initial route discovery. Similarly, disabling all of the cross links in

Scenario	Failure Type	MoteAODV	MoteAODV+AES
(a)	Close to source	160.5 ms	425.7 ms
(a)	Close to dest	209.9 ms	601.0 ms
(b)	Central node	138.2 ms	382.6 ms
(c)	Disable all x-links	138.2 ms	201.5 ms

Table 6. Timings for alternate route discovery after link failure.

Scenario (c) results in a longer alternate route and consequently longer discovery times. The results presented in this section should be considered in light of the security mechanisms used. In particular SHA-256 contributes a significant portion to the MoteAODV+AES overhead (see Table 3). Optimizing the SHA-256 implementation or using a faster hash function will reduce the protocol's overhead considerably.

6 Security Analysis of MoteAODV+AES

In determining suitable use case scenarios for MoteAODV+AES, we must consider the security provided by the protocol. In order to assess the security of MoteAODV+AES, we use the attacker model presented in [5] to briefly identify the necessary attacker capabilities to compromise the protocol. As specified in the model, attackers have two main capabilities which we must account for: communication and computation. We investigate communication capabilities equivalent to an ordinary node (Type I) and with limited sending and long range receiving (Type II). Since we are using a symmetric primitive with a group key, we consider two types of computational capabilities, specifically two categories of attacker knowledge: inside attackers (knowledge of key material) and outside attacker (no knowledge of key material). It is important to note that with multiple colluding attackers only one inside node needs to be compromised or malicious as key material then spreads via collusion.

We first address insider attacks. With the group key, a node can decrypt transmitted messages and generate properly encrypted messages that others will accept as valid. The communication capabilities of the attacker(s) determine(s) the influence they can have with these messages. However, the communication capabilities in no way limit the types of attacks possible as an inside attacker already eliminates all security guarantees and reduces the protocol to the equivalent of MoteAODV. Such and attack can be prevented by tamper-proof and self-erasing motes which prevent nodes from being successfully compromised or disclosing key material, a topic which is outside the scope of the routing protocol and this paper.

Outside attackers are limited by the infeasibility of performing cryptographic operations without the group key. As per [5], this limits their computational capabilities. The only messages that an attacker can generate are messages it has overheard. Therefore, attackers can only perform message replay attacks. However, when an honest node receives a `RREQ` with a `RREQ-ID` it has already seen, it discards the message. Thus, an attacker will have to deliver its relayed

message prior to the delivery of the authentic routing message. An attacker with Type I communication cannot exploit this as its neighbors either received the message at the same time or will receive it in the next hop transmission.

Considering attackers with Type II and higher communication capabilities, it is tempting to declare that since they cannot decrypt the packets they are replaying, they have no way of using them to compromise the protocol. However, it is reasonable to assume that an attacker can make an educated guess as to the contents of the encrypted packets (whether via timing, or due to low amounts of network traffic, or some other means). Such an attacker can take messages and replay them in other areas of the network. If the attacker were to replay a `RREQ` message in an area of the network significantly closer to the destination[5] (prior to the `RREQ`s actual propagation to that area), the destination would receive a `RREQ` with a significantly lower hop count. If the attacker is also able to make an educated guess to identify the corresponding `RREP`, it can perform the replay attack in the opposite direction. After doing so the attacker will have successfully caused the advertisement of a route with a hop count significantly lower than the actual hop count.

Our analysis shows several *possible* attacks on the MoteAODV+AES protocol. The *probability* of such attacks depends on the actual application and the specific attacker. With regard to the packet replay attack, networks with a low level of traffic may allow for easier identification of routing packets, but allow for fewer opportunities to perform the attack. Networks with a higher traffic volume may make identification more difficult, but would open up more opportunities to perform the attack. The size of the network can also have an effect. Small networks have a shorter window of attack as `RREP`s will arrive at the source faster, while larger networks have a larger window due to the increased route length. Lastly, there are no penalties for the attacker if they make an incorrect guess as packets that do not verify are quietly dropped. For evidence of the practicality of this type of attack consider the use of guessed ARP packet replay to break WEP and WPA (TKIP) wireless network keys [28]. All of these factors must be taken into account when deciding if MoteAODV+AES is appropriate for a particular scenario.

7 Conclusion and Future Work

In this paper we introduced MoteAODV, an implementation of the AODV routing protocol for the TinyOS 2.x operating system. Unlike TinyAODV, an implementation of the AODV protocol for an earlier, out-dated version of TinyOS, MoteAODV supports timer-based events. In order to limit the computational overhead, we introduced a new method using recurring timers to implement these features. To minimize the risk of detection of the sensor network, we propose the use of micro-acknowledgment's replacing `HELLO` messages to monitor link state changes. We provided a performance analysis for our AES-128, MD5

[5] in relation to where the message was overheard

and SHA-256 implementation on the Crossbow TelosB motes and detailed the improvements of our AES-128 implementation by taking advantage of the 16 bit capabilities of the TI MSP430 CPU. For Mica2 motes we propose to hard-code lookup tables in order to reduce the main memory usage. Using different network scenarios (different connectivity and mobility) we analyzed the performance of MoteAODV for route discovery and link failure recovery with and without AES-128 for encryption and SHA-256 for the message integrity. Our results show that the overhead for encryption is acceptable and in most cases outweighs the disadvantages of having unprotected routing messages. The performance can be further improved by optimizing our SHA-256 implementation or using a more efficient hash function. Finally we provide an assessment of the security of MoteAODV+AES using a novel attacker model.

Future work includes reducing the overhead caused by the hash functions by optimizing our implementations for the 16 bit processor of the TelosB motes. We are also in the process of implementing the trust-based routing protocol TAODV using the techniques and methods of MoteAODV.

Acknowledgments

This work is supported in part by the US Army, Picatinny, under Contract No. W15QKN-05-D-0011.

References

1. Announcing the Advanced Encryption Standard (AES). 2001. Federal Information Processing Standard 197.
2. Secure Hash Standard. 2002. Federal Information Processing Standard 180-2.
3. C. N.-R. Baruch Awerbuch, David Holmer and H. Rubens. An On-Demand Secure Routing Protocol Resilient to Byzantine Failures. In *ACM Workshop on Wireless Security (WiSe)*, Atlanta, Georgia, September 2002.
4. S. Buchegger and J.-Y. L. Boudec. Nodes Bearing Grudges: Towards Routing Security, Fairness, and Robustness in Mobile Ad Hoc Networks. In *Proceedings of the Tenth Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 403 – 410, Canary Islands, Spain, January 2002. IEEE Computer Society.
5. J. Cordasco and S. Wetzel. An Attacker Model for MANET Routing Security. In *Proceedings of The 2nd ACM Conference on Wireless Network Security (WiSec '09)*, pages 87–93, 2009.
6. Crossbow Wireless Module Portfolio. `http://www.xbow.com/Products/productdetails.aspx?sid=156`.
7. J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.
8. M. Dworkin. Recommendation for Block Cipher Modes of Operation - Methods and Techniques. *NIST Special Publication 800-38A*, 2001.
9. M. Dworkin. Recommendation for block cipher modes of operation: The CCM mode for authentication and confidentiality. *NIST Special Publication 800-38C*, 2004.

10. R. Fonseca, O. Gnawali, K. Jamieson, S. Kim, P. Levis, and A. Woo. TEP 123: Collection Tree Protocol. Technical Report 123, 2006. `http://www.tinyos.net/tinyos-2.x/doc/`.
11. Y. Hu, D. Johnson, and A. Perrig. SEAD: Secure Efficient Distance Vector Routing for Mobile Wireless Ad Hoc Networks. *Ad Hoc Networks*, I:175–192, 2003.
12. Y.-C. Hu, A. Perrig, and D. B. Johnson. Ariadne:: A Secure On-Demand Routing Protocol for Ad Hoc Networks. In *MobiCom '02: Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, pages 12–23, New York, NY, USA, 2002. ACM Press.
13. D. B. Johnson and D. A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. In *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.
14. P. Levis, N. Patel, S. Shenker, and D. Culler. Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *In Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–28, 2004.
15. S. Marti, T. J. Giuli, K. Lai, and M. Baker. Mitigating Routing Misbehavior in Mobile Ad Hoc Networks. In *Mobile Computing and Networking*, pages 255–265, 2000.
16. K. Meka, M. Virendra, and S. Upadhyaya. Trust Based Routing Decisions in Mobile Ad-hoc Networks. In *Proceedings of the Workshop on Secure Knowledge Management (SKM 2006)*, 2006.
17. Mica2 Mote Product Details. `http://www.xbow.com/Products/productdetails.aspx?sid=174`.
18. TI MSP430 Microcontroller. `http://www.ti.com/msp430`.
19. P. Papadimitratos and Z. Haas. Secure Routing for Mobile Ad Hoc Networks. In *Proceedings of SCS Communication Networks and Distributed Systems Modeling and Simulation Conference (CNDS)*, 2002.
20. C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. July 2003. RFC 3561.
21. C. Perkins and P. Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In *ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications*, pages 234–244, 1994.
22. C. Perkins and E. Royer. Ad-Hoc On-Demand Distance Vector Routing. In *MILCOM '97 Panel on Ad Hoc Networks*, 1997.
23. A. Perrig, R. Canetti, D. Tygar, and D. Song. The TESLA Broadcast Authentication Protocol. In *Cryptobytes, Volume 5, No. 2*, pages 2–13. RSA Laboratories, 2002.
24. A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar. SPINS: Security Protocols for Sensor Networks. In *Mobile Computing and Networking*, pages 189–199, 2001.
25. R. Rivest. The MD5 Message-Digest Algorithm. April 1992.
26. B. Schneier and W. Diffie. Applied cryptography: protocols, algorithms, and source code in C. 1996.
27. TelosB Mote Product Details. `http://www.xbow.com/Products/productdetails.aspx?sid=252`.
28. E. Tews, R.-P. Weinmann, and A. Pyshkin. chapter Breaking 104 Bit WEP in Less Than 60 Seconds, pages 188–202. Lecture Notes in Computer Science. Springer, 2008.
29. TinyOS Webpage. `http://www.tinyos.net/`.

30. M. G. Zapata and N. Asokan. Securing Ad Hoc Routing Protocols. In *WiSE '02: Proceedings of the ACM workshop on Wireless security*, pages 1–10, New York, NY, USA, 2002. ACM Press.

Appendix

Comparison of the textbook and improved implementation of `AES_mix_columns`.

AES_mix_columns (textbook)

```
(1)  t = st[0] ∧ st[1] ∧ st[2] ∧ st[3]
(2)  u = st[0]
(3)  st[0] ∧= AES_times_02(st[0] ∧ st[1]) ∧ t
(4)  st[1] ∧= AES_times_02(st[1] ∧ st[2]) ∧ t
(5)  st[2] ∧= AES_times_02(st[2] ∧ st[3]) ∧ t
(6)  st[3] ∧= AES_times_02(st[3] ∧ u) ∧ t
```

AES_mix_columns (improved)

```
(1)  u = st[3] ∧ st[0]
(2)  t = (v=st[0] ∧ st[1]) ∧ (w=st[2] ∧ st[3])
(3)  st[0] ∧= AES_times_02(v) ∧ t
(4)  st[1] ∧= AES_times_02(st[1] ∧ st[2]) ∧ t
(5)  st[2] ∧= AES_times_02(w) ∧ t
(6)  st[3] ∧= AES_times_02(u) ∧ t
(7)  ...
```

Assembler optimized version of `AES_mix_columns`.

AES_mix_columns (assembler)

```
(1)  mov @%0, r10     // st[0], st[1] into r10
(2)  mov 2(%0), r11   // st[2], st[3] into r11
(3)  mov r10, r12
(4)  mov r11, r13
(5)  swpb r12         // r12 is r10 swapped
(6)  swpb r13         // r12 is r11 swapped
(7)  mov r10, r14     // compute u and
(8)  xor r13, r14     // st[1] xor st[2]
(9)  xor r10, r12     // compute v and w
(10) xor r11, r13     // v in r12, w in r13
(11) mov r12, r15     // compute t (16 bit)
(12) xor r13, r15
(13) mov.b r14, r9
(14) add %1, r9
(15) mov.b @r9, r9    // compute times_02(u)
(16) swpb r9
(17) mov.b r13, r7
(18) add %1, r7
(19) mov.b @r7, r7    // compute times_02(w)
(20) add r7, r9       // combine to 16 bit
(21) xor r15, r9      // xor with t
(22) xor r9, r11      // xor with st[2],st[3]
(23) swpb r14
(24) mov.b r14, r7
(25) add %1, r7
(26) mov.b @r7, r9    // times_02(st[1] xor st[2])
(27) swpb r9
(28) mov.b r12, r7
(29) add %1, r7
(30) mov.b @r7, r7    // compute times_02(v)
(31) add r7, r9       // combine to 16 bit
(32) xor r15, r9      // xor with t
(33) xor r9, r10      // xor with st[2],st[3]
(34) mov r10, @%0
(35) mov r11, 2(%0)
(36) ...
```

Assembler optimized version of `AES_sub_bytes_shift_rows`.

AES_sub_bytes_shift_rows

```
(1)  mov.b @%0, r11      // read st[0]
(2)  mov.b 5(%0), r15    // read st[5]
(3)  add %1, r11
(4)  add %1, r15
(5)  mov.b @r11, r11     // compute sbox[st[0]]
(6)  mov.b @r15, r15     // compute sbox[st[5]]
(7)  swpb r15
(8)  add r15, r11        // combine to 16 bit
(9)  mov.b 4(%0), r12    // read st[4]
(10) mov.b 9(%0), r15    // read st[9]
(11) add %1, r12
(12) add %1, r15
(13) mov.b @r12, r12     // compute sbox[st[4]]
(14) mov.b @r15, r15     // compute sbox[st[9]]
(15) swpb r15
(16) add r15, r12        // combine to 16 bit
(17) mov.b 8(%0), r13    // read st[8]
(18) mov.b 13(%0), r15   // read st[13]
(19) add %1, r13
(20) add %1, r15
(21) mov.b @r13, r13     // compute sbox[st[8]]
(22) mov.b @r15, r15     // compute sbox[st[13]]
(23) swpb r15
(24) add r15, r13        // combine to 16 bit
(25) mov.b 12(%0), r14   // read st[12]
(26) mov.b 1(%0), r15    // read st[1]
(27) add %1, r14
(28) add %1, r15
(29) mov.b @r14, r14     // read sbox[st[12]]
(30) mov.b @r15, r15     // read sbox[st[1]]
(31) swpb r15
(32) add r15, r14        // combine to 16 bit
(33) mov r11, @%0        // store 16 bit results
(34) mov r12, 4(%0)
(35) mov r13, 8(%0)
(36) mov r14, 12(%0)
(37) ...
```